Alma Mater Studiorum · Università di Bologna

Dottorato di Ricerca in
COMPUTER SCIENCE AND ENGINEERING

Ciclo XXXV

Settore Concorsuale: 01/B1 INFORMATICA

Settore Scientifico Disciplinare: INF/01 INFORMATICA

TEACHING INFORMATICS TO NOVICES: BIG IDEAS AND THE NECESSITY OF OPTIMAL GUIDANCE

Presentata da:
MARCO SBARAGLIA

Coordinatore Dottorato:
Chiar.ma Prof.ssa
ILARIA BARTOLINI

Supervisore:
Chiar.mo Prof.
SIMONE MARTINI

Esame finale anno 2023

Contents

Abstract

This thesis reports on the two main areas of our research: introductory programming as the traditional way of accessing informatics and cultural teaching informatics through unconventional pathways.

The research on introductory programming aims to overcome challenges in traditional programming education, thus increasing participation in informatics. Improving access to informatics enables individuals to pursue more and better professional opportunities and contribute to informatics advancements. We aimed to balance active, student-centered activities and provide optimal support to novices at their level. Inspired by Productive Failure and exploring the concept of notional machine, our work focused on developing Necessity Learning Design, a design to help novices tackle new programming concepts. Using this design, we implemented a learning sequence to introduce arrays and evaluated it in a real high-school context. The subsequent chapters discuss our experiences teaching CS1 in a remote-only scenario during the COVID-19 pandemic and our collaborative effort with primary school teachers to develop a learning module for teaching iteration using a visual programming environment.

The research on teaching informatics principles through unconventional pathways, such as cryptography, aims to introduce informatics to a broader audience, particularly younger individuals that are less technical and professional-oriented. It emphasizes the importance of understanding informatics's cultural and scientific aspects to focus on the informatics societal value and its principles for active citizenship. After reflecting on computational thinking and inspired by the big ideas of science and informatics, we describe our hands-on approach to teaching cryptography in high school, which leverages its key scientific elements to emphasize its social aspects. Additionally, we present an activity for teaching public-key cryptography using graphs to explore fundamental concepts and methods in informatics and mathematics and their interdisciplinarity. In broadening the understanding of informatics, these research initiatives also aim to foster motivation and prime for more professional learning of informatics.

Introduction

This thesis reports the results of my research as a doctoral student within the University of Bologna group on informatics education. I brought years of experience as a specialized high school informatics teacher, and this small yet strong group has allowed me to grow scientifically as a researcher in informatics education, professionally as an educator, and as a person.

The informatics CS1 for math majors was one of the first opportunities to research introductory programming, the outcomes of which are mainly related to using information technology to mitigate the limitations of remote-only education that the COVID pandemic imposed. In that course, which we teach every year, the intuition behind *Necessity Learning Design*, our original learning design, came to life. Our sensibilities as educators led us to believe that we could create educational conditions for students to feel the necessity of the programming concept planned immediately afterward in the course. Such a necessity could have sustained students' motivation and better prepared them for learning.

The scientific context in which we have developed this idea is a "balanced" constructivism. Our intention (supported by scientific, practical, and also ethical reasons) to propose student-centered experiences that are as autonomous as possible must be balanced against the need for students to develop essential learnings without being exposed, as novices still lacking the tools to be truly autonomous, to "uncontrolled" failures.

Within this context, *notional machines*, educational devices to support learning programming, were a "thinking device" for us, a research tool whose flexibility (and independence from implementation constraints) enabled us to imagine the *necessity mechanism* in practice. Such a mechanism could provide the driving force that sustained an ideal progression of notional machines that could gradually become more and more complete and correspond to the actual models of computation to be taught.

The necessity mechanism is at the heart of (and gives the name to) the learning design we developed to support novice students when a new programming concept is introduced. However, this mechanism, despite being particularly suited to support motivation and understanding of essential learning in programming, is actually of more general application. Indeed, the progression of cryptosystems we developed to teach cryptography principles in Italian high schools to young students unfamiliar with informatics and programming is nothing more than a progression of notional machines of such cryptosystems. They are educational abstractions built ad hoc to expose specific concepts and hide technical aspects that are too difficult or irrelevant at that point in the learning path. This progression is always sustained by the

necessity of solving a specific limit of a particular cryptosystem (e.g., Caesar's vulnerability to frequency-based attacks) to push students into wanting to know and understand the next cryptosystem. In this course, too, we tried to balance open, independent activities with the appropriate level of scaffolding that would allow students to enter their zone of proximal development as much as possible throughout the learning path (being continually adapted along the way).

On the other hand, the cryptography course shows the second heart of our research work during my doctoral path, that of cultural teaching of informatics, which is broader and more ecumenical than the traditional programming path. Indeed, learning introductory programming, the more traditional access to our scientific field, is far too specialized and technical for most people (even when declined for novices). Data tell us that about half of the programming students encounter serious learning problems, often resulting in dropout and hostility to informatics. Moreover, most students may never be in a situation to take a programming course since informatics is not yet a compulsory school subject in most educational systems around the world. On the other hand, a basic understanding of (some of) the fundamental principles of informatics from a cultural and citizenship perspective (thus not immediately paying the price for its more challenging technical and scientific components) may enable many more people to understand the informatics value (per se and interdisciplinary) and ways and to focus on the impacts of informatics on today's society and everyone's lives. The cultural perspective of informatics is essential today for exercising informed and active citizenship in our digital society. Besides, earlier exposure to core informatics principles improves the chances of success of subsequent, more traditional introductory programming courses.

This different and complementary perspective of teaching informatics to novices also allowed us to clarify our view on computational thinking, which we intend as a set of ways of thinking and skills that can only be acquired through the study of informatics.

Our research in this area has been greatly influenced by the framework of *big ideas* (of science and especially informatics education), which is an essential navigation system for orientation in the scientific sea of a discipline. Indeed, big ideas enable educators to trace a path in this vast sea and to pay attention (from a learning perspective) to the right signals from students as they navigate. Big ideas also help students place the (necessarily) specific learnings in the coherent perspective of a journey of personal development that should be first and foremost cultural but also scientific.

To summarize in one last note, a defining element of our research is the search for mediation. Mediation through the lens of big ideas to make the vast scientific and technical knowledge of our discipline accessible to a broader audience, proposing culturally relevant learnings without, however, diluting them too much or sacrificing the necessary scientific rigor. Also, mediate between the demands of learning informatics (which often requires converging on necessary, technical, and highly interrelated learnings) and the perspective of novices, whose motivation needs to be supported with student-centered activities and learnings to be built independently. This mediation can be achieved by providing a level of support that enables them not to get lost and to understand the significance of the various learnings in the bigger picture.

Thesis outline

The thesis is structured in four parts; the first two are literature reviews that frame our work, and the third and fourth report the original contributions of our research.

Part I discusses the difficulty of learning how to program and the issue of participation in informatics. It reports what it entails to know how to program and what are the main educational trends to mitigate the difficulty of learning introductory programming. It then motivates the search for optimal guidance, balancing the benefits of active and constructivist methodologies with the necessary support that novices need. Finally, it reviews some active methodologies where this optimal guidance can be sought.

Part II focuses on access to informatics through unconventional pathways that are less technical and more suitable for a broad population. It clarifies the meaning of computation thinking and how it relates to informatics. The central focus is the big ideas of informatics, which can help educators and students recognize the field's culturally and scientifically relevant elements. It then discusses cryptography as a topic relevant in today's society, valuable for exposing informatics principles and ways to a broad audience, and the interdisciplinary nature of informatics. Finally, this chapter discusses interdisciplinarity in STEM education.

Part III, ideally framed in the first literature review, presents our original research on introductory programming, exploring active constructivist-inspired approaches. It describes our learning design for introductory programming and reports on the design and implementation of experimentation of such learning design in a high-school setting. It also reports on teaching programming in emergency remote situations and co-designing a learning module for teaching iteration to students with primary school teachers.

Part IV, ideally framed in the second literature review, presents original research on teaching informatics principles through unconventional pathways, such as cryptography, always using a balanced constructivism approach and the big ideas reference system. The goal is to enable a broader audience to understand the importance of informatics and its interdisciplinary implications while increasing motivation to learn more about it. It reports on our course to teach cryptography big ideas and the teacher training we developed to expose informatics value and reflect on interdisciplinarity between math and informatics.

Overall research goals

Introductory programming. Convinced of the LEM hypothesis, also based on our experiences teaching programming, we wanted to mitigate the negative effects of the *learning edge momentum* on novice students facing introductory programming. In particular, we wondered what a learning model that could support novice students in learning how to program. Then, becoming more concrete, we tried to answer what kind of learning design could support introducing a new elementary programming concept. More generally, in developing learning paths and materials for concrete educational contexts (e.g., CS1 for math majors, teaching iteration in Italian primary schools), we pursued an "optimally-guided" constructivist approach to facilitate learning programming. Adopting both qualitative and quantitative perspectives, we sought to investigate the effects of our choices on students' learning and perceptions.

Despite not being definitive and still evolving, the answers to these questions also influence the research initiatives of the other core of my research, which is teaching informatics as a necessary cultural and citizenship perspective.

Informatics for all. The more general (and still most open-ended) research question that spawned the other research initiatives is inspired by the big ideas of CS education. Specifically, we are still investigating the big ideas of cryptography and, most importantly, how it is possible to use a "big ideas approach" to help young students without informatics knowledge understand the transformative impact of cryptography (and informatics) on our society. From this, derive the following. What kind of "optimally-guided" constructivist learning activities and pathways (see the paragraph before) can be designed to effectively teach those minimal and essential scientific principles that substantiate those ideas and thus promote their effective understanding? More generally, how can teaching informatics concepts from a cultural and citizenship perspective (through less traditional paths than programming, e.g., cryptography education) help non-informatics students understand the value and impact of informatics in our society? How can it affect their predisposition to learn more about informatics?

In trying to answer these questions, especially the educational mindset of always considering a framework of big ideas has also influenced the research initiatives on introductory programming.

Main results

- A theoretical model proposal for teaching introductory programming. The model is inspired by a spiral approach to learning programming and is based on a progression of notional machines driven by a "Productive Failure-like" mechanism to stimulate the necessity of the next notional machine in the progression.

- *Necessity Learning Design*, our original learning design for supporting novices when introducing a new programming concept. Inspired by Productive Failure and aiming at balancing problem-based learning with support, it involves autonomous problem solving before instruction to stimulate the students' necessity for the next programming concept. We also provide an ideal CS1 progression in which we put four detailed ready-to-use "necessity sequences".

- The design and implementation of an high school experimentation of the use of Necessity Learning Design to support the introduction of arrays. The design is partly general (thus reusable) and partly specific to the experiment's unique context. The design includes the experimentation material (programming exercises, teaching materials, learning assessments, and student experience questionnaires). The implementation produced quantitative (e.g., learning assessment grades) and qualitative (e.g., students' open-ended answers, a researchers' journal) data for analysis.

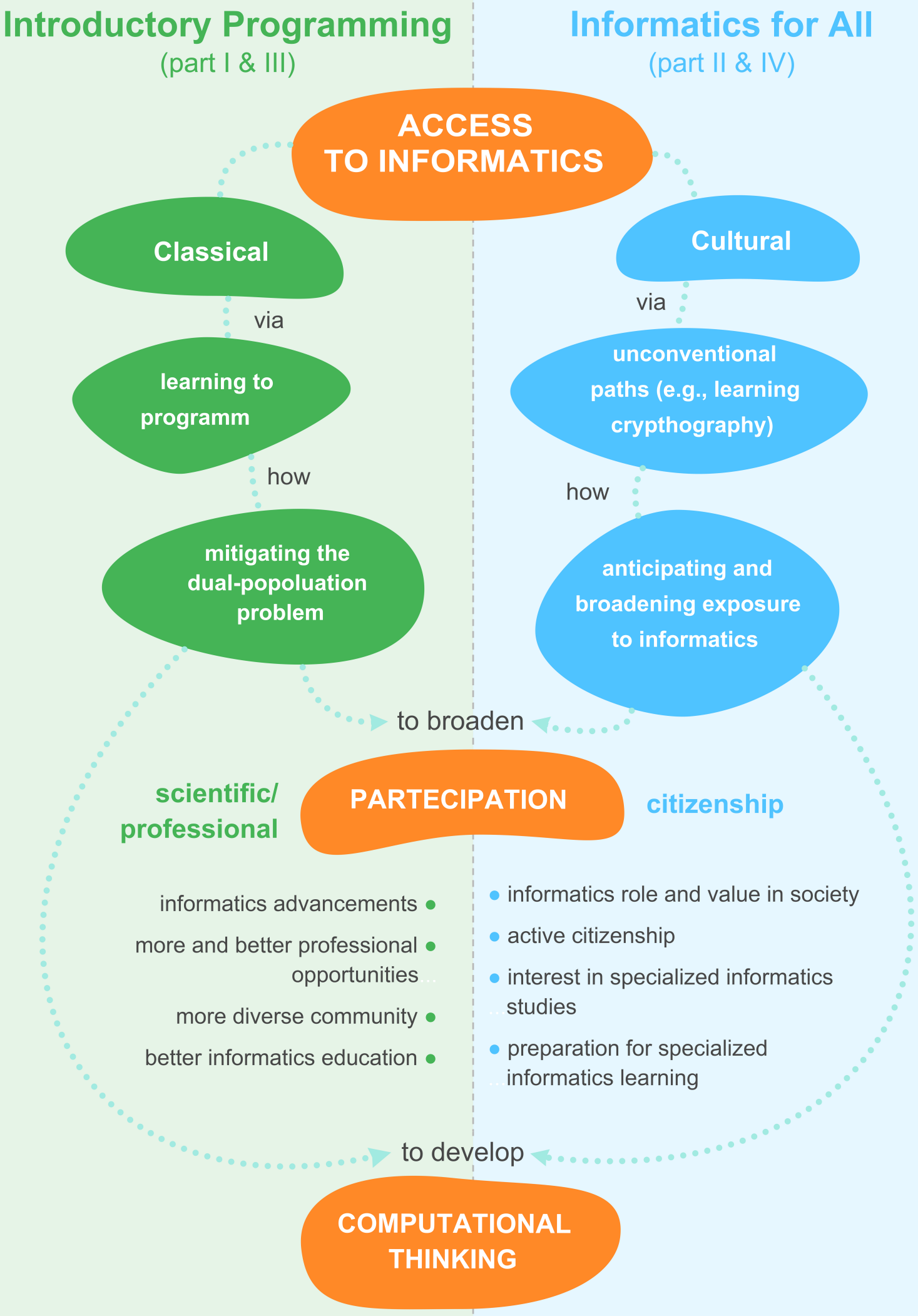
- A CS1 course for math majors redesigned to be taught online during the COVID pandemic. The course design includes a clearly defined learning path, all course

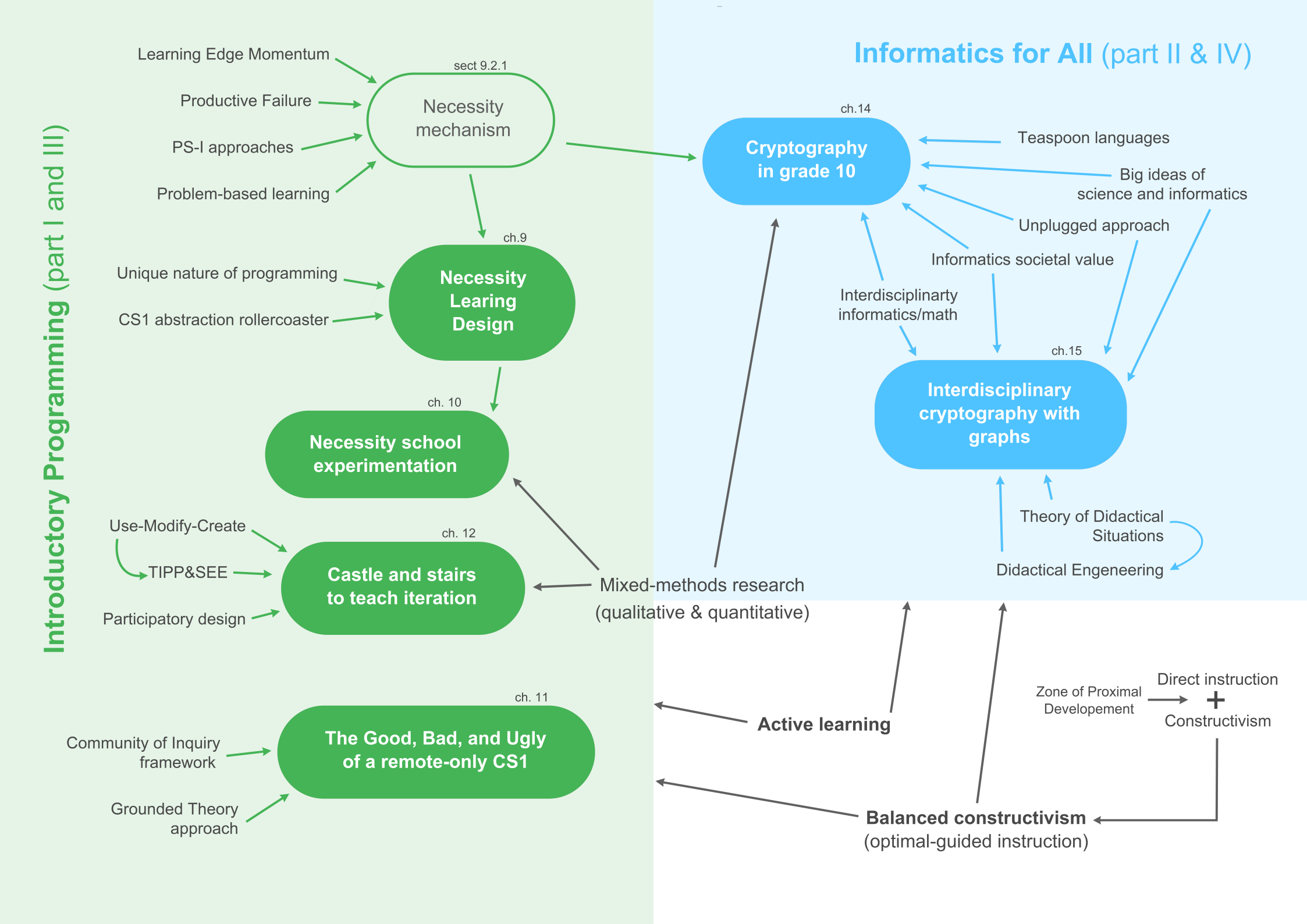
materials, and choices (didactic and technological) made so that the benefits of an in-person experience are not lost.

- A learning module to teach iteration in Italian primary schools using a visual programming environment. The module – the result of a co-design process with primary school teachers – seeks to balance openness and guidance by adopting a *Use-Modify-Create* methodology.
- An introductory course to teach the "big ideas" of cryptography to non-informatics high school students. Students experience cryptographic systems in practice, their characteristics, potential attacks, and the limitations that lead to the necessity for overcoming them. The course includes task-specific programming environments for students to experiment with the cryptosystems and an unplugged activity to teach the Diffie-Hellman protocol. The course was taught three times and produced many data, some of which, analyzed quantitatively and qualitatively, produced significative results.
- A Didactical Situation to introduce public-key cryptography in mathematics and informatics to pre-service STEM teachers. The activity is based on an unplugged activity using graphs and was designed using the Theory of Didactical Situations. Participants need to recognize and apply concepts from mathematics and informatics to develop problem-solving strategies. The activity was implemented and evaluated positively through qualitative analysis.

Thesis maps

Two graphic representations follow in the next two pages to provide a bird-eye view of the thesis. The first concerns the general motivations and goals of our research. The second is about our original contributions, their foundations, and connections.


Introductory Programming
(part I & III)
Informatics for All
(part II & IV)
ACCESS TO INFORMATICS
Classical
via
learning to programm
how
mitigating the dual-popoluation problem
Cultural
via
unconventional paths (e.g., learning crypthography)
how
anticipating and broadening exposure to informatics
to broaden
PARTECIPATION
scientific/ professional
citizenship
informatics advancements
more and better professional opportunities
more diverse community
better informatics education
informatics role and value in society
active citizenship
interest in specialized informatics studies
preparation for specialized informatics learning
to develop
COMPUTATIONAL THINKING


Introductory Programming (part I and III)
Informatics for All (part II & IV)
Learning Edge Momentum
Productive Failure
PS-I approaches
Problem-based learning
sect 9.2.1
Necessity mechanism
ch.9
Necessity Learing Design
Unique nature of programming
CS1 abstraction rollercoaster
ch. 10
Necessity school experimentation
Use-Modify-Create
TIPP&SEE
Participatory design
ch. 12
Castle and stairs to teach iteration
ch. 11
The Good, Bad, and Ugly of a remote-only CS1
Community of Inquiry framework
Grounded Theory approach
ch.14
Cryptography in grade 10
Teaspoon languages
Big ideas of science and informatics
Unplugged approach
Informatics societal value
Interdisciplinarty informatics/math
ch.15
Interdisciplinary cryptography with graphs
Theory of Didactical Situations
Didactical Engeneering
Mixed-methods research
(qualitative & quantitative)
Active learning
Zone of Proximal Developement
Direct instruction
+
Constructivism
Balanced constructivism
(optimal-guided instruction)

Part I

Literature Review – Introductory programming

Introduction to part I

The common thread running through this first part of the literature review is the search for the optimal guidance to teach introductory programming effectively, balancing the benefits of active and constructivist methodologies with the necessary support that novices need, particularly when they face a path that is both introductory (thus rich in essential and interrelated knowledge) and technical (thus requiring great precision), such as that of introductory programming. The ultimate goal is to mitigate the problems suffered by the traditional access to informatics through programming and therefore increase participation for broader and more substantial citizenship (given the apical informatics role in our society), but also the scientific advancements that informatics needs and enable more people to seize its countless professional opportunities. Indeed, access to informatics through introductory programming is the primary access to prepare students for entry into the professional and academic worlds, directly in informatics or in fields where informatics is an indispensable strategic perspective.

Chapter 1

The Role (and Issue) of Programming in Informatics

The first informatics[1] courses - most often called CS1 in higher education - are, for the most part, introductory programming courses, which therefore constitute the typical access to this science.

> The defining characteristic of the computer is its programmability and programming is the essence of computing/informatics. Indeed, computing is much more than programming, but programming – the process of expressing one's ideas and understanding of the concepts and processes of a domain in a form that allows for execution on a computing device without human interpretation – is essential to computing. [Caspersen, 2018, p. 109]

By learning the basics of programming, in fact, one is also introduced to fundamental concepts of informatics itself, such as algorithm, language, automaton, and gains first-hand experience of methods (e.g., modelling and simulation, analysis and evaluation) and mental processes (e.g., abstraction, problem decomposition and modularization) characteristic of this science and typical repertoire of informaticians (for a more detailed overview, see Lodi and Martini [2021], and Lodi [2020a]).

1.1 The early days of informatics and the programming issue

Ever since the early days of informatics, when computers and automatic processing began to spread in the business sector between the 1950s and 1960s, it was realized that programming - initially thought of as a mere manual activity and secondary to computer development - was, in fact, a crucially important, complex and difficult task for many.

[1] In this thesis, we use 'informatics' as a synonym used in Europe for 'computer science' (CS) or 'computing'; we believe its use is etymologically more accurate since informatics contains the root of the words *information* and *automatic*.

> Skilled programmers developed a reputation for creativity and ingenuity, and programming was considered by many to be a uniquely intellectual activity, a black art that relied on individual ability and idiosyncratic style. [...] By the early 1960s, the "problem of programming" had eclipsed all other aspects of commercial computer development. [Ensmenger, 2010, p. 29]

From the beginning, informatics suffered from a shortage of programmers, and this was, first of all, a problem for commercial developments [Gibbs, 1994], but also - as informatics entered the academy and awareness of its scientific status was established - a problem for its scientific advancement (manca REF). The shortage of informaticians has never been resolved [Gibbs, 1994]. As an example, in 2024, there will be about 4.5 million jobs with high salaries in informatics and related fields in the United States alone [U.S. Bureau of Labor Statistics, 2015], but not enough people to occupy them. This shortage is most severe among women and minorities in general. Difficult access to informatics has, in fact, more severe consequences than the lack of informaticians in the labour market and leads to two complementary problems today. A participation problem particularly burdens women, racial/ethnic minorities and persons with learning differences/disabilities (for a comprehensive review of sources, see [Denner and Campe, 2018, sect. 14.1]). Informatics is indispensable for acting on reality and pursuing goals in every activity and discipline, scientific and humanistic. Anyone who is excluded from it, not by choice, misses out on many professional opportunities and personal advancement. If the fragile and underrepresented groups are the most excluded, they miss out on opportunities for socio-economic empowerment, and their marginalization worsens[2]. Not only that, but difficult access to informatics today also leads to an even broader and more critical problem of citizenship. Concepts such as algorithm, language, and automaton provide descriptions of reality that are complementary to those of the other sciences. Grasping these concepts and the basic mechanism of informatics is essential today to act as responsible and conscious citizens, fully engage in civic and social life and express their potential while respecting their inclinations.[3] As for the participation issue, the citizenship problem further aggravates the disadvantaged position of fragile and underrepresented groups.

1.1.1 Is it really (still) difficult to learn programming?

In 2003, Robins et al. [2003] published a thorough review of the research on introductory programming so far. The first paragraph is a glimpse into the main findings of their work.

> Learning to program is hard [...] Novice programmers suffer from a wide range of difficulties and deficits. Programming courses are generally regarded as difficult, and often have the highest dropout rates. [Robins et al., 2003, p. 137]

After almost 20 years, is the situation changed? To this day, is it still difficult for many to learn to program? First and foremost, the problem of access to informatics persists, which

[2] Please, refer to Lewis et al. [2019b] and Denner and Campe [2018] for a comprehensive discussion on the participation problem, the opportunities and strategies to deal with it.

[3] Digital competence is one of the eight broad citizenship competence areas listed in 2018/C 189/01 European council recommendation for lifelong learning.

means that introductory programming is yet too high a barrier. The shortage of informaticians in the labour market, with hundreds of thousands of positions open [see, e.g., Grover and Pea, 2013; Google LLC. & Gallup Inc., 2016] and set to rise in the near future given the ever-increasing relevance in all areas of data science and artificial intelligence. Moreover, introductory informatics courses (most often 'CS1') have always been considered difficult, and there have been constant reports of high levels of failure and dropout [see, e.g., Newman et al., 1970; Garcia, 1987; Allan and Kolesar, 1997; Sheard and Hagan, 1998; Guzdial and Soloway, 2002; Beaubouef and Mason, 2005; Kinnunen and Malmi, 2006; Howles, 2009; Guzdial, 2010; Corney et al., 2010; Mendes et al., 2012; Watson and Li, 2014]. Furthermore, students often report high levels of anxiety and frustration [see, e.g., Morgan et al., 2018]; continuing to fail is linked to frustration and reduced self-efficacy [see, e.g., Burleson and Picard, 2004; Eckerdal et al., 2006], which can result in demotivation and dropout [see, e.g., Bosch and D'Mello, 2013; D'Mello and Graesser, 2012]. That these negative indicators were high was mostly anecdotal knowledge until 2007, when a study attempted to quantify the problem involving 63 international institutions, reporting a failure level of about one-third [Bennedsen and Caspersen, 2007]. A subsequent study involving 51 institutions internationally, which was methodologically more rigorous, confirmed this result, registering a failure rate of 33% [Watson and Li, 2014]. Another study - while inviting caution due to the limitations of the sample - suggests that this negative figure (33%) is worse than the average 18% for all other courses ("*across all degree-level courses*" in New Zealand) and consequently also for introductory courses in the other sciences [Luxton-Reilly, 2016]. The authors of the influential 2007 research, given the persistent relevance of the topic, replicated the same investigation 12 years later and noted a slight improvement in the failure rate (28% in 2019 vs. 33% in 2007) but emphasized that the problems of introductory informatics still remain numerous and mostly open, requiring a continuous and increasingly focused research effort [Bennedsen and Caspersen, 2019]. We omit in this discussion the role that national and international institutions have (and should have) in promoting the introduction of informatics as a discipline in the lower levels of education, both because this is a broad and complex subject itself (with political and social implications), and because the situation all around the world is too highly diversified. We firmly believe that informatics is a science (see, Lodi and Martini [2021], Denning [2013], and [Tedre, 2018, sect. 2.4]) and therefore has full right to be one of the basic disciplines from the earliest levels of any school system (see, e.g., Lodi and Martini [2021], Tedre [2018], Guzdial [2015], and Code.org, CSTA, & ECEP Alliance [2022]). However, a reform of the curricula in this sense is far from complete [The Committee on European Computing Education (CECE), 2017; Code.org, CSTA, & ECEP Alliance, 2020]. The aim of such reform would not only be to train future professionals (i.e., increase participation) but also and above all, to enable citizenship for everyone, given the relevance of informatics in today's society and world. Above all, we are aware of how an increasing presence of informatics in all school orders will foster the quality (and quantity) of informatics (introductory and non) education research.

1.2 Research in the 20th century: programmers by birth

Initially (the early 1960s) for commercial and military development (i.e., finding and training programmers to serve companies and armies), and then in the following decades ('70s-'90s) also for scientific investigation (which constituted the first steps of informatics education research), attempts were made to determine and measure what characteristics of people made them more or less suitable for programming. The belief, widely accepted back then, that good programmers are born, not made, supported this kind of research [Dauw, 1967; Webster, 1996]. Aptitude tests have been developed for decades, one of the earliest and most significant of which is the IBM Programmer Aptitude Test (PAT), released in its first version in 1955. Despite the widespread use in the corporate sector of tools such as this, which have become increasingly sophisticated, the results provided have never proved particularly effective in predicting people's aptitude for programming, nor have they been able to mitigate the lack of programmers. The research in the new millennium, also starting from analysing what was produced in the preceding decades, has revealed such inconclusiveness. Looking back at one of his previous works [Robins, 2010], Robins summarizes: "*No reliable predictor of programming ability was found, however, and even large-scale analysis of multiple factors results in only limited predictive power*" [Robins, 2019, sect. 12.2.1]. The difficulty in identifying such predictive factors has further reinforced the belief that good programmers are born. This belief held back the development of research in informatics education while helping consolidate and spread the stereotypes that programmers are male and antisocial, artsy geniuses with rare and unfathomable qualities [Ensmenger, 2010].

1.3 New millennium research: two populations

Despite these stereotypes and beliefs, such as 'programmers are born' and 'programming is more difficult than other activities', research has become increasingly rigorous in the new millennium, showing a more complex and faceted reality. In particular, it emerged not only that programming was difficult for a significant portion of people but also, specularly, that around one-fifth of students find learning to program easy [Guzdial, 2007]. These "*two populations: those who can, and those who cannot*" [Dehnadi and Bornat, 2006], have been observed in numerous programming courses worldwide over a long period of time, and they emerged independently of geographical and social factors [Kölling, 2010]. This paradoxical situation, whereby higher than usual rates of failure and great success occur (and consequently a lower number of students with an average performance), has stimulated the interest of researchers who have started to refer to a 'bimodal' distribution of grades. Numerous studies of informatics education research ('IEdR' from now on) between 2008 and 2018 found polarized learning outcomes in CS1 courses [Bornat et al., 2008; Corney et al., 2010; Robins, 2010; Yadin, 2013; Elarde, 2016; Guzdial, 2010; Utting et al., 2013], a situation that - in retrospect - also emerged in a 1990 Cognitive Psychology paper [Hudak and Anderson, 1990]. Despite having contributed to the use of the term 'bimodal' [Robins, 2010] and having provided further evidence in support of bimodal distributions in introductory programming courses [Robins, 2018], Robins reflects on the use of this term. In particular, he

points out how it has overly influenced research, which has focused heavily on developing tests and statistical tools that could confirm a bimodal distribution. This trend has been at the expense of a more valuable investigation of the more general and, at the same time, situation-specific reasons for any polarized learning outcomes [Robins, 2019, sect. 12.2.2.3].

1.3.1 Is there such a thing as the programming gene?

So what is the reason why introductory programming courses have bimodal results all over the world and in most conditions (institutional, social, and so on)? Are there really people who are suited to programming and people who are not suited at all? In other words, is there such a thing as a "*geek gene*" for programming [Lister et al., 2010] whose existence would confirm the beliefs and stereotypes we have been mentioning? Everything seems to indicate not, even after a careful review (also informed by the awareness of bimodal results) of previous literature that had looked for cognitive, attitudinal or demographic factors showing a person's ability to learn programming. The most promising correlations among those investigated were those with mathematical abilities and IQ and with some relevant affective factors (e.g., motivation, positive attitudes to learning, high self-efficacy or effort). However, none was able to explain the polarized learning outcomes of introductory programming courses. In general, "*these factors are moderate predictors of success in many domains, and thus are not likely to account for any particular properties or pattern of outcomes in programming*" [Robins, 2019, sect. 12.3.7], establishing the substantial failure of 40 years of research to find the "*geek gene*".

1.4 The LEM hypothesis: learning to program is easy/difficult

Robins [2010]'s influential research does not merely point out that it is not people who are more or less suited to programming but proposes a change of perspective: it is programming that is a particularly difficult activity (more on this in the next 1.4.1). What is more, Robins proposes a hypothesis that seems to be able to explain not only the difficulties of learning to program but also the bimodal nature of the results in introductory programming courses. The LEM (Learning Edge Momentum) hypothesis holds that there is a moment in learning that, depending on the conditions, can either facilitate or hinder it.

> I suggest that LEM arises as a consequence of the interaction of two factors: the widely accepted principle that we learn at the edges of what we know; and the new claim that the concepts involved in a programming language are unusually tightly integrated. In short, successfully acquiring one concept makes learning other closely linked concepts easier, while failing makes further learning harder. This interaction between the way that people learn and the nature of the CS1 subject material creates an inherent structural bias which drives CS1 students towards extreme outcomes. [Robins, 2010, p. 40]

According to this hypothesis, learning to program would also be particularly challenging because, unlike most other introductory learning, the knowledge and skills required would

be numerous and interconnected. The fact that there is a lack of shared agreement among educators and in research as to the order in which the introductory programming – and even which concepts should be taught (see, e.g., Bruce [2004] and, more generally, ACM/IEEE-CS [2013]; more on this topic in 2.1) – concepts should be taught would be further evidence of the structural interdependence of the knowledge involved [Robins, 2019, sect. 12.3.7]. Therefore, the so-called 'LEM effect' would explain why the bimodal results are so widely observed. Students who fail to understand even very few of the first topics, given the interconnectedness mentioned above, would not be in a position to catch up by simply learning the subsequent ones [see also Meyer and Land, 2006]. In other words, these students experience a negative moment, bound to become more and more intense as the course progresses, and indeed get overwhelmed by the course. Conversely, students who succeed from the outset in learning the proposed topics would benefit from a positive moment favouring subsequent learnings, hence riding the wave of the course with relative ease. Either way, such a moment is self-reinforcing. Several studies have confirmed the predictions of the LEM hypothesis for the outcomes of introductory programming courses, particularly highlighting how crucial the first three weeks are [Porter and Zingaro, 2014; Porter et al., 2014; Hoda and Andreae, 2014; McCane et al., 2017]. The importance of the initial moments in learning programming will be one of the elements underlying our *Necessity Learning Design* (see 9).

1.4.1 LEM, Cognitive Load and Dual Process Theory

The concept of 'cognitive load' comes from cognitive science. Cognitive load theory describes and studies the effort that a task requires to the working memory [Paas et al., 2003; Plass et al., 2010; Sweller, 1988; 1994]. This theory defines three kinds of cognitive load: *intrinsic* (i.e., the effort related to the specific task or concept to be learned), *extraneous* (i.e., the effort to process the task or concept information and the way it is presented) and *germane* (i.e., the effort to learn the actual task or concept). The concept and theory of cognitive load will recur several times throughout this work. For now, it is worth noting that one of the factors that most influence intrinsic load is the interactivity of the elements involved, that is, how much the task requires to consider interactive elements that must be maintained in working memory all at once. Based on this, it becomes clear how programming imposes a high intrinsic load since it requires considering and orchestrating numerous interacting elements [Robins, 2019, sect. 12.3.5]. This is further evidence that programming is a complex activity. Robins [2022] explores *Dual Process Theory* (DPT), an influential theoretical framework in cognitive psychology[4], as a context for informatics education topics to define them more richly and precisely. "*DPT postulates the existence of two qualitatively different kinds of cognitive systems, a fast, intuitive 'System 1' and a slow, reflective 'System 2'. System 1 is associated with cognitive factors such as crystallized intelligence, long-term memory and associative learning; System 2 with fluid intelligence, working memory, and rule learning.*" [Robins, 2022] The continuous interaction between these two systems is embodied in the *Dual Process Cycle* (DPC), a learning model that shows S1 and S2 interactive and interdependent nature. While

[4]Dual Process Theory originates in the distinction between associative and true reasoning made by William James in 1890; it gained popularity recently for the book 'Thinking, Fast and Slow' by Kahneman [2011].

it is vastly accepted that "*knowledge builds on knowledge*" [Hunt, 2001], the self-reinforcing LEM effect might more broadly concern the entire Dual Process Cycle in the context of DPT: "*Knowledge builds on knowledge is true, but passive, a version of this observation which emphasizes its active implications, is that learning builds on learning*" [Robins, 2022, p. 15]. Whereas the original LEM theory is formulated in terms of passive knowledge S1, active factors, based on S2, also come into play in programming. Developing a program or building the mental model of a running program requires considering many highly interactive elements, resulting in a high cognitive load that can easily overload the working memory. An analogy with learning natural languages may further clarify: "*whereas learning the vocabulary of a foreign language is a low element interactivity task, learning the grammatical properties is likely to be a high element interactivity task because the elements interact and because learning them as individual elements may make no sense*" [Sweller et al., 1998, p. 260]. Shortly, the highly interconnected programming concepts require consistent success in S1 learning and also high demand for S2 processing. The framing of LEM theory in the context of DPT (also considering the high cognitive load of programming), on the one hand, more rigorously supports the presence of the LEM effect in learning to program. On the other hand, even better highlights how programming is a complex activity that requires considerable cognitive effort.

Chapter 2

Teaching To Make People Learn To Program

From what has been said in the previous section, teaching introductory programming emerges as a particularly tough challenge that we are still losing despite the growth of IEdR [Lunn et al., 2021] and, most notably, the growth of introductory programming research growing [Luxton-Reilly et al., 2018][1]. Even though more recent studies suggest that the situation is not as bad as previously thought by reporting less than alarming dropout rates [see, e.g., Watson and Li, 2014; Bennedsen and Caspersen, 2019], the concerning issue of access to informatics remains (as discussed in the previous section). Therefore, focusing more on the challenge of introductory programming could be crucial to improve the status quo since "*the nature and characteristics of a 'grand challenge' require articulation*" [Mcgettrick et al., 2005, sect. 1.2][2]. We will briefly discuss just some of the main dimensions of the challenge of teaching introductory programming to be able to navigate approaches and methodologies proposed by research more consciously. These methodologies (or pedagogical practices) are numerous (see Luxton-Reilly et al. [2018], Caspersen [2018] and Robins [2019] for the latest comprehensive reviews on introductory programming), despite the fact that informatics education is a more recent and less mature field of research compared to educational research of the other well-established disciplines. In fact, despite the many proposals and experiences out there, there is not a shared and clear consensus in IEdR as to which methodologies are generally effective for teaching programming. And still, introductory programming courses do not register good enough results [Watson and Li, 2014; Bennedsen and Caspersen, 2019].

2.1 Lack of agreement

One aspect of the challenge of teaching programming is the lack of agreement on the best ways to teach it to novices. There is also no agreement on what concepts should be taught

[1] Please, note that the ITiCSE working group that conducted the research emphasizes that this does not simply mean more relevant and high-quality publications.

[2] For the interested reader, Mcgettrick et al. [2005] indeed do so very thoughtfully in section 2.4 of their work on the grand challenges of informatics education.

in introductory courses, in what order they should be introduced and even when it is best to start teaching to program. The chapter on informatics introductory courses from the Joint Task Force on Computing Curricula (the most authoritative scientific source on informatics curricula expression of both ACM and IEEE) describes this situation at its very beginning.

> Computer science, unlike many technical disciplines, does not have a well-described list of topics that appear in virtually all introductory courses. In considering the changing landscape of introductory courses, we look at the evolution of such courses from CC2001 to CS2013 [...] we believe that advances in the field have led to an even more diverse set of approaches in introductory courses than the models set out in CC2001. Moreover, the approaches employed in introductory courses are in a greater state of flux. [ACM/IEEE-CS, 2013, ch. 5]

One of the main reasons for this uncertain scenario is that informatics is an ever-changing field, so rapidly evolving.

> Due to rapid growth and changes in our field, computer science standards cannot be static. These standards must be reviewed and updated on a regular basis, and not considered complete and finalized. CSTA[3] is committed to an inclusive, iterative process that allows multiple drafts and revisions of the CSTA K–12 CS Standards. [CSTA, 2016, p. 8]

However, it is not that there is a lack of scientific awareness on the subject of curricula. For example, Barendsen and Schulte [2018] devote an entire chapter of the 'Computer Science Education' book [Sentance et al., 2018] to provide background and tools to critically review curriculum documents and practices, discussing theories and examples, and even instructors' factors influencing curricula implementations. The other main reason for this lack of agreement is the strong interconnectedness we already mentioned in previous sections, especially in 1.4. Bruce [2004] tried to summarize a discussion that happened in 2004 between SIGCSE members on how to teach introductory programming courses using Java. Despite the specific programming technology, a general picture emerges that shows there is no right path in introductory courses, partly because of the tightly related and interdependent programming concepts.

2.2 Abundance of programming languages and tech

Another aspect of the challenge of teaching programming is the vast plethora of programming languages and technologies (i.e., the combination of a programming language with a developing environment) that educators can use to this day. Programming languages can be categorized by language paradigm, using categories such as, e.g., functional, logic, procedural and object-oriented. These categories are not necessarily disjointed since a particular language can fit more than one. Languages can also be categorized by their syntactic appearance, i.e.,

[3] CSTA is the international (although quite US-centric) 'Computer Science Teacher Association'; see www.csteachers.org.

text-based (lexical) or block-based (graphical). Such an abundance of choices may confuse less experienced educators. It often happens in Italian schools, for example, that teachers uncritically rely on the choices made by the textbook, which is likely to be the result of a choice over which they had no say. From the perspective of programming education research, there is a lot of production specific to a particular language or technology. This contributes to a rich but disorganized landscape of theories and methodologies within which it is difficult to find one's way around, especially for those educators who are not also researchers. In this work, as far as possible, we will avoid referring to specific languages and technologies so that what we present will hopefully be valid in all introductory programming contexts. However, this will not always be possible. We will occasionally need to refer to concrete contexts, particularly when describing the implementation of specific teaching sequences, discussing concrete examples presented to students, or describing the design of an experiment.

2.2.1 A two-speed evolution

Despite a significant evolution from the 1950s to the present of programming languages and technologies (which has produced, at least for the most widely used languages, various specific learning tools and methods), Robins et al. [2003] observe that programming teaching practices had not evolved as much. Witness to this is that most introductory programming textbooks focus mainly on the language of choice, organizing a learning path based on knowledge of its constructs and uses. Similarly, Kölling [2003] finds that the 39 programming books analyzed (among the best sellers) often leave the development process implicit, and all are structured around the language constructs. de Raadt et al. [2005], comparing 40 introductory programming books, register a tendency to provide much detail on constructs at the expense of the big picture. Lin and Wu [2007] review 32 Thai informatics books and finds that the programming content was inadequately treated.

Typically, the book section on a language construct (e.g., the conditional construct) presents a problem and illustrates a program that solves that problem, discussing the program elements. The program thus appears to have been developed in a single stroke, from problem to working solution. This approach illuminates to learners that programming is straightforward (thus contributing to their frustration when they fail). The development process (i.e., starting with an incomplete solution, proceeding by trial and error, extending, restructuring and refining to arrive at a solution) is omitted. While the solution (the final program) is explained in detail, the solution development process is almost completely neglected in introductory programming textbooks and courses [Caspersen and Kölling, 2009].

As far back as the '70, Gries [1974] notes that in no other field is expected that teaching the use of tools and then showing a finished creation made with those tools is sufficient to learn how to replicate that creation without any explanation of the realization process.

Learning to program requires much more than being instructed in detail on the syntax and semantics of language constructs. "*Students also need knowledge about the programming process, i.e. how to develop programs, and they need to extend that knowledge into programming skills.*"[4] [Caspersen, 2018, sect. 9.2]. In the early 2000s, Mcgettrick et al. [2005]

[4] In the next section, we will describe programming skills more formally to understand better one of the

point out that teaching and learning to program remains one of the great challenges of IEdR.

> [M]ajor concerns exist among the academic community internationally that when we set out to teach programming skills to students, we are less successful than we need to be and ought to be [...]. The particular concern is that, after more than forty [now fifty] years of teaching an essential aspect of our discipline to would-be professionals, we cannot do so reliably. Indeed, there are perceptions that the situation has become worse with time. [Mcgettrick et al., 2005, sect. 2.4]

And still today, Caspersen writes: "*[i]n a time where computing/informatics education is becoming general education for all and students don't choose to learn programming out of personal interest, the challenge not only persists, but is reinforced*" [Caspersen, 2018, sect. 9.2].

These influential voices, from Gries's early warnings in 1974 to Caspersen recognizing that the challenge of introductory programming is even greater today, clearly picture a two-speed evolution. The teaching practice of introductory programming is failing to evolve significantly despite the research effort, while programming languages and technologies are running and informatics are in short supply.

2.3 What really means learning to program?

To fully understand the challenge of teaching introductory programming, it is essential to define what learning to program means. As discussed in the previous section, knowing how to program cannot coincide with knowing, albeit completely, the constructs of a programming language. Programming is a tool for automating problem solving and, more generally, accomplishing tasks. Knowing how to program means having developed various types of knowledge and requires the ability to reason and operate simultaneously. According to Falkner and Sheard [2019]:

> The act of learning programming involves concurrently developing skill and knowledge in several related areas: planning, design, programming language structure, and also an understanding of how programs are to be executed. Students must learn both how to craft programs to achieve a fixed output as well as comprehending what an existing program will do. [Falkner and Sheard, 2019, sect. 15.4.4]

2.3.1 The three dimensions of learning to program

One of the most influential and widely used frameworks for analysing programming knowledge, proposed in 1997 and still widely used today, is the conceptual framework by McGill and Volet [1997]. It is based on the fact recognized by cognitive science that programming requires

fundamental aspects of the challenge of teaching programming, that is, what knowledge and skills we aim to teach.

different types of knowledge [Bayman and Mayer, 1988], which according to McGill and Volet's framework, are syntactic, conceptual and strategic knowledge. *Syntactic knowledge* refers to knowledge of the syntax of a programming language and its constructs, in short, the language's "*grammar rules*". *Conceptual knowledge* refers to the functioning of the programming language and its constructs. It concerns the dynamics of programs, i.e., knowledge of how constructs function and how this functioning determines code execution. *Strategic knowledge* refers to the ability to use syntactic and conceptual knowledge to solve new problems and, in general, to pursue desired goals. While textbooks and programming courses often focus primarily on and are organized around syntactic knowledge (see the previous section), being able to program requires certainly syntactic but also conceptual and strategic knowledge.

Interestingly, according to the KSC taxonomy – formally defined by the work of Winterton et al. [2006] and widely accepted in education[5] – which distinguishes knowledge, skills and competencies, we can venture a mapping of McGill and Volet's three knowledge onto KSC. That is, syntactic knowledge remains knowledge, but conceptual knowledge can be more accurately described as conceptual skills and strategic knowledge as strategic competencies.

Morevoer, in his Block Model – an educational model of program comprehension – Schulte [2008] also identifies *three dimensions in program comprehension*, which are rather self-explanatory: text surface, code execution (i.e., data flow and control flow) and program purpose. His model is a double-entry table consisting of these three dimensions and four levels (relating to the code structure, from a single instruction to the entire program). Schulte uses the concept of 'dimensions of understanding' as an alternative to 'types of knowledge'. However, the three dimensions of the Block Model are very much akin to McGill and Volet [1997] three dimensions of knowledge.

Several studies [see, e.g., McCracken et al., 2001; Whalley et al., 2006] show that syntactic knowledge/skills (dimension of code execution) and especially strategic knowledge/competences (dimension of program purpose) are the most difficult for novices to develop.

2.3.2 The SOLO taxonomy

The *Structure of the Observed Learning Outcome* (SOLO) taxonomy is a framework designed to evaluate and categorize the quality and complexity of learning outcomes. It provides a hierarchical structure that allows educators to assess learners' depth of understanding and skills. Biggs and Collis [1982] introduce the SOLO taxonomy as an alternative to traditional assessment methods. They categorize learning outcomes into five levels, ranging from prestructural to extended abstract, each representing a progression in understanding and complexity to assess the depth of understanding.

1. *Prestructural.* At this level, learners have minimal or no understanding of the concept or skill being assessed. They may provide incorrect or irrelevant responses and demonstrate

[5]The KSC taxonomy is the result of an extensive research initiative commissioned by the EU, and one of the reasons for its wide dissemination is that it became an integral part of the *European Qualifications Framework for Lifelong Learning* (2008 Recommendation of the European Parliament and Council).

a lack of grasp on the foundational knowledge required.

2. *Unistructural.* Learners exhibit a basic understanding of isolated elements or aspects of the concept or skill. They can identify or describe individual components or procedures but struggle to make connections or apply their knowledge in a meaningful way.

3. *Multistructural.* Learners demonstrate a more developed understanding by recognizing and incorporating multiple elements or aspects of the concept or skill. They can apply different techniques or strategies independently but still view them as separate entities, without fully integrating them.

4. *Relational.* At this level, learners can connect and integrate various elements or aspects of the concept or skill, demonstrating a deeper understanding. They can analyze the relationships between different programming concepts, recognize patterns, and apply their knowledge in more complex contexts.

5. *Extended Abstract.* Learners have a comprehensive and flexible understanding of the concept or skill, allowing them to generalize and apply their knowledge in novel or unfamiliar situations. They can think critically, create innovative solutions, and evaluate the broader implications of their programming choices.

This taxonomy is found to be effective in evaluating the quality of learning outcomes beyond surface-level performance. In general, the taxonomy provides a more nuanced assessment by focusing not only on the correctness or quantity of responses but also on the underlying understanding and reasoning behind them.

When applied to learning to program, the SOLO taxonomy becomes a valuable tool for assessing the development of programming skills and understanding among learners. It allows educators to evaluate the quality of learners' program design, algorithmic thinking, and problem-solving. It provides for classifying learners' programming solutions into different levels based on the depth of understanding and complexity demonstrated in their programs. This taxonomy enables educators to track learners' progression from basic understanding to more advanced levels of programming proficiency. Ginat and Menashe [2015] focus on assessing novices' algorithmic design skills using the SOLO taxonomy. They adapt the taxonomy to evaluate the complexity and quality of students' algorithmic solutions. The study demonstrates the usefulness of the taxonomy in capturing the different stages of algorithmic design skills among novices and provides insights for enhancing the assessment of programming assignments. Izu et al. [2016] investigate the program design skills of novice programmers using the SOLO taxonomy. They use the taxonomy to assess and categorize the complexity and quality of students' program design solutions. The study highlights the effectiveness of the SOLO taxonomy in evaluating program design skills and demonstrates the progression of novice programmers from lower to higher levels of code design proficiency.

The SOLO taxonomy also supports the design of assessments and tasks that target specific programming skills and facets. By identifying different dimensions of programming skills, such as problem decomposition, pattern recognition, or debugging, the SOLO taxonomy helps create a comprehensive assessment framework that captures the various aspects of

programming competence. Castro and Fisler [2017] extend the application of the SOLO taxonomy by designing a multi-faceted version to track program design skills throughout an entire course. They identify various facets of program design skills and developed assessment tasks targeting each facet. This multi-faceted SOLO taxonomy enables instructors to assess specific strengths and weaknesses in students' program design skills, informing targeted interventions and instructional adjustments.

Overall, these studies support the value of the SOLO taxonomy as an assessment framework in programming education. The taxonomy facilitates a comprehensive evaluation of learning outcomes, allowing educators to identify areas for improvement, tailor instruction, and promote deeper understanding and proficiency in algorithmic and code design skills.

2.3.3 (Faulty) Mental models in learning to program

The concept of a 'mental model' has long been used in IEdR and comes from cognitive sciences, where it is extensively used [Gentner, 2002; 1983; Johnson-Laird, 1983]. Mental models are internal models of the functioning of something, an iconic description of specific aspects of objects or systems. Mental models can be used to understand an observed behavior and make predictions about the evolution of that behavior.

For example, Smith and Webb [1995] argue that the difficulties novices experience in learning to program are often related to inadequate mental models of computer they are trying to program. Also Perkins and Simmons [1988] suggest that without a correct mental model of the computer, programming learning is fragile.

In his influential paper, Pea [1986] explores the concept of "conceptual bugs" in novice programming. These errors arise from faulty mental models that programmers develop when learning to program. Pea argues that these conceptual bugs can be language-independent, meaning they are not specific to any particular programming language or system. To emphasize the criticality of faulty mental models, Pea also notes that these errors are often difficult to detect and correct, as they can arise from deep-seated cognitive biases and misconceptions.

Pea identifies several sources of conceptual bugs. One common source is the tendency for programmers to rely on analogies to familiar systems or concepts when developing mental models of programming languages or systems. For example, a programmer might conceive a programming language as a natural language and rely on familiar grammatical structures to guide program design. However, this can lead to errors when the analogy breaks down, such as when the programmer encounters a programming construct that does not align with their mental model of the language. Another way mental models can contribute to conceptual bugs is by using mental shortcuts or heuristics that are not always appropriate for the task at hand. For example, a programmer might use a mental shortcut to assume that a particular piece of code will behave in a certain way without fully understanding the underlying logic or implications. This can lead to errors when the shortcut fails to account for essential nuances or edge cases. Finally, as a common source of conceptual bugs, Pea identifies the tendency to overgeneralize from a limited set of examples.

Overall, Pea [1986] highlights the importance of developing accurate and detailed mental models when learning to program. The connections between mental models and programming suggest that developing accurate and detailed mental models is critical for effective

programming. By understanding the potential sources of conceptual bugs and how mental models can contribute to planning and organizing code, learners (and educators, too) can develop more effective strategies for debugging and problem-solving in programming contexts.

The literature on misconceptions shows (for a comprehensive review of programming misconceptions, see Lewis et al. [2019a] and Sorva [2018]), that novices are not able to construct correct mental models of the execution simply by writing programs.

2.3.4 Notional machines as mental models of execution

As already cited, "*[s]tudents must learn both how to craft programs to achieve a fixed output as well as comprehending what an existing program will do*" [Falkner and Sheard, 2019, sect. 15.4.4]. To this end, Du Boulay [1986] came up with the concept of 'notional machine' that refers to "*the general properties of the machine that one is learning to control*", which are determined by the semantics of the programming language in use. In other words, a notional machine is an abstract, idealized representation that describes the behavior (i.e., the runtime execution) of a programming language (or part of it), independent of the characteristics of the physical machine. Shortly, a notional machine is human-friendly model of computation. The notional machine has the educational purpose of supporting the understanding of how a (piece of a) program is executed, so it is not necessarily consistent with the abstract machine of the programming language either (more on this, later on this section). Understanding the notional machine of a (part of a) programming language means possessing the conceptual knowledge [McGill and Volet, 1997] or getting the dimension code execution [Schulte, 2008].

As it is easy to understand, a notional machine can also be seen as a mental model that educators want learners to build. As the literature on misconceptions shows (see 2.3.3), teaching programming without making the notional machine explicit can lead learners to build incomplete or conflicting mental models, based on conjectures and affected by misconceptions. For instance, novices often see a program only as a series of consecutive instructions, failing to understand how those same instructions control the computer from which they are executed [Du Boulay, 1986]. Furthermore, novices often assume human reasoning in computers [Ragonis and Ben-Ari, 2005]. Sorva [2013], in an influential literature review on notional machines, collects the many misconceptions derived from incorrect models, even on topics that most programmers consider apparent.

The notional machine concept has substantially impacted IEdR, particularly in introductory programming research [see Sorva, 2013; Robins, 2019]. Various methodologies have been developed to use notional machines to abstract program execution. Their common goal is to make evident elements of execution that could favour comprehension (e.g., instruction sequencing, memory usage) and omit details that are inessential to a valid mental model of language semantics. Notional machines obviously differ by programming language. However, most of all, the educational context, learning objectives and learners' characteristics influence the level of detail of a notional machine [Sorva, 2013]. In particular, educators should "*acknowledge the notional machine as an explicit learning objective and address it in teaching*", and teaching "*may benefit from using multiple notional machines at different levels of abstraction*" [Sorva, 2013]. Educators indeed use a diverse repertoire of notional machines to draw students' attention to different aspects of programming (e.g., from "variables as boxes"

to "table with IDs" when introducing references) and sequences of notional machines, that expand when new concepts are introduced[6] [Fincher et al., 2020]. In fact, at different stages of learning, educators may show different notional machines whose features are chosen to facilitate learning rather than to be faithful to the abstract machine of the chosen language. However, the more advanced the learning path, the target notional machine may get closer to the abstract language machine. Positive accounts of the use of notional machines in teaching introductory programming are present in IEdR literature. For example, students who did tracing through a specific notional machine from the course's beginning performed better than those who only did code-writing activities [Nelson et al., 2017]. diSessa and Abelson [1986], presenting their *Boxer* system for novice programmers, which shows the computational objects of a program in boxes that change as the execution goes (e.g., to clarify the scoping mechanism), already emphasize that learning a notional machine is a necessary challenge.

In any case, to this day, IEdR literature lacks research evidence on which notional machines. Even worse, despite extensive attention in the literature, notional machines are not present in informatics curricula or textbooks. According to Krishnamurthi and Fisler [2019], few papers propose or test teaching proposals explicitly presenting notional machines alongside programming constructs. The many visualization and debugging tools do not explicitly explain the same execution models they implement [Hundhausen et al., 2002; Kölling et al., 2015; Naps et al., 2002; Sorva, 2012]. The situation is not much better among teachers. Surveying almost 500 CS1 teachers on various programming topics, Bennedsen and Caspersen [2006] reported that less than 30% of respondents explicitly addressed a notional machine in their courses. Even if concepts like the execution model and program dynamic were rated important by the teachers, the notional machine was not. However, research argues that explicitly teaching notional machines early in the programming learning path can be very helpful. From a cognitive perspective, in fact, building a new mental model (i.e., the notional machine of program execution) is much easier than updating a pre-existing (incorrect) one [Gupta et al., 2010; Schumacher and Czerwinski, 1992; Slotta and Chi, 2006]. Activities that require learners to engage with content are more effective in building sound mental models than passive activities, such as merely using visualization tools [Freeman et al., 2014; Kessel and Wickens, 1982; Savery and Duffy, 1995]. Therefore, the underestimation and underuse of notional machines emerge as a challenge in the great challenge of teaching programming.

2.3.5 Schemas in learning to program

Rist explores the topic of schema creation, application, and evaluation in the context of programming education. *Schemas* are mental structures that help individuals organize and represent knowledge; schemas aid problem-solving and program design in the context of programming. He emphasizes the significance of schemas in programming learning and problem-solving, which are mental structures that help organize and represent knowledge.

[6] This movement along different notional machines is akin to the movement across levels of *hierarchies of abstract machines* [Gabbrielli and Martini, 2010, ch. 1], where a system is designed in terms of *layers of abstraction* to help manage the system complexity. This view applies to programming languages, but also networks, operating systems, and so forth.

Rist [1989] investigates schema creation in programming and its impact on cognitive processes, Rist – who reviews the early literature on empirical studies of expert and novice programming strategies in his article [e.g., Guindon et al., 1987; Visser, 1987] – discusses how programmers form schemas by identifying patterns and regularities in programming tasks, emphasizing the role of experience, practice, and exposure to different tasks in schema development. The paper distinguishes between surface-level schemas, specific to programming languages or domains, and deep-level schemas, which capture broader programming concepts and strategies. Rist recognizes three pairs of antithetical program implementation strategies: top-down vs. bottom-up, forward vs. backward, and breadth-first vs. depth-first. In particular, we consider only the first pair. A top-down design strategy initiates with a high-level abstract problem and breaks it down into subproblems. These subproblems are further decomposed until reaching the lowest level, where program code is written to solve them. Conversely, bottom-up design starts with lower-level abstractions, creating pieces of the solution that serve as a foundation for solving higher-level problems. These pieces are then joined hierarchically to fulfill the overall program's purpose. Novice programmers often struggle when writing a program, lacking a clear overall plan or organization: they cannot consistently employ top-down strategies due to their limited possession of abstract plan schemas necessary for matching problems with solutions and decomposing them into connected pieces. This observation aligns with previous research findings indicating that novices possess predominantly programming knowledge representations at a low level, whereas experts have representations spanning abstract and concrete levels. He also discusses how novices in programming encounter language-independent "conceptual bugs" due to the absence or misalignment of schemas (see 2.3.3). By recognizing these issues, educators can better design instructional strategies to facilitate schema creation and improve programming skills.

Rist [1991] compares novice and intermediate student programmers' knowledge creation and retrieval processes. He investigates how these two groups create and retrieve knowledge during the program design process. The intermediate students typically adopt a top-down design approach, while the behavior of novice programmers is more complex and challenging to capture. The top-down design depends on the programmer's expertise and the complexity of the problem being solved. In this study, the top-down design emerges as based on refining a known solution or schema that is retrieved and expanded at increasing levels of detail. If the programmer is proficient in all required schemas, the design will exhibit a consistent top-down and forward expansion pattern at all levels. Conversely, when a schema cannot be retrieved and must be created, the top-down design breaks down and is replaced by a bottom-up design. In cases where the programmer is a complete novice, all required plans must be created, and the program design exhibits a consistent pattern of bottom-up and backward solution development. The paper highlights the importance of knowledge creation and retrieval processes in program design. It emphasizes the need for instructional interventions that facilitate the development of effective problem-solving strategies and the retrieval of relevant knowledge to enhance learners' program design capabilities.

Rist [2005] delves into learning to program and the role of schema creation, application, and evaluation. The author highlights the importance of schema creation in programming education, as it enables learners to develop effective problem-solving strategies and program-

ming techniques. The research explores various methods for teaching programming, such as worked examples and problem-solving exercises, which aid in schema application and consolidation. The paper also addresses the evaluation of programming learning and proposes different assessment approaches to measure students' schema acquisition.

Overall, Rist provides valuable insights into the role of schemas in programming education. He highlights the importance of considering cognitive factors, knowledge organization, and instructional design principles to support effective program design and problem-solving skills. By understanding the process of schema creation, educators can develop targeted interventions and instructional approaches to promote successful programming learning outcomes.

Rist's schema theory on top-down vs. bottom-up

Adelson and Soloway [1985] found that expert programmers employ a top-down strategy when working on familiar problem domains. They mentally simulate solutions at each lower level of abstraction. However, experts switch to a bottom-up strategy in unfamiliar domains, using simpler local models and combining them to form complete solutions.

Expanding on this work, Rist [Rist, 1989; 1991; 2005] developed a theory on how programmers use and create plan schemas during programming. When facing familiar problems with suitable plan schemas, top-down strategies are employed. Programmers revert to bottom-up strategies for unfamiliar or challenging problems to develop new solutions. Programmers alternate between these strategies based on the familiarity of the parts of the problem. Rist's theory suggests that experts possess more plan schemas, allowing them to rely more on top-down strategies. Novices, on the other hand, primarily use bottom-up strategies due to a lack of schemas. The longitudinal study of Rist [1989] revealed that novices transition from bottom-up to top-down strategies as they gain experience. The theory also explains findings indicating that experts analyze problems and plan, while novices focus on concrete code and local changes. Expertise growth is marked by the ability to use top-down strategies due to familiarity with problem types and solutions.

2.3.6 Abstraction in informatics and programming languages

Before proceeding, it is needed to clarify the concept of abstraction in informatics. Since it is a very complex and multi-faceted concept, we do not aim to cover all about it, just see its multiple dimension and put the focus on the one that's relevant to our work.

Informatics makes frequent reference to *abstract* entities and to activities identified as *abstractions*. The literature on informatics epistemology [Colburn and Shute, 2007; Turner, 2021] and on programming languages [Gabbrielli and Martini, 2010], identifies *information hiding* as the main abstraction objective – layers of abstraction are built in such a way that layer n uses the functionalities of layer $n-1$ to provide functionalities to the layer $n+1$ via an interface which also hides the information needed to implement them. This helps manage the system complexity and allows for the independence and replaceability of a layer without influencing the others. This view applies to programming languages (e.g., from a high-level language like Java to its bytecode implementation, to assembly, to the hardware machine), networks (e.g., the ISO/OSI stack), operating systems, and so forth. According

to this view, a given programming language can be seen as a (specific) abstraction of the underlying physical machine.

It is important to observe that (raw) expressivity has little to do with abstraction. Machine languages and high-level languages have the same expressivity, being all Turing-complete, yet we assign to them different abstraction levels, based on the (intentional) hiding of information that happens when we move from a machine language to a high-level one (e.g., specific representations of numbers as bit sequences are hidden, or "abstracted away," when passing from machine to high level).

We should note, though, that it is possible to identify different levels of abstraction *inside a single programming language*. Each programming language provides *abstraction mechanisms*, which are "*the principal instruments available to the designer and programmer for describing in an accurate, but also simple and suggestive, way the complexity of the problems to be solved*" [Gabbrielli and Martini, 2010, p. 165]. A `foreach` loop on a sequence is an abstraction from the `for` loop (of that same language), which would use explicit indexes on that sequence: the `foreach` hides those very indexes. In its turn, a `for` loop is an abstraction from a `while` loop because the `for` loop hides the explicit initialisation and increment of the index variable. In this paper, abstraction is therefore always connected to specific linguistic mechanisms.

Finally, observe that under this perspective:

- More expressive does not mean more abstract. We have already observed that expressivity and abstraction are in general orthogonal concepts. We remark here that even when expressivity is genuinely expanded, there are cases where there is no increase in abstraction. If we introduce first the *sequence* control structure, and then the *selection* control structure with the conditional command (`if-else`), we increase the expressive power of the language, but we do not have abstraction, because no information hiding occurs.

- More abstract does not always mean more general. In a C-like `for` construct we may specify almost arbitrary termination conditions and "increment" commands, and there is no constraint on the possibility of modifying the control variable in the body of the loop. Such a construct may be seen as the generalization of Pascal-like `for` and `while`. However, it is *not* more abstract of them, because no information hiding occurs. Conversely, a `foreach` construct is more abstract and less general of its `while` "translation".

We can distinguish between two general classes of abstraction mechanisms in programming languages: *control* and *data* abstraction.

Control abstraction mechanisms "*provides the programmer the ability to hide procedural data*" [Gabbrielli and Martini, 2010, p. 165]. Simple *control* abstraction comes into play early in a programming language learning path, for example, when dealing with structure control mechanisms like expressions, assignment, conditionals commands or iterative commands. Modern languages provide advanced mechanisms like procedures and exception handling constructs. For example, when dividing a program into subprograms (using functions or procedures), a programmer realizes a functional abstraction, separating what the clients of

such subprogram need to know to use it (e.g., name, parameters, return type) and what they do not need to know (the body, i.e., the implementation of the function, which could be changed – for example, for efficiency reasons – without the client knowing it) [Gabbrielli and Martini, 2010].

Data abstraction mechanisms "allow the definition and use of sophisticated data types without referring to how such types will be implemented" [Gabbrielli and Martini, 2010, p. 165]. Programming languages allow a variety of *data* abstraction mechanisms, hiding data representation details. These mechanisms range from simple ones, like the use of names to refer to memory locations, to the predefined language data types (collection of homogeneous, effectively presented values, with a set of operations on them), to more powerful mechanisms like the possibility to define *abstract data types*, up to all the data abstraction mechanisms provided by object-oriented programming.

From an educational perspective, the introduction of a new construct is often a movement across levels of abstractions. When, after a `foreach`, we unveil the possibility of an index-based iteration, we move down the abstractions. When we introduce functions, we provide a way to move up, and the same happens with the constructs for the definition of new data types. This movement across abstraction levels is a specific issue adding to the difficulty of learning. Students have to learn a new linguistic construct (or a new detail which was not previously introduced), its pragmatics when writing a program, *and* how this relates to the abstraction levels. Also choosing the correct construct (or the correct way to use a construct) is related to the abstraction levels. For example, students are instructed to prefer a `for` loop over a `while` for a sequential and complete scan of a sequence because the abstraction level of the `for` loop (the tool) matches the abstraction level of the "scan" (the problem).

Abstraction in program design strategies

Remaining within the domain of informatics education, we have just briefly reported on planning and program design strategies (2.3.5). These strategies can be analyzed, among other dimensions, by the direction of the movement of abstraction: downward (top-down strategies, usually employed by more experienced programmers) or upward (bottom-up strategies used by novice programmers). This is just another one of the countless facets of abstraction in informatics.

2.4 Active learning

The article 'Folk Pedagogy: Nobody Doesn't Like Active Learning' presents the findings of a survey of the informatics education community. Sanders et al. [2017] report that almost all respondents believe that 'active learning' favours student success in informatics. Is this 'folk pedagogy'[7], then, the answer to the issues with introductory programming seen so far? Is active learning really "*the almost silver bullet*" [Hicks et al., 2020] for CS1?

Let us take a step outside informatics education. First, we must define better what 'active learning' is in general, and then we will discuss the benefits of using active learning to teach

[7] J.S. Bruner defines the teacher's understanding of students' minds and learning as 'folk pedagogy'.

STEM (science, technology, engineering, mathematics) disciplines.

In an influential monograph on active learning, Bonwell and Eison [1991] note that, until then, educational literature had "*typically relied upon an intuitive understanding of the term*" [Bonwell and Eison, 1991, p. 18]. They define 'active learning' as anything that "*involves students in doing things and thinking about the things they are doing*" [Bonwell and Eison, 1991, p. 19]. The combination of doing and thinking is rooted in John Dewey's philosophy.

> [M]ethods which are permanently successful in formal education [...] give the pupils something to do, not something to learn; and the doing is of such a nature as to demand thinking, or the intentional noting of connections; learning naturally results. [Dewey, 1916, p. 154].

Moreover, according to Kolb [1984], the learning cycle includes action and reflection. Schön [1995] further develops the importance of action *and* reflection.

The benefits of active learning are well-established in the STEM educational communities. In an influential review of the literature on active learning, focusing on what is more relevant to engineering faculty, Prince [2004] finds that all forms of active learning are beneficial, albeit with varying effects depending on the specific active methodology. In general, greater student engagement is unquestionable, and the positive effects of an active approach go far beyond this dimension. He concludes that "*[t]eaching cannot be reduced to formulaic methods*"; however, at the same time, he warns that "*active learning is not the cure for all educational problems*".

Adopting a more quantitative approach, Freeman et al. [2014] examine 225 studies that, in undergraduate STEM courses, compare results (i.e., examination scores and failure rates) obtained with (various forms of) active learning with those obtained with traditional teaching. Two findings, in particular, stand out from the largest and most comprehensive (up to 2014) meta-analysis of active learning in undergraduate STEM education: an average 6 percentage points improvement in exam scores with active learning and a 1.5 higher probability of failure with traditional teaching. It also emerges that the improvement is greater in concept inventories than in exam scores, suggesting deeper understanding when using active learning. Finally, active learning appears effective regardless of class size (favouring, however, small classes). This work argues that active learning is an "*empirically validated teaching practice*" and suggests that "*calls to increase the number of students receiving STEM degrees could be answered, at least in part, by abandoning traditional lecturing in favor of active learning*". Such a recommendation could be an inspiration for responding to the problematic access to informatics discussed above (see 1.1).

Back to informatics education, McConnell [1996] is among the first to report on active learning strategies for teaching informatics. He discusses only a few specific active learning techniques (e.g., algorithm tracing and physical activities, such as having the class embody a protocol) and how, over two academic years, he integrates them into his informatics course in a US college. Despite the practical nature of his work (which has very few theoretical references the author does not delve into) and the very limited sample (i.e., only one course and one class, his own, over just two years), McConnell makes some very bold claims. In particular:

> The reality of today's higher education in the United States is that students do not seem to be as interested in learning as they once were. By employing active learning strategies, students not only learn content, but process as well. This makes them better students in later courses, and better professionals after finishing their degree. [McConnell, 1996, sect. 4]

Although not very in-depth and almost anecdotal, this article was published in multiple venues in 1996 and is widely cited, even today. Its bold statements probably encouraged the development of numerous active learning teaching strategies specific to informatics and the spread in the informatics education community of a folk pedagogy superficially supporting active learning.

As Sanders et al. [2017] show, this folk pedagogy is widespread today; unfortunately, it remains imprecise and not in-depth enough. In both the informatics education community and literature, "*[a]ctive learning doesn't refer to learning; instead, it is used as a shorthand for a collection of teaching techniques*" [Sanders et al., 2017, sect. 6].

Moreover, they report that the many techniques associated with active learning vary widely (e.g., in the type and level of activity required), and only some of them are specific to informatics. They also find that the respondents and literature rarely define active learning and rarely refer to any underlying educational theory. As in the educational literature prior to Bonwell and Eison [1991], informatics education mostly relies on an "*intuitive understanding*" of the term. Therefore, it does not a surprise that they identify another significant flaw. Few (among papers and respondents) talk explicitly about student reflection (one of the two pillars of active learning; [see Bonwell and Eison, 1991; Kolb, 1984]); even when they do, arguments are often built on weak foundations.

Speaking of weak foundations, they also write what follows. "*There is a striking lack of learning theories in the papers in the present study. Theory may seem abstract, difficult, or irrelevant.*" [Sanders et al., 2017, sect. 7.5] While *constructivism*, *situated learning* and *student-centered learning* receive some mention, little research provides a solid theoretical background. Therefore, the authors advocate more research based on the educational theories underlying active learning. Indeed, Suppes states: "*[a] powerful theory changes our perspective on what is important and what is superficial*" [Suppes, 1974, p. 4], and also: "*[o]ne of the thrusts of theory is to show that what appear on the surface to be simple matters of empirical investigation, on a deeper view, prove to be complex and subtle*" [Suppes, 1974, p. 5].

2.5 Major learning paradigms

Theories of learning describe how people learn. Philosophy, psychology and pedagogy have generated various theories of learning over the centuries. Theories in education are necessary beacons for researchers but also for practitioners. Knowing the fundamentals of theories of learning can help, for example, researchers to develop effective new methodologies, analyze existing methodologies and identify their flaws and developments, and educators to choose and correctly apply a methodology, modulating it for their particular context.

We have used the word 'theory'; let us introduce 'paradigm'. Although the use of these terms (i.e., theory and paradigm) is not univocal and can vary considerably across

scientific fields and authors, in the context of this work, we define 'learning paradigm' according to Leonard [2002]. Paradigms are "supersets" of theories; in particular, a paradigm categorizes learning theories into schools of thought based on their most dominant traits. Leonard classifies learning theories into four main paradigms: *behaviorism*, *cognitivism*, *constructivism* and *humanism*. Others (e.g., Baker et al. [2019]) also add *transformative* among the paradigms. Another significant distinction often found is between *instructivism* and *constructivism*.

It does not serve this work to detail and analyze all the paradigms, theories and distinctions (see Leonard [2002] for a compendium of learning theories organized by paradigms). However, a selective exploration of learning paradigms is necessary to illustrate the theoretical underpinnings of our vision for tackling introductory programming (and introductory informatics in general) and can help the reader understand the motivations behind our research. In the next subsections, we briefly explore *behaviorism*, *cognitivism* and *constructivism*.

2.5.1 Behaviorism

Behaviorism derives from behavioral psychology. This psychology is based exclusively on individuals' observable behaviors, ignoring internal mental states altogether, considered unknowable as they are derived from introspection, hence subjective and impossible to verify. Its pillar is *classical conditioning*, which is based particularly on the concepts of *stimulus-response* (from Ivan Pavlov's research with dogs) and *operant conditioning* (from B. F. Skinner's studies on positive and negative reinforcement). Behavioral psychology was predominant in the first half of the 20th century (before scientific attention was paid to the processes of the mind and phenomena of the psyche) and greatly influenced the way we think about learning.

Behaviorism as a learning paradigm went further the related psychology: it was dominant in education for much of the last century. Its influence remains even in today's education systems, despite being considered outdated philosophically and effectiveness-wise.

According to this paradigm, learning is guided and shaped through the "*process of conditioning by [...] sequences of stimuli, responses, feedback, and reinforcement*" [Falkner and Sheard, 2019, sect. 15.2], aiming to change the frequency, form, or both, of observable learner behaviors. Desired behaviors become habits or dispositions when they are systematically reinforced over time. Attention, memory and, in general, internal mental states and processes are not taken into account. Learners are viewed as blank slates whose behavior must be shaped. Baker et al. [2019] identify the key principles of behaviorism as a learning paradigm.

- Purpose of education is to shape desirable behaviour
- Learning is change in form or frequency of observable behaviour
- Emphasis is on producing observable and measurable outcomes
- Learners are blank slates who passively receive information
- Instruction is repetitive; teachers shape behaviour through reinforcement

Philosophically speaking, behaviorism assumes an objective view of knowledge. There is one ontological reality whose structure can be (studied,) taught and learned.

Behaviorism is the paradigm underlying *instructivist teaching*, which is still widely used today. Instructivist teaching focuses on the transmission of knowledge, thus paying attention especially "*on the structure and presentation of the learning material rather than the learners who act as recipients of the instruction*" [Falkner and Sheard, 2019, sect. 15.2].

Programmed instruction and *mastery learning* are two of the most influential learning theories of behaviorism.

- In programmed instruction, developed by Skinner, knowledge is divided into small units; in this way, it is possible to give learners frequent and individualized reinforcements.
- Mastery learning expands on the previous one. The overall learning objectives are allotted into small learning units of increasing difficulty. Learners proceed at their own pace and have to develop mastery of the fundamental objectives in each unit.

As said, behaviorism still significantly influences today's school system and educators. For example, Martinez et al. [2001] analyzed the reflections of 50 experienced teachers and found that "*the majority of these teachers shares traditional metaphors depicting teaching and learning as transmission of knowledge*". Also, *gamification* (in learning but not only), although it may seem modern and cool, is based on the behaviorist idea of reinforcement [Rice, 2012].

Why is it outdated?

Although its influence is still significant, behaviorism is considered outdated for two main reasons.

First, it does not take into account the internal mental states and processes of learners. Cognitive psychology has made tremendous strides since the middle of the last century, and today not considering the cognitive processes involved in learning is inconceivable.

Second, as mentioned above, behaviorism regards learners as blank slates to be filled. The learner's personal characteristics are not considered, and this lack has two dimensions. From a cognitive point of view, we now know that learning is strongly influenced by previous knowledge and experiences, mental models and strategies of each individual. Then, the noncognitive factors (i.e., motivations, attitudes, and dispositions) are widely considered decisive, along with the emotional-affective aspects.

2.5.2 Cognitivism

In the 1950s, *cognitivism* arose as a major learning paradigm, mainly to overcome behaviorism, which could not adequately explain all types of learning (e.g., problem solving). The purpose was to explore the mind (in contrast to behaviorism, which considers it a "black box") and investigate what happens between stimulus and response.

Cognitivism focuses on what learners know and how they acquire knowledge. It observes external behaviors (like behaviorism) but does so to investigate the learner's internal mental states and processes. Internal information processing in humans is compared to computer

information processing, regarding "*the brain as a processor of information and thought as a form of computation*" [Robins et al., 2019, sect. 9.5.3].

According to cognitivism, learning is about remembering and applying information. More specifically, the desired outcome is the perception, processing, storage and retrieval (memory), and application (transfer) of information. Learners are active participants, and educators focus on how learners structure, organize, and sequence information to facilitate their learning. Indeed, learning is not merely the result of external stimuli but arises from cognitive processes. Learning is achieved "*through a variety of learning strategies that depend on the type of learning outcomes desired, reflecting associated cognitive processes [. . . , including]* memorization, drill and practice, deduction, and induction" [Falkner and Sheard, 2019, sect. 15.2].

Baker et al. [2019] identify the key principles of cognitivism as a learning paradigm.

- Purpose of education is for learners to remember and apply information
- Learning is a change in symbolic mental constructions (or schema)
- Emphasis is on structuring, organizing, and sequencing information in the mind
- Learners are information processors
- Teachers facilitate optimal processing

From the philosophical perspective, cognitivism spans between an objectivist epistemology and a more constructivist one. While considering the internal cognitive processes involved in learning, an *objectivist epistemology* maintains the instructive idea of knowledge transmitted from the outside. Instead, according to a *constructivist epistemology*, everyone constructs their own knowledge building on their previous knowledge, experiences and social interactions (more on constructivism in 2.5.3).

Within the cognitivist learning paradigm, many theories and concepts were developed that are still relevant and used today, for example, the multi-store memory model, cognitive load (see 1.4.1), mental schemas and models (e.g., notional machines in informatics education; see 2.3.4), prior conceptions and misconceptions. Generally, cognitivist learning theories focus on (re)organizing knowledge to facilitate its learning, for example, by chunking, reducing cognitive load, and using learning taxonomies (e.g., Bloom's taxonomy).

2.5.3 Constructivism

Constructivism is a learning paradigm that emerges in the second half of the last century to challenge the instructive idea of knowledge transmitted by educators and received by learners; such an idea is present in behaviorism and some kinds of cognitivism. Highly influential in the origin of constructivism was the swiss psychologist and pedagogist Jean Piaget.

> [. . . T]he use of active methods [. . .] give broad scope to the spontaneous research of the child or adolescent and require that every new truth to be learned be rediscovered or at least reconstructed by the student, and not simply imparted to him. [Piaget, 1973, p. 15]

One of the philosophical foundations of constructivism is *relativism*. Objective reality does not exist, nor is it knowable; therefore, there is no "true" knowledge to be transmitted or taught. Instead, everyone constructs their own personal knowledge both individually and within groups and contexts. Indeed, constructivism is learner-centred and advocates active methods of learning.

Nowadays, this paradigm is very popular in education, manifesting itself in many forms and degrees of radicalism. However, Sorva [2012, p. 77] identifies two main ideas that emerge from most constructivist learning approaches.

- The learning of something new builds on the learner's existing knowledge and interest that learners bring into the context.
- Learning is the construction of new understandings through the interaction of the existing knowledge and new experience.

Of the various forms of constructivism, we can recognize two more specific learning paradigms within the broader one: *cognitive constructivism* and *social constructivism*.

2.5.3.1 Cognitive constructivism

Cognitive constructivism emerges in 1950s. The main contributors to cognitive constructivism were philosopher and pedagogist John Dewey, psychologist and pedagogist Jean Piaget, and psychologist Jerome Bruner. Dewey was an early proponent of hands-on learning and experimental education; Piaget theorized the stages of child development; Bruner laid the foundation for discovery learning.

Since knowledge is actively constructed, learning is a process of active discovery. Educators foster this discovery by guiding learners in the process and providing the necessary resources. In order to effectively structure content and materials, educators must consider learners' prior knowledge, cognitive development stage, cultural background, and personal history.

Baker et al. [2019] identify the key principles of cognitive constructivism as a learning paradigm.

- Purpose of education is to enable learners to create new knowledge
- Learning is the process of constructing meaning
- Emphasis is on active discovery
- Learners actively construct new knowledge, building on what they already know and past experiences
- Teachers facilitate discovery by providing necessary resources

2.5.3.2 Social constructivism

Lev Vygotsky (widely known for the concept of *zones of proximal development*, ZPD[8]) laid the foundation for *social constructivism* by theorizing that knowledge is constructed through social interaction.

[8]A ZPD is a development level that a learner can potentially reach with guidance and encouragement from a skilled partner (i.e., an educator or a more competent peer).

Social constructivism argues that knowledge construction cannot be isolated from the social environment where such knowledge is created. Learning is situated since it happens continuously through collaboration between learners and their social context. Whereas cognitive constructivism focuses on mental representations, social constructivism is more interested in how knowledge is built through social interaction. It focuses on social activities (learning through participation and relationships) and communities.

Baker et al. [2019] identify the key principles of social constructivism as a learning paradigm.

- Purpose of education is for learners to co-create knowledge
- Learning is co-constructing knowledge and norms through social interaction
- The emphasis is on human relationships, learning through participation (activity) in social contexts (communities)
- Learners are active participants
- Teachers facilitate social interactions and collaborative work

2.5.3.3 Cognitive vs. Social constructivism

The debate between cognitive and social constructivism has revealed two knowledge acquisition approaches. While cognitive constructivism emphasizes an individual's active engagement in the construction of knowledge, social constructivism focuses on the collaborative development of knowledge through learners' interactions.

Both approaches have advantages and disadvantages, with cognitive constructivism providing an essential foundation for individual inquiry and social constructivism offering a valuable opportunity for learners to discuss, collaborate and share ideas. Ultimately, both approaches can benefit modern classrooms and create a more diverse, engaging, and meaningful learning environment.

2.5.3.4 Constructivism in practice

Whether it is more cognitivist-inspired or social-inspired, constructivism is a highly influential paradigm in education nowadays (although it is not immune to criticism, as we will see in the following 2.5.3.5).

Prince and Felder [2006, p. 125] summarize the principles for effective constructive education, noting that proponents of constructivism use variations of these principles.

- Instruction should begin with content and experiences likely to be familiar to the students, so they can make connections to their existing knowledge structures. New material should be presented in the context of its intended real-world applications and its relationship to other areas of knowledge, rather than being taught abstractly and out of context.
- Material should not be presented in a manner that requires students to alter their cognitive models abruptly and drastically. In Vygotsky's terminology,

the students should not be forced outside their "zone of proximal development," the region between what they are capable of doing independently and what they have the potential to do under adult guidance or in collaboration with more capable peers Vygotsky [1978]. They should also be directed to continually revisit critical concepts, improving their cognitive models with each visit. As Bruner [1961] puts it, instruction should be "spirally organized."

- Instruction should require students to fill in gaps and extrapolate material presented by the instructor. The goal should be to wean the students away from dependence on instructors as primary sources of required information, helping them to become self-learners.
- Instruction should involve students working together in small groups. This attribute—which is considered desirable in all forms of constructivism and essential in social constructivism—supports the use of collaborative and cooperative learning.

The traditional teaching method through lectures does not align with the principles of constructivism. Constructivism, which has strong research support, emphasizes that effective instruction should involve setting up experiences for students to construct their own knowledge, adjusting or discarding their previous ideas as needed based on the evidence gained from these experiences [Prince and Felder, 2006]. In his doctoral thesis, Lodi argues that "*[m]ost of the contemporary learning theories and approaches, making their way into schooling and certainly driving much of the educational research [. . .], can be included in the constructivist paradigm*" [Lodi, 2020b, p. 41]. He reports some of the most relevant today.

- Active learning
- Cognitive apprenticeship
- Productive Failure
- Problem-based learning
- Community of practice
- Collaborative learning
- Cooperative learning

We will discuss some of them (e.g., Productive Failure) later, as they have been crucial for developing our research.

For a general review of constructivist theories and approaches from the informatics education perspective, see Fincher and Robins [2019, chapters 8, 9, 15] and Sorva [2012].

2.5.3.5 Critics to constructivism and the education debate

One of the criticisms raised against constructivism concerns its *knowledge relativism* (i.e., no objective reality hence no "true" knowledge). Philosophy of education debates are beyond the scope of this thesis and are not relevant to our research, even more so since we assume and refer to so-called *pedagogical constructivism*. Matthews [1997, p. 8] describes *pedagogical constructivism* as concentrating "*solely on pedagogy, and improved classroom practices, and [. . . calling] 'constructivist' [. . .] anything which is pupil-centered, engaging, questioning, and progressive*". He goes on and state that, to pedagogical constructivism, "*the details of epistemology and cognitive psychology are unimportant, and not worth disputing about*".

The most significant criticism of constructivism, however, claims that it is not efficient and effective for conveying information and skills and is too time-consuming compared to more instructivist approaches, such as direct instruction. This criticism is mainly directed at the purest and most extreme forms of pedagogical constructivism, in which student freedom is highest, and teacher intervention is reduced to the bare minimum, often gathered under the definition of *minimally-guided constructivism*.

There is an ongoing debate in the education community about which approach is more effective between (minimally-guided) constructivism and direct instruction and, more generally, instructivist approaches. A fundamental difference in the definition of learning is at the heart of the debate. On the one hand, those favoring instructivism believe that learning is a change in the brain resulting from the storage of new information and thus argue that direct instruction is the most efficient and easy method for learning. On the other hand, constructivists view learning as a change in knowledge that is only valuable with corresponding changes in learners' skills, such as specific problem-solving, and soft skills, such as collaboration. These changes are more challenging to study in controlled experiments than specific learning results, leading to another criticism of constructivism. The research methods employed in constructivism, for example, design-based research, are blamed for poor scientific rigor. According to Margulieux et al. [2019, sect. 8.2.1], the most accepted answer – which we agree with – is that "*scientific rigor is not worth research [. . .] conducted in sterile environments (i.e., labs) that are fundamentally different from the authentic environments (e.g., classrooms) in which the research will be applied*".

Methodological discussions aside (because they are out of the scope of this thesis), supporters of more instructivist approaches argue that they are more efficient and effective in developing the necessary knowledge and skills and are less time-consuming. The most significant criticisms came from Kirschner, Sweller, and Clark [2006] in the article "*Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching*". Based on cognitive science research, they argue that direct instruction is more effective than minimal guidance approaches. They bring evidence that expertise is gained through exposure to many examples, that beginners benefit from much guidance, and that working on complex and authentic examples is too mentally taxing for novices who are not yet accustomed to processing much new information at once.

Several studies have offered counterarguments to these criticisms, leading to a productive discussion found in the book by Tobias and Duffy [2009]. In general, we concur with Taber

[2012] asserting that it is overly simplistic to categorize all constructivist methods as "minimally guided". Indeed, some constructivists embrace entirely the philosophy of *discovery learning* (see 5.1.1 in part II), which emphasizes minimal guidance and encourages students to explore and learn independently. However, Taber [2012, p. 39] proposes a "more moderate" constructivism, in which the guidance adapts to the situation.

> [T]he level of teacher guidance (a) is determined for particular learning activities by considering the learners and the material to be learn; (b) shifts across sequences of teaching and learning episodes, and includes potential for highly structured guidance, as well as more exploratory activities.

According to Taber, this approach aligns with the concept of student-centered instruction but is not the same as minimal guidance. He advocates for varying the level of support provided, a concept known as *scaffolding*. This idea comes from Vygotsky and is preeminent in his production [see, e.g., Vygotsky, 1978], but it has been formalized by Wood, Bruner, and Ross [1976]. They defined *scaffolding* as a process that enables a child or novice to accomplish a task or reach a goal that learners would not be able to achieve independently. This process is made possible by controlling the elements of the task that are initially beyond the learners' ability, allowing them to focus on and complete only the elements within their competence. Taber acknowledges that if learning is viewed as a cyclical interaction between personal perspectives and experience, the complete lack of external guidance can only exacerbate the formation of unique worldviews. He believes society and education should aim to mitigate this by promoting the development of perspectives that align with general agreement. Therefore, Taber [2012, p. 57] recommends a form of constructivism centered on the student but directed by the teacher, which he refers to as "*optimally guided instruction*".

> The aim of constructivist teaching then is not to provide 'direct' instruction, or 'minimal' instruction, but *optimum* levels of instruction. Constructivist pedagogy therefore involves shifts between periods of teacher presentation and exposition, and periods when students engage with a range of individual and particularly group-work, some of which may seem quite open-ended. However, even during these periods, the teacher's role in monitoring and supporting is fundamental.

Guzdial, in the context of informatics, also suggests a balance between hands-on projects and direct instruction. However, this does not mean returning to traditional lecture-based approaches but instead incorporating active learning techniques.

> Students in computing should work on projects. It's authentic, it's motivating, and there are likely a wide range of benefits. But if you want to gain specific skills, e.g., you want to achieve learning objectives, teach those directly. Don't just assign a big project and hope that they learn the right things there. If you want to see specific improvement in specific areas, teach those. So sure, assign projects — but in balance. Meet the students' needs AND give them opportunities to practice project skills.
>
> And when you teach explicitly: Always, ALWAYS, ALWAYS use active learning techniques like peer instruction. It's simply unethical to lecture without active learning. [Guzdial, 2019b]

2.5.3.6 Optimal guidance in constructivism

It is worth noting that this debate is not necessarily a zero-sum one, with many researchers and educators recognizing the value of both approaches. Mixed approaches combine both constructivism and direct instruction elements and can lead to more effective learning outcomes by leveraging the strengths of both. Students may engage in self-directed, discovery-based learning activities and problem-solving tasks while also receiving direct instruction and guidance from the teacher as a form of scaffolding. This can allow students to actively construct their own understanding of the material while also ensuring they have a solid understanding of key concepts and skills. The type of instruction used should be based on the learner's needs and what is most appropriate. For instance, when introducing novice programmers to Java, a more direct approach would be more efficient, but this type of instruction may lead to shallow learning and loss of motivation. On the other hand, when the learners are already experienced with Python, a constructivist activity that scaffolds the connection between new Java knowledge and prior Python knowledge will likely support their motivation (since it should be active, reflective, and based on previous knowledge) and help them learn Java more deeply without significantly affecting efficiency. Ultimately, a critical question around constructivism is still which is the optimal amount of guidance to support learning, and that should vary depending on the context.

To wrap up, we saw how lecture-based instructivist approaches are still prevalent in informatics and, more specifically, programming education. At the same time, the problem with introductory programming and the related difficult access to informatics testify that direct instruction alone is insufficient. We agree with Taber [2012], Guzdial [2019b], and in general, with the idea of adopting a constructivist approach but balancing it to offer variable scaffolding and, thus, constantly adapting it to a particular context and the specific moment in the learning path. We are aware that for introductory programming, where essential critical knowledge and technical skills are involved at the same time, the level of scaffolding may also be high (thus far from minimally-guided approaches). However, we maintain that the focus must remain on the learners and take into account, first and foremost, the construction of knowledge rather than cognitive processes (while important and to be considered) to foster meaningful learning and not undermine motivation. On the other hand, we believe it is essential to adopt the *zone of proximal development* perspective as a reference for finding optimal guidance to provide the proper scaffolding in the context of active activities... "*Always, ALWAYS, ALWAYS use active learning techniques*"! [Guzdial, 2019b]

2.6 Influential active methodologies with scaffolding

Optimal guidance is an educational strategy that seeks a balance between direct instruction and more active, constructivist activities. Direct instruction provides students with the foundational knowledge they need to understand new concepts and skills. On the other hand, constructivist activities allow students to apply and deepen their understanding through hands-on, experiential learning (see 2.5.3.6 and, more generally, 2.5.3).

By guiding the development and use of an active methodology, optimal guidance helps

ensure that instructional activities are appropriately tailored to the learner's needs and aligned with the desired outcomes. Thus educational design may involve incorporating a mix of direct instruction and constructivist activities, resulting in a continuous scaffolding that can enable students' *zone of proximal development*. Ultimately, optimal guidance aims to create a learning environment that supports student engagement and motivation while also helping students build the knowledge and skills they need to succeed.

After a more general introduction that frames optimally guided approaches in the context of introductory programming education, some approaches and methodologies that balance active, constructivist activities with various forms of scaffolding follow. We selected them because we believe they can help students understand programming concepts and build related strategic knowledge (see 2.3.1).

What definition of active learning

Before proceeding, it is appropriate to specify the definition of *active learning* this thesis subscribes to. The following definition synthesizes our interpretation of active learning from the various definitions and frameworks reported in 2.4. Active learning is an approach to instruction that engages students in the learning process through activities and discussion in class, as opposed to passively listening to lectures. Active learning approaches can include but are not limited to, problem-based learning, case-based learning, inquiry-based learning, collaborative learning, and peer instruction. Additionally, active learning involves a reflective phase where students are given the opportunity to reflect on their learning and experiences and to make connections between their prior knowledge and new information or skills acquired. This definition emphasizes that active learning involves both an initial active phase, where students are engaged in activities and discussions, as well as a subsequent reflective phase, where students are encouraged to reflect on what they have learned and how it relates to their prior knowledge and experiences. This reflective phase is essential to the active learning process, as it promotes deeper learning and metacognition.

2.6.1 The introductory programming context

Today, most education research agrees that active methodologies – whereby learners actively explore and construct knowledge – are helpful for learning [Prince, 2004; Freeman et al., 2014]. On the other hand, educators often have to teach specific introductory or technical concepts that students are unlikely to learn or even discover through free exploration informatics also faces this issue since it is a discipline with many technical aspects, especially in introductory programming [Guzdial, 2017]. As a result, in introductory programming courses, a common approach remains directly teaching language elements, usually followed by their application in programming assignments. Direct instruction of technical concepts does not seem ideal for novice learners: it may bore them, also because they may not grasp the significance of the presented concepts from their perspective [Caspersen, 2018]. This can result in low motivation and poor learning outcomes.

As recognized by Robins [2019, p. 356], "*the teaching of knowledge structures must be anchored in, and learning may most effectively emerge from, practical experience and*

examples". Also, "*[t]his highlights again the need for well-designed example tasks and the practical opportunities for students to engage with them*". Furthermore, Robins recognizes two essential features of programming tasks.

- Compiler feedback is "*immediate, consistent, and (ideally) informative*". We add that this can allow for impartial and accurate feedback from the artifact one is creating.
- The "*reinforcement and motivation derived from creating a working program can be very powerful*".

In 1.4.1, we presented the *cognitive load theory*, a general theory describing how much a task loads a student's working memory. Often problem-solving tasks can put a high load on students. Extensive research in informatics education investigates the relationship between learning to program and cognitive load. Robins [2019] summarizes four principles relevant to our work.

> The *worked-out-example effect* suggests that extraneous load is reduced by studying worked examples of problems rather than trying to solve the problems from scratch, and similarly the *completion effect* suggests that load is reduced when the learner starts with partial solutions. Other examples include the *guidance-fading effect*, stating that novices need extensive support that can be reduced over time, and the *isolated/interacting elements effect*, stating that tasks with high element interactivity will be learned more successfully if elements are first introduced in isolation before being combined. [Robins, 2019, p. 344]

Moreover, according to Caspersen [2018, p. 117], "*a good example must effectively communicate the concept(s) to be taught. There should be no doubt about what exactly is exemplified. [...] Conceptual knowledge is improved by best examples [...], where the best example represents an average, central, or prototypical form of a concept. To minimize cognitive load, an example should exemplify only one new concept (or very few) at a time.*"

In 1.4, we also discussed the *Learning Edge Momentum* hypothesis. We briefly recall that it is an explanation (to this day one of the most widely accepted) for the fact that, while many students fail introductory programming courses, many students perform exceptionally well. The hypothesis is that given a specific target concept to be learned, the acquisition of the earlier concepts facilitates (i.e., momentum toward success) the acquisition of the later concepts. Similarly, failure to acquire the earlier concepts makes learning difficult (i.e., momentum toward failure).

Robins [2010] argues that programming is a domain of highly integrated topics with clear and well-defined edges (like puzzle pieces). That is why the LEM effect is exceptionally strong in programming, often resulting in bimodal results. Again according to Robins, teachers and educators must try to establish positive momentum from the very beginning of introductory programming courses, and the introduction of a concept is a particularly critical stage for this goal. "*[P]articular attention should be paid to the careful introduction of concepts and the systematic development of the connections between them*" because "*there is no point in expecting a student to acquire a new layer of complex concepts if the foundation of prerequisite concepts does not exist*" [Robins, 2019, p. 360].

2.6.2 Problem-based learning

According to Prince and Felder [2006, p. 128], "*[p]roblem-based learning (PBL) begins when students are confronted with an open-ended, ill-structured, authentic (real-world) problem and work in teams to identify learning needs and develop a viable solution, with instructors acting as facilitators rather than primary sources of information*".

Problem-based learning was first introduced in the 1960s at McMaster University. Traditionally, medical students had to memorize much information that they perceived to be superfluous to medical practice, but they were very involved when they actually worked with patients [Barrows, 1996]. Therefore, they successfully experimented with a new methodology in which students are presented with problems (clinical cases) and have to study autonomously the material needed to understand and (possibly, but not necessarily) explain them. This methodology has been used in many other fields like Law, Social sciences, and Engineering[9].

According to a recent review [Bawamohiddin and Razali, 2017] on PBL for teaching programming, the problem is used to initiate and trigger the learning process. Problems must be ill-structured, real-world situated, complex, open-ended, motivational, unique, and solvable. Other researchers recognize similar characteristics [see, e.g., Oliveira et al., 2013; Peng, 2010].

Drawing from medical education, PBL has been often implemented with the so-called "seven-step method" (adopted by many informatics educators as well). The steps, as synthesized by Bawamohiddin and Razali [2017, p. 2036], are:

1) terms and concept clarification;
2) problem identification;
3) brainstorming;
4) explanatory model sketching;
5) learning issue formulation;
6) self-learning and
7) information synthesising and testing.

Steps from 1 to 5 and 7 are conducted in small groups, while step 6 is individual study time. Usually, the process is quite long: for example, steps 1-5 can take from half an hour to several hours, step 6 (individual study) can take a week or more, and finally the last step can take other 2 to 10 hours (e.g., according to the proposals of PBL for programming, by Nuutila et al. [2008] and Bawamohiddin and Razali [2017]). Teachers act as facilitators: they guide students by motivating them, helping them understand the problem and guiding them in establishing the relevant learning objectives. On the other hand, they usually do not directly teach the target concepts.

Implementing problem-based learning can be challenging for instructors as it requires subject expertise and flexibility. Instructors may have to step out of their comfort zone as student teams may take unpredictable and unfamiliar directions. PBL also puts more

[9] For a recent comprehensive handbook, see Moallem et al. [2019].

responsibility on students for their own learning. Many students may initially have a negative reaction to PBL, which can be intimidating for unprepared instructors. Therefore, instructors, should also use scaffolding, providing the *optimal amount of guidance* (see 2.5.3.6) to students new to PBL and gradually reducing it as they become more familiar with the method and the subject [Tan et al., 2003].

PBL has been experimented with in informatics courses, especially for programming. Reviews of experiences of using PBL show that it proved to be effective for introductory programming [see, e.g., Bawamohiddin and Razali, 2017; Oliveira et al., 2013; O'Grady, 2012]. In particular, Nuutila et al. [2008] successfully used the seven-step method in introductory programming courses. They found PBL very useful for replacing (at least partially) lectures. They respected the original PBL view from medical education: the focus is not on solving the problem but on autonomously set learning goals to acquire the knowledge needed to understand and explain "a case". Hence, they recognize that "*some aspects of the programming skills require supplementary learning methods*" like "*supervised programming exercises [...] to teach the use of programming tools and effective work practices [...and also, at the end of the course,] a larger programming project*" [Nuutila et al., 2008, p. 64]. However, they acknowledge that, because of the nature of informatics and software development, elements traditionally more resembling "project-based learning", like focus on solving open-ended, multi-answer problems or working on complex, real-world tasks, have been integrated by researchers in PBL for programming. On the other hand, Kay et al. [2000] warns that calling the small and well-defined exercises used in conventional courses 'problems' (as many authors tend to do, as well as us) is not enough to claim to use PBL.

Deek et al. [1998, p. 314] discusses the so-called "alternative method" specific for teaching programming, that is "*to introduce the problem in the lecture, engage the students in defining the statement of the problem, and allow the students to seek possible solutions independent of the programming language. Once the problem is solved, the language features necessary to implement the solution are presented. Finally, equipped with both the algorithmic solution, which the students develop, and the language syntax, the complete solution is translated into code and is then tested.*" This approach (which shows some, but not all, of the characteristics of PBL) is relevant to us because it shares with our design important features (e.g., a late direct instruction phase) of PS-I approaches (see the following 2.6.3.2). To it, we add the motivational aspect supported by the initial failure. Also, we do not explicitly distinguish between 'algorithmic design' and 'implementation' since the examples we propose are directly linked to language features. However, precisely because they are focused – as we will see – on the *necessity* of those constructs, we argue that our methodology can be helpful to stimulate *strategic knowledge* rather than just *syntactic knowledge* (see 2.3.1).

2.6.3 Activities and difficulties that prepare for instruction

In educational research, a growing body of literature argues that it is better not to start learning a concept from direct instruction.

For example, in *A Time For Telling*, Schwartz and Bransford [1998] describe a method in which undergraduate students analyse *contrasting cases* to develop prior knowledge that primes them to learn from direct instruction. Contrasting cases are presented side-by-side

and consist of small sets of data, examples, or strategies. They recognize that "*[n]oticing the distinctions between contrasting cases creates a 'time for telling'; [that is] learners are prepared to be told the significance of the distinctions they have discovered*". Ultimately, they show that engaging in "*analyzing contrasting cases [representing a target concept] provided students with the differentiated knowledge structures necessary to understand a subsequent explanation at a deep level [on that concept]*".

In teaching statistics to advanced students, Schwartz and Martin [2004] provide further evidence to support their method (i.e., *Invention*) to prepare students before instruction. They demonstrated the efficacy of invention activities preceding direct instruction, despite such activities failing to produce canonical understandings and solutions during the invention phase.

Furthermore, Bjork and Bjork [2011, p. 57] – discussing the results of several psychology studies in which they took part between 1975 and 2009, together with what they learned from their teaching experience (also as teacher-researchers) – affirm the following.

> Conditions of learning that make performance improve rapidly often fail to support long-term retention and transfer, whereas conditions that create challenges and slow the rate of apparent learning often optimize long-term retention and transfer.

They use the expression *desirable difficulties* to describe those challenging conditions that are "*desirable because they trigger encoding and retrieval processes that support learning, comprehension, and remembering*". Among these difficulties, Bjork and Bjork [2011] recognize the *generation effect*, that is "*the long-term benefit of generating an answer, solution, or procedure versus being presented that answer, solution, or procedure*".

About learners experiencing difficulties in preparatory activities, various forms of failure in activities preceding instruction are more and more investigated in educational research as a drive for better learning outcomes.

For example, VanLehn et al. [2003] conducted a study to help developers of intelligent tutoring systems understand which tutors' activities lead to success, confronting problem-solving episodes where tutoring does and does not result in learning a physics principle. They found that learning was infrequent when students did not reach an *impasse* (i.e., "*when a student gets stuck, detects an error, or does an action correctly but expresses uncertainty about it*") in problem-solving situations despite the tutor explicitly explaining the target concept. Conversely, "*when students reach an impasse, they discover that they need to learn something, so they may adopt a learning orientation*", making explanations more effective. Therefore, instructors should encourage impasses and delay instructional structure (e.g., feedback, questions, or explanations) until learners reach some form of failure and are consequently unable to proceed.

2.6.3.1 Productive Failure

Kapur and Bielaczyc [2012] leap forward in combining these two significant trends emerging from research – i.e., preparatory activities before instruction and failure as a drive to better prepare for learning – and propose the Productive Failure (PF) learning design. In the last ten

years, Productive Failure has generated a considerable amount of research, much of which seems to confirm its effectiveness [Kapur, 2016; Sinha and Kapur, 2019].

PF integrates four interdependent mechanisms: "*(a) activation and differentiation of prior knowledge in relation to the targeted concepts, (b) attention to critical conceptual features of the targeted concepts, (c) explanation and elaboration of these features, and d) organization and assembly of the critical conceptual features into the targeted concepts*" [Kapur and Bielaczyc, 2012].

PF learning design develops in two phases. The generation and exploration phase – when students engage in complex problem solving and generate multiple *representations and solution methods* (RSMs), followed by the consolidation phase – when teachers organize and assemble relevant students' RSMs into canonical RSMs. Three core design principles guide PF in order to embody the cited four interdependent mechanisms.

> 1. Create problem-solving contexts that involve working on complex problems that challenge but do not frustrate, rely on prior mathematical resources, and admit multiple RSMs (mechanisms a and b);
> 2. Provide opportunities for explanation and elaboration (mechanisms b and c); and
> 3. Provide opportunities to compare and contrast the affordances and constraints of failed or sub-optimal RSMs and the assembly of canonical RSMs (mechanisms b–d).
>
> [Kapur and Bielaczyc, 2012, p. 49]

These core principles translate into many specific principles to guide the implementation of the two phases. We briefly report only those relevant to our work, all related to the problem-solving phase. "*Designing the activity: 'sweet-spot' calibration of complex problems*" involves challenging but not frustrating students. "*Complexity of the problems*" requires complex problems' scenarios allowing multiple RSMs. "*Prior mathematical resources of students*" states that the problem complexity also depends on students prior knowledge, around which problems must then be built.

Productive Failure is often associated with problem-based learning (see 2.6.2). However, according to Falkner and Sheard [2019], "*[w]hile associated in structure with problem-based learning approaches, productive failure has a specific emphasis on the use of failure as a pivotal point in the learning process*".

2.6.3.2 PS-I approaches

In a broader perspective, Loibl et al. [2017] made a comprehensive attempt to summarize the features that define approaches in which preparing activities precede instruction, terming them PS-I. They consider Productive Failure and Invention as emblematic examples of PS-I approaches.

PS-I approaches involve an initial problem-solving phase in which learners are asked to develop solutions to a given problem. Then, the canonical solution and related target concepts are introduced in the following formal instruction phase. PS-I aims to most effectively combine

these two core learning activities (i.e., problem solving and formal instruction) while preserving their strengths and limiting their disadvantages.

PS-I is different from other instructional methods with prior instruction because it demands learners to engage in problem solving before receiving the target knowledge. At the same time, the explicit instruction phase sets PS-I apart from other inductive methods – e.g., discovery learning [Loibl and Rummel, 2013] and PBL (see 2.6.2), where different forms of support guide learners to discover the target knowledge. In PS-I, by contrast, problem solving is not designed to acquire the target knowledge since that is what the instruction phase is dedicated to. The originality of this approach lies not in its constitutive elements, i.e., inductive problem solving and explicit instruction, but in combining them in a specific order.

In the problem-solving phase, students face a problem that requires applying the knowledge they have yet to learn. For example, in Glogger-Frey et al. [2015], practice teachers received learning diaries excerpts and faced the problem of inventing criteria to evaluate the use of learning strategies in them. The targeted evaluation criteria for learning diaries were introduced and discussed later during the following instruction phase.

Loibl et al. [2017] recognize two main variants of the problem-solving phase. One is that of Productive Failure approaches [Kapur and Bielaczyc, 2012; Loibl and Rummel, 2013], which require presenting data in a rich story that does not highlight the deep features of the topic and for which the solution cannot be intuitively guessed. The other variant is that of Invention approaches [Schwartz and Martin, 2004; Glogger-Frey et al., 2015], in which contrasting cases serve to give students the relevant information.

They also distinguish two main variants in the implementation of the instruction phase. In Invention-like approaches, the canonical solution is presented without referring back to student solutions. In contrast, in PF-like approaches, the teacher starts from the students' solutions and uses them to explain the relevant elements of the canonical solution.

The efficacy of delayed instruction over instruction-first approaches has been shown across diverse learning domains and student populations by a substantial body of research [Loibl et al., 2017; Kapur, 2016]. Such efficacy lies particularly in three cognitive mechanisms that need to be activated in learners during the problem-solving phase: prior knowledge activation, awareness of their knowledge gaps, and recognition of deep features of the problem [Loibl et al., 2017].

In the last few years, given the consistent results supporting PS-I, research has investigated whether it is possible to increase this learning design's effectiveness further, examining different PS-I approaches. One of the leading research questions is whether the problem-solving phase should be scaffolded or not. According to Sinha et al. [2021], problem solving could be designed to nudge learners towards the canonical solution (i.e., success-driven scaffolding), towards sub-optimal solutions (i.e., failure-driven scaffolding), or to let learners experience failure without any form of scaffolding (in the problem-solving phase), resembling a straightforward Productive Failure design.

2.6.4 UMC approaches

2.6.4.1 Use-Modify-Create

Lee et al. [2011] proposed *Use-Modify-Create* (UMC) as a way to engage young learners in the development of *Computational Thinking*. It is a three-stage progression in which students are increasingly engaged in interacting with computational artifacts.

In the first phase, students *Use* an artifact created by someone else (e.g., play a computer game or run a simulation). In the second phase, students begin to *Modify* that artifact (e.g., starting from cosmetic changes and moving towards behavioral changes). Iteratively, they begin to *Create* their own (computational) artifacts.

> Moving through this progression, it is important to maintain a level of challenge that supports growth while limiting anxiety. As Repenning [...] notes, students can maintain their sense of cognitive flow [...] as they progress iteratively through a series of projects. In this work, students tackle progressively higher design challenges as their skills and capacities increase. Activities that were once "too hard" and were anxiety-inducing become possible with appropriate, incrementally challenging experiences. Conversely, boredom will set in if challenges don't keep pace with growing skills. [Lee et al., 2011, p. 35]

The three stages are not rigidly separated: they are fuzzy, with students going back and forth several times. Lee et al. [2011, p. 36] suggest that moving along the phases "*requires increasing levels of abstract representation and understanding*" and gives students an "*increasing ownership of their learning*".

Lee et al. [2014] propose different ways to introduce computational thinking in K-8. Among those, they mention both the transition from "puzzles to open sandbox" (exemplified with the Code.org K-5 curriculum, in which students have to solve a series of puzzles before reaching the open environment where they can create freely) and the use of UMC progression for computational science investigations (reporting observed transfer from using models in simulations to building models to solve community problems).

In middle school, Lytle et al. [2019] compare a UMC course with a "create-first" course on science simulations with block programming. The pupils in the UMC group perceived the activities to be more accessible and felt a greater sense of ownership of the work. The teachers in the UMC group gained confidence while delivering the UMC sequence, requiring less and less support from researchers and even managing to add new tasks. In contrast, the teachers in the other group felt the need for more scaffolding.

2.6.4.2 PRIMM (Predict-Run-Investigate-Modify-Make)

Stemming from the original UMC proposal, various specializations have been proposed. PRIMM (*Predict*, *Run*, *Investigate*, *Modify*, *Make*) is a informatics-specific model for teaching text-based programming in middle and high school. Sentance and Waite [2017] present PRIMM as primarily built on UMC research but also strongly influenced by Vygotsky's sociocultural theory of learning (more on this in Sentance et al. [2019b]), as it leverages

teacher mediation, language, and dialogue to aid understanding. Indeed, PRIMM particularly emphasizes reading and tracing as fundamental activities before writing. It expands the *Use* stage of UMC into *Predict*, *Run*, and *Investigate*, thus increasing scaffolding at the beginning. Students are asked to read the code and predict how the (part of the) program will work before executing it (*Use*). Then they investigate the program structure with program comprehension tasks at different levels of abstraction, which is in line with the original goal of PRIMM, i.e., teaching text-based programming to secondary students. Unlike UMC, which is more CT-inspired and STEM-inspired, PRIMM is indeed informatics-specific. The subsequent, more open phases are also scaffolded: *Modify* requests are narrow scope, and *Make* assignments have explicit and limited objectives.

A vast 2018 research study (14 teachers and 300 students) shows improved understanding by PRIMM students and positive feedback from teachers. Sentance et al. [2019a] report that PRIMM helped structure productive lessons and engaged low-achieving students; authors argue that PRIMM can be easily adapted to a wide range of abilities and objectives.

2.6.4.3 TIPP&SEE

To shed light on student learning during the *Use* and *Modify* stages, Franklin et al. [2020a] conducted an extensive study on 536 students (age 9-14) on a Scratch-based UMC curriculum, *Scratch Encore*, which attempts to balance structure with flexibility and creativity. Students engaged in UMC activities following the TIPP&SEE algorithm (*Title*, *Instructions*, *Purpose*, *Play*, & *Sprites*, *Events*, *Explore*), which provides more scaffolding to the UMC stages. This approach effectively balanced structure and student agency.

Salac et al. [2020] conducted another study that showed that students who followed TIPP&SEE performed significantly better on all intermediate and difficult assessment questions. In addition, the authors define more formally the TIPP&SEE approach as "*a metacognitive learning strategy that further scaffolds student learning*" (in the *Use* and *Modify* phases, as well as PRIMM) by trying to balance free exploration with learning, especially to overcome the *play paradox*[10]).

Like PRIMM, it is influenced by Vygotsky's theories. Specifically, it seeks the 'Zone of Proximal Flow' by providing more scaffolding in the *Use* and *Modify* stages to benefit struggling students while trying to create an optimal experience, that is through "learning experiences that are not too challenging as to overwhelm students, but not too easy as to lead to little learning". However, compared to PRIMM (aimed at text-based programming), it is designed for Scratch, making it an implementation of UMC for young students with no programming experience. Similarly to PRIMM, TIPP&SEE uses program reading and comprehension techniques: its structure is explicitly based on the studies on 'reading comprehension' in natural languages (especially on 'previewing' and 'text structure' constructs). In a later study, Franklin et al. [2020b] tries to shed light on the conduct of the TIPP&SEE students, showing different and more effective learning behaviors than the control group.

[10]The *play paradox* is summarized effectively by journalist Hulbert [2007], who, in her article 'The Paradox of Play', states that "*the crusade to restore the primacy of play runs the risk of eroding the very playfulness the crusaders are eager to see more of*".

In general, both PRIMM and TIPP&SEE methodologies have been tested in different contexts [e.g., Sentance et al., 2019a; Franklin et al., 2020b], proving to be engaging, effective even for low-achieving students and appreciated by teachers. Furthermore, using PRIMM and TIPP&SEE seems to increase confidence in teaching introductory programming for inexperienced teachers.

2.7 Languages for teaching programming

One form of optimal guidance concerns focusing on the type of programming language chosen in an introductory programming learning path. Up to this point, every consideration we made has been independent of the programming language. However, it is logical to assume that the language used to learn programming is a very impactful variable in learning outcomes. Indeed, there is a lengthy and still active discussion in informatics education and its community involving researchers, educators, teachers, and to some extent, legislators about which language is most appropriate for teaching programming.

It is out of the scope of this thesis to get into this discussion. However, we emphasize that simple languages with fewer syntactic elements are increasingly emerging as more suitable for learning programming. For example, Wainer and Xavier [2018] compared the same course in C and Python and found that Python students had a higher probability of passing the course and better grades on the exam. Moreover, learning programming with a simple language does not seem to interfere with learning a more complex language later. For example, Mannila et al. [2006] found no particular disadvantage in students who had studied Python before switching to Java. In addition, two studies argue that Python is good preparation for learning C++ later [Enbody et al., 2009; Enbody and Punch, 2010].

These indications in favor of using Python for teaching programming because of its lighter syntax confirm that learning syntax is one of the most common and tough challenges among novices. Denny et al. [2011] discovered, for instance, that even the best students submit source code with syntax errors half of the time, and fragile students do so at a 73% rate. Altadmri and Brown [2015] examined 37 million compilations by 250.000 students and discovered that mismatched brackets, the most frequent syntax error, appeared in almost 800.000 compilations. Stefik and Siebert [2013] demonstrated the difficulties that beginners encounter in understanding syntax by finding that popular programming languages Java and Perl are not easier to understand than a random language. It is interesting to note that students find learning syntax more difficult than teachers [Piteira and Costa, 2013], which may help explain why many programming classrooms make little effort to explain syntax explicitly. According to Lister [2016], learning programming concepts, syntax, and problem-solving simultaneously is likely to raise students' cognitive load too high. As the *cognitive load* increases, the *Learning Edge Momentum increases* as well (see 1.4). This situation may result in novice programmers acting like younger students while learning to write [Hermans, 2020].

Broadening the perspective on learning languages in general (beyond just programming languages), it was found that learning proper punctuation takes time [Fayol and Lemaire, 1989], and novice students frequently temporarily forget what they already know [Simon, 1973].

Long-term practice can hasten the process of mastering proper punctuation [Leonard, 1930]. Additionally, research demonstrates the value of repetition in language learning. For instance, a word must be read seven times in order to be retained in long-term memory [Verhallen and Verhallen, 1994]. Since learning a programming language and learning a natural language have many similarities (e.g., requiring students to learn both semantics and syntax), Hermans and Aldewereld [2017] and Portnoff [2018] suggest that programming education could be improved by using instructional techniques used in natural language teaching.

Building on this kind of assumption, some informatics education research has tried to define specific languages for learning programming as an ideal scaffolding, that optimal guidance we have written about and which is the focus of this review. The search for optimal guidance can be declined (other than the search for specific methodologies that balance autonomy and support) as the development of programming languages for learning, with precise characteristics to benefit novice students learning to program. Previous research has identified three distinct approaches to languages for introductory programming [Brusilovsky and Others, 1994], and we report them here as described by Hermans [2020].

> **Mini-languages** Mini-languages are languages that are small and especially designed to support learning to program. A well-known example of a mini-language is Papert's LOGO [Papert, 1980]. More modern examples of mini-languages are Scratch [Resnick et al., 2009] and Karel the Robot [Becker, 2001]. Mini-languages are said to *"provide a solid foundation for learning a general purpose language"* [Brusilovsky et al., 1997], but learning a mini-language can also be a goal in itself, leading to the acquisition of algorithmic thinking.
> **Sub-languages** In the sub-language approach, programming is taught to novices using only a set of commands from a bigger programming language, which typically is one that is used in practice such as Pascal or later, Java. Initially the idea of sub-languages was not to have them successively grow, but to simply select a subset to teach. Examples are Helium, a subset of Haskell for educational purposes [Heeren et al., 2003], and MiniJava [Roberts, 2001] and ProfessorJ [Gray and Flatt, 2003] for Java.
> **Incremental approach** The incremental approach first teaches a small subset of a programming language where each subset introduces new programming language constructs. This approach was first implemented for PL/1 by Holt et al. [1977] and later also applied to Fortran [Balman, 1981] and Pascal [Atwood and Regener, 1981]. Some other versions of incremental teaching used subsets that were explicitly not arranged as a hierarchy where "higher level" contained the "lower level" but instead divided the language into overlapping languages like chapters in a textbook would [Tomek et al., 1985]. More recently, DrScheme used a similar approach for Scheme [Felleisen et al., 2004]. [Hermans, 2020, p. 260]

Building on the incremental language approach and deeply inspired by the spiral approach to learning programming by Shneiderman [1977], Hermans [2020] developed a gradual language, *Hedi*. Shneiderman [1977] proposes the "spiral approach" to learning programming to

accommodate the cognitive limits of learners. He suggests that programming courses should begin with a small amount of syntax and simple semantics, then progress to the simplest forms of the assignment statement and arithmetic expressions step by step. "*At each step the new material should contain syntactic and semantic elements, should be a minimal addition to previous knowledge, should be related to previous knowledge, should be immediately shown in relevant, meaningful examples and should be utilized in the student's next assignment. This is the spiral approach.*" [Shneiderman, 1977, p. 195] Hedi embodies the spiral approach, and it is based on some of the most effective methods used in natural language education to teach punctuation to younger students. Similarly to how pupils learn their first written language, Hedy begins as a small, elementary programming language whose syntax rules gradually evolve until students are programming Python. In other words, the growth seen in the incremental approach in Hedi also applies to the syntax, which goes from being very simple to gradually introducing new elements (brackets, quotes, indentation) and rules. Hedi offers a variety of levels, each with new commands and a more complex syntax. We hypothesize that, despite what emerges from the literature and that we, too, agree that an opportunely "simplified" language helps to support novices in learning programming, it may not be necessary to develop new languages to achieve this end. Given the existence of languages such as Python, which are already very abstract and syntactically lightweight (compared to older but still dominant languages such as C++ and Java), the challenge of finding the right language for learning – simple enough, expressive and incremental enough – can also be played out on the more agile terrain of notional machines. In other words, instead of designing and developing a new language, one can just define an appropriate notional machine (see 2.3.4) starting from an existing language, textual (e.g., Python) or graphical (e.g., Snap!). Similarly, rather than developing an incremental language, a progression of notional machines can be defined to offer novice students the right balance between expressive freedom and support, between power and simple syntax, along all stages of the introductory programming learning path. Given the greater simplicity in defining a notional machine, which requires careful analysis and design but has no implementation costs, the granularity of support could be extremely precise. Notional machines can change at every learning unit or even at each activity for scaffolding that is always optimal and adherent to the growing yet building students' capabilities. Such incremental support would be more costly to achieve through a programming language, which would have to be designed and developed in multiple versions (e.g., a progression of sub-languages), resulting in a multiplication of efforts. Alternatively, the language chosen to teach programming would have to be incremental by design, making it more complex to design and develop. On the other hand, moving the challenge of the optimal level of scaffolding entirely to the notional machine front requires using an existing programming language and defining notional machines feasible with that specific language and also supported by existing and usable development environments for learning.

Finally, to show the variety of approaches to programming languages for learning, *task-specific programming languages* and *teaspoon languages* (one of the latest and more specific forms of task-specific languages proposed by Guzdial [2021]) have recently emerged in informatics education. Although they enable novices to program, that is, to specify with due precision the computational processes to be performed by a computational agent, they do

not (nor could they) aim to teach students how to program. They are made to be extremely simple and learned in about 10 minutes. They aim to support non-programming learning tasks teachers (typically non-informatics) want students to accomplish by leveraging computers (for automaticity, computational power, and because they can provide meaningful real-world scenarios). Thus – although their use may convey some general principles about programming – they are presented and discussed in part II of the literature review (where the focus is indeed on general principles of informatics and its intedisciplinary aspects), specifically in 5.1.3.

2.8 Emergency remote teaching of CS1

Online learning has grown dramatically and has been extensively studied by educational researchers. As observed by Rudestam and Schoenholtz-Read [2010, p. 11], institutions chose modalities of online teaching coherent with "*their dominant pedagogical principles and historical attitudes towards education*". Traditional universities naturally tend towards a synchronous online approach, while newer distance universities tend towards asynchronous approaches. Asynchronous formats offer flexible scheduling, thoughtful participation, and richer and inclusive interchanges [Rudestam and Schoenholtz-Read, 2010]. However, some profit institutions are accused of using online learning as an excuse to provide low-quality instruction to many students. By contrast, synchronous approaches offer greater spontaneity and social interaction [Rudestam and Schoenholtz-Read, 2010]. In either case, students agree that the instructor is one of the critical elements of the learning experience, which is confirmed for online courses (see [Rudestam and Schoenholtz-Read, 2010; Bower, 2009]).

Some research found that synchronous approaches provide "*learning opportunities to collaborate [. . . and] better course and program completion rates for students who participate in synchronous interactions with their teacher and peers rather than relying solely on asynchronous communication*" [Bower et al., 2014, p. 15]. Bower et al. [2014, p. 15] recognizes that synchronous learning allows "*remote participants to experience an instructor's lesson, ask and answer questions, offer comments in class and generally allow engagement 'in a similar manner to on-campus students' [. . .], providing them with both access to knowledge and social interaction*"[11]. According to Ihantola et al. [2020], "*most studies [outside informatics: like medicine, physics, engineering] found that video lectures are at least as beneficial for learning as live lectures [. . .], even though students might prefer live lectures [. . .]*". However, in their multi-year study on an Algorithms and Data Structures course, they found that students who preferred to attend face-to-face lectures outperformed in attendance rate and grades those who favored watching recorded lectures.

Online learning poses challenges for both students and instructors. According to Peachey [2017], students face a sense of isolation and the need for self-discipline and are required to develop technological literacy. Instructors who engage in synchronous online teaching need skills to establish relationships without using paralinguistic features and gestures. Also, paralinguistic clues are not available for students not sharing their webcams, so instructors cannot use those clues to check students' understanding. The webcam is considered a

[11] Bower refers to blended learning, but talks about the distance students, hence closely matching our situation.

fundamental tool to reduce the sense of isolation and develop rapport. Another essential tool is the chat, which helps check class comprehension by quick polls or short student answers [Peachey, 2017, p. 146-148]. Moreover, "*[s]ocial presence and responses can be facilitated by a variety of emoticons and voting features providing a mix of communication and participant management modes*" [Bower et al., 2014, p. 19].

Generally, web conferencing use in online education has been positively correlated with student satisfaction, higher marks, and better learning experience [Bower, 2011], also inside informatics [Bower, 2009; Coffey, 2010].

While many blog posts Guzdial [2020] have discussed *informatics emergency remote teaching* over the past year, giving teaching advice, there is still not much research published on this specific topic in literature.

On the one hand, informatics educators seem to have been more prepared than other educators for the online transition, probably because of their familiarity with technology [Crick et al., 2020; Malmi, 2020]. On the other hand, they expressed concerns regarding online teaching – both abstract and mathematical concepts in informatics and practical and collaborative activities like programming projects, fearing a fallback to more traditional and transmissive teaching styles [Crick et al., 2020].

Challenges for community colleges during pandemic Tang and Servin [2020] include difficulty providing essential services (e.g., computers, healthy food, places to concentrate), need for teachers' professional development, difficulty adapting hands-on labs and courses never taught online before, caring for students with special needs, avoiding plagiarism. Authors claim that some students strongly prefer being face to face because of these limitations. However, the pandemic has also been an opportunity to update curriculum, methodologies, and assessments. Our qualitative analysis is in line with these observations.

2.9 Participatory design with teachers

A *participatory design* approach includes the perspective of users and stakeholders within the design process of the products they will use [Bjerknes and Bratteteig, 1995]. Derived from *Scandinavian Cooperative Design* in the 1960s, this approach was initially developed in the context of urban planning and applied in other fields such as software development and product design [Bødker et al., 2000]. This approach has been used for over fifty years in various fields, such as information technology, human-computer interaction, and workplace studies.

The participatory design utilizes methods that support mutual learning among participants to design contextually relevant design solutions cooperatively. The process is usually supported by a *facilitator*, who works to make meetings and group interactions easier and more effective. Facilitators are neutral in terms of content, their focus being the process itself.

Druin [2002] analyzes how users can take part in design processes and defines a framework with a set of roles: users, testers, informants, and design partners. *Users* are the primary audience for an existing technology/artifact; their use of the artifact can be investigated to improve it. *Testers* use an artifact that has yet to be released with the aim of developing it for a larger audience. *Informants* play an active part throughout the design process and

provide input before, during, and after the product is developed. *Partners* are acknowledged as legitimate decision-makers and promoted to a role that equals that of the designers and researchers.

Participatory design methods such as *co-design* have also been applied in educational contexts. Roschelle and Penuel [2006] analyze the key components of co-design, defined as "*a highly-facilitated, team-based process in which teachers, researchers, and developers work together in defined roles to design an educational innovation, realize the design in one or more prototypes, and evaluate each prototype's significance for addressing a concrete educational need*" [Roschelle and Penuel, 2006, p. 606]. For a recent review of participatory design studies involving teachers, see [Tuhkala, 2021].

In informatics education, participatory methods are not frequently used. Notable recent exceptions are Coenraad et al. [2022] and Novak and Khan [2022], where co-design is used with teachers and students to integrate their perspective in their future educational material/approaches, with a particular focus on the cultural relevance of the curriculum. In particular, the work by Coenraad et al. [2022] is relevant to our research (see chapter 12), being about the design of a curriculum that uses the UMC methodology with Scratch.

A final note. 7.5 covers educational theories (i.e., *Theory of Didactical Situations*) and methodologies (i.e., *Didactical Engineering*) we used in our research, which, like participatory design, do not come from informatics education. Since we exploited them – similarly to what we did with participatory design – to design learning environments and activities that could meet teachers' and students' needs and expectations, 7.5.3 briefly shows the general relationships between them and participatory design.

Part II

Literature Review – Part 2

Introduction to part II

The common thread running through this part of the literature review is access to informatics not through its more traditional programming route but through pathways that are less professionalizing and technical and more suitable for a broad, potentially younger, population. Such pathways are especially relevant from a cultural and citizenship perspective. In particular, the perspective of the big ideas of informatics is central and helpful in guiding educators and students in recognizing the culturally and scientifically relevant elements of informatics (beyond its technological dimension). Those who wish to approach it from a scientific and professional perspective would also benefit from this perspective. Beginning with a reflection on computational thinking and its relationship with informatics, frameworks, approaches, and even topics (such as cryptography) helpful in communicating the interdisciplinary value of informatics to a broad population are presented.

Chapter 3

Informatics for All

In the previous part I we explored the most traditionally practiced access to informatics, through learning introductory programming, appropriate for older or early college students (indeed, often the literature reviewed is about CS1 courses). On the other hand, this part aims to review the literature that tackles access to informatics for a broader population, also starting with younger students. The primarily dimension here is citizenship: informatics is taught for its cultural value, which is necessary to correctly interpret the world we live in and act in it. As anticipated in 1.1, the citizenship dimension does not exclude that of participation, which can come as a consequence.

Indeed, Curzon et al. [2018, sect. 8.1] argue about *computing* (used as a synonym of informatics) the following.

> Computing does not just develop sophisticated skills such as programming. It is also a rigorous academic subject akin to physics or history, consisting of a rich conceptual framework, both around programming and more generally. Computational thinking is the core skill set that students develop by studying computing. This too is rich with concepts such as abstraction, decomposition and generalization. Understanding these concepts is an important part of being able to do computational thinking. For many aspects of the subject, a solid understanding of earlier concepts is a prerequisite for understanding later ones [...], especially with respect to programming. [...]
>
> The learning of [computing] concepts is therefore a critical part of the subject.

As the expression *computational thinking* emerges and given its popularity in educational discourse today, it is worth clarifying it and briefly discussing its relationship to informatics. This will help to clarify the context of informatics education for younger students in the early grades and to understand the need to teach the fundamental concepts of informatics from a cultural and citizenship perspective.

3.1 Informatics and Computational Thinking

Expressions such as *computational thinking* and *coding* are becoming increasingly common in the school world, used by trainers, textbooks, institutional documents, and national laws. These terms risk generating misconceptions in teachers [Corradini et al., 2017b; 2018] who have not received rigorous education in the fundamental concepts of informatics, the discipline in which these expressions are located. However, the expression *computational thinking* originates in mathematics education, probably first used by Seymour Papert in 1980 [Papert, 1980]. Papert advocates using programming as an interdisciplinary tool, useful for students to make abstract concepts concrete and build mental models of what they are learning. Starting from Piaget's *constructivism*, Papert and Harel [1991] theorize a revised version: *constructionism*.

> Constructionism [...] shares constructivism's connotation of learning as "building knowledge structures" [...]. It then adds the idea that this happens especially felicitously in a context where the learner is consciously engaged in constructing a public entity, whether it's a sand castle on the beach or a theory of the universe. [Papert and Harel, 1991, ch. 1]

With its potential to simulate worlds, the computer provides each student with different and meaningful (cognitively and also emotionally) construction materials. Computational thinking "à la Papert" highlights the high interdisciplinary value of informatics, which allows the execution of constructed abstractions, that is, the simulation of phenomena and concepts to be learned. "*Programming [is] an interdisciplinary tool for learning (also) other disciplines*" [Lodi and Martini, 2021, p. 884]. Moreover, in Papert's vision, computational thinking is always referred to a specific *executor* (in this, it differs from *algorithmic thinking*). Having an external automaton solve a problem, an automaton limited to a narrow set of instructions, which does not make human inferences, is an excellent exercise for understanding the problem thoroughly.

On the other hand, in today's society, permeated with digital devices and applications and influenced by algorithms governing multiple aspects, it is essential to learn informatics's scientific and cultural foundations [Lodi et al., 2017; Lodi and Martini, 2021]. Such learning is the goal of the movement for computational thinking in school, which originates in the second half of the 2000s from the highly influential article of Wing [2006]. She defines computational thinking as *computational problem solving*, that is, formulating solutions in a way that is executable by an information processor. We argue that computational thinking "à la Wing" is precisely "*the natural sediment of disciplinary learning of computer science*" [Lodi and Martini, 2021, p. 889], or, in other words, "*the conceptual sediment of [its] disciplinary learning*" [Lodi and Martini, 2021, p. 898].

For a more comprehensive review of computational thinking, including its possible definitions, an analysis of its nature and elements, its practical implications, and the major contributions from research, please refer to Curzon et al. [2019].

3.1.1 Why computational thinking belongs in informatics

Chick et al. [2009], discussing *signature pedagogies*[1], assert that "*effective teaching results from core values and principles of our courses and of our disciplines, rather than from generic views of learning. [. . .] higher-level thinking is inhibited by such generic conceptions and lays the groundwork for questions about the central values, habits, and ways of thinking within their disciplines*" [Chick et al., 2009, p. 4].

We are aware that disciplinary thinking cannot regard only domain-specific "values and principles" and must include specific and general aspects. Indeed,

> [. . .] domain-specific thinking and domain-general thinking are not dichotomous, as thinking itself is a complex process involving many different components. [. . .] Domain-specific thinking is often characterized in terms of its disciplinary content but also involves more general cognitive components. [. . . For example,] some aspects of mathematical problem solving are largely discipline specific (e.g., the knowledge base), some heavily discipline oriented (e.g., strategies and beliefs), some much like discipline domain-general (e.g., metacognition). [Li et al., 2019, p. 8]

However, as Lodi [2020a] points out, we keep noticing that educators and policymakers focus only on the more general aspect of computational thinking.

Voogt et al. [2015, p. 718], analyzing various definitions of computational thinking used in compulsory education, note a tension between "*the 'core' qualities of CT versus certain more 'peripheral' qualities*". When it comes to computational thinking, these peripheral qualities overlap strongly with the so-called "soft skills". Voogt et al. [2015, p. 719] also warn that an exaggerated focus on the soft skills of computational thinking at the expense of its core informatics qualities "*runs the risk of diluting the idea of CT, blurring and making it indistinct from other 21st century skills*".

However, with the growth of the computational thinking movement in education, numerous unsubstantiated claims about the effects of computational thinking are being spread. Lewis [2017] reports superficial and optimistic beliefs about computational thinking. For example, it would automatically transfer to logical thinking, foster perseverance development, help achieve better results in science and mathematics, and even solve almost any practical problem in everyday life. While most of these claims are unsupported by research, they "*appear in blog posts, opinion pieces, and other 'grey literature'*" [Duncan, 2019, p. 32]. About this trend, Denning et al. [2017, p. xiii] remark as follows.

> [Computational thinking] is sometimes portrayed as a universal approach to problem solving. Take a few programming courses, the story in the popular media goes, and you will be able to solve problems in any field. Would that this were true! Your ability to solve a problem for someone depends on your understanding of their context in which the problem exists. For instance, you cannot build simulations of aircraft in flight without understanding fluid dynamics. You cannot

[1] A signature pedagogy "*engage students in the ways of knowing, the habits of mind, and the values shared by experts in [a] field*" [Gurung et al., 2009, p. xvii].

> program searches through genome databases without understanding the biology of the genome and the methods of collecting the data. Computational thinking is powerful, but not universal.

In response to this trend, we believe computational thinking develops within informatics learning. In particular, its development should be pursued through learning its "core values" (i.e., the fundamental principles of informatics) rather than by pursuing its "peripheral qualities" (the very appealing yet generic soft skills). In this, we agree wholeheartedly with Lodi, who states the following.

> [N]on-specialist teachers that most probably never studied informatics in their schooling or training may tend to stick to some "general versions" of the listed characteristics (and especially on the peripheral qualities), not necessarily related to informatics. We believe, by contrast, CT must be understood inside informatics: while many characteristics are (more or less apparently) shared with other disciplines, we need to focus on their specific "informatical" instantiation. [Lodi, 2020a, p. 10]

3.1.2 Computational thinking and coding

The term *coding* often refers to a playful introduction to programming activities. Technically, coding is the act of writing an informatics program and denotes only the more mechanical phase of writing code. As informaticians, we know that *programming* is much more than just coding since it consists of analysis, design, implementation, testing, and debugging. Coding activities are trendy in schools today. On the other hand, if these activities aim to develop computational thinking (that is, to solve problems by making use of the fundamental ideas of informatics), they must necessarily require expressing the solutions in a language that is understandable and executable by an automaton.

Nowadays, in the lower grades of schools, trainers and textbooks propose activities like *pixel art* or *coding robots* on grids as coding activities. Hands-on activities like these can be excellent first steps into informatics and toward computational thinking, only if part of a conscious and scientifically-sound learning path. Teachers should not just make children play, although the game and fun dimensions are essential. They must highlight the relationship between such activities and the operation of computers. For example, *pixel art*[2] should not be (only) a manual, mechanical activity of coloring but an experience of encoding and decoding information in a precise and systematic way. It should help to understand how computers represent images, more generally, that computers represent all information by numbers (bits).

On the other hand, it is possible to do problem solving without bothering informatics, just as it is possible to learn to orient oneself in space without a robot on a grid. Similarly, it makes no sense to trace all activities that use logical reasoning back to informatics; logic is a fundamental element of "thinking like an informatician" only when used to make an

[2]Pixel art is a form of digital art that utilizes graphical software to create images using pixels as the only constitutive element. It is often proposed as an educational activity for children via simple educational software, or without computers, as an *unplugged* activity requiring just paper and colors.

automaton to process information. Educational platforms such as *CS Unplugged*, *Code.org*, and *Scratch* offer an excellent entry point for teaching informatics concepts as early as elementary school. Also, as mentioned, computational problem solving can be used to teach other disciplines effectively.

To summarize, when we talk about computational thinking, we must always consider two levels. On the one hand, computational thinking is nothing but the cultural sediment of informatics: we teach informatics principles, and what remains is computational thinking ("à la Wing"). On the other hand, we must not forget Papert's dimension. Those informatics principles cannot be "external" notions; however, they must be transformed into meaningful building materials in the hands of students so that they concretely realize (and thus understand) concepts from other disciplines as well.

Chapter 4

Big Ideas

One way to respond to and capitalize on the challenge of teaching computational thinking is through the big ideas approach. Research on big ideas in science education has begun to show how focusing on the scientific core of a discipline (rather than peripheral concepts and skills) is beneficial for learning.

In the following subsections, we describe what big ideas are, the science education context in which they emerge, the reasons for and benefits of their formulation, and the general principles guiding their distillation from an entire body of scientific knowledge. Then we also introduce the more recent big ideas of informatics by Bell et al. [2018].

4.1 Big ideas of science

In very few words, the big ideas of science are "*the key ideas that students should encounter in their science education to enable them to understand, enjoy and marvel at the natural world*" [Harlen et al., 2015]. There are at least two major frameworks for big ideas in science education.

The one by Chalmers et al. [2017] sits in the context of STEM education and aims to meet the interdisciplinary challenges that STEM integration requires. In response to the struggle to produce in-depth learning (found in the literature), they propose using a framework to scaffold teachers' development of integrated STEM curriculum units. The framework is based on three types of big ideas: "*within-discipline big ideas that have application in other STEM disciplines, cross-discipline big ideas, and encompassing big ideas*" [Chalmers et al., 2017].

The other framework arises in science education in response to the widespread perception among students that science curriculums, usually overcrowded and fragmented, are little more than a series of unconnected facts with very circumscribed relevance. According to Harlen et al. [2010], a group of experts in science education, "*[p]art of the solution to this problem was to conceive the goals of science education, not in terms of the knowledge of a body of facts and theories, but as a progression towards understanding key ideas – 'big ideas' – of relevance to students' lives*". The result of their work is a first book, 'Principles and Big Ideas of Science Education' [Harlen et al., 2010]. The book immediately receives much attention, prompting the same authors to a second work to extend the previous one,

resulting in the publication of their second book on the matter, 'Working with Big Ideas of Science Education' [Harlen et al., 2015].

Our research on big ideas is based mainly on the theoretical and methodological framework proposed by Harlen and colleagues in their books (particularly the second).

4.1.1 Context and motivation

Harlen et al. [2015, p. 1] introduce the rationale of their work as follows.

> To prosper in this modern age of innovation requires the capacity to grasp the essentials of diverse problems, to recognize meaningful patterns, to retrieve and apply relevant knowledge. [...] Science education [...] needs to take account of changes in the work place that require ability to link science with engineering, technology and mathematics (STEM), the urgent need for attention to major global issues such as the adverse impacts of climate change, the positive and negative influences of student assessment and the growing contribution of neurosciences to the understanding of learning. All of these add to the reasons for the development of big ideas to provide a framework for decisions about science education.

In particular, they identify specific reasons why a framework for developing big ideas of science is timely.

- Address learners' perception of science as a fragmented set of facts and theories of little relevance to them.
- Provide a framework for school activities that support learners in explaining things they consider relevant.
- Provide a tool for selecting from the massive body of potential curricular content.
- Organize the development of scientific curriculums on progression toward big ideas.
- Concretely support the demand in science education for inquiry-based pedagogy.
- Foster recognition of the connections of STEM disciplines with multiple contexts of everyday life.
- Support learning systems of connected ideas for understanding the world and our experiences rather than a series of disconnected knowledge. Indeed, new evidence from neuroscience shows that connected ideas are more easily and readily used in new situations.

Harlen et al. [2015] also recognize two additional perspectives in which the framework of big ideas is relevant, perspectives that pose open challenges in education today.

- *Student assessment.* Traditional tests and exams often present a series of disjointed questions or problems, encouraging the teaching and learning of disconnected notions. The impact of this way of assessment holds back the development of an interconnected system of key knowledge and skills in students. The framework of big ideas can help move in the right direction, but it must be supported by change on the assessment front.

- *Teacher education.* Teachers must consider how individual lessons' objectives fit into a larger framework of broader and stronger ideas that can help students make sense of a wide range of related phenomena and events. This general development direction is necessary for selecting content and constructing learning pathways. It also clarifies what teachers need to observe and look for in students' actions, informs their decisions about feedback, and helps them adjust formative assessments. This is particularly challenging for primary school teachers (who teach all disciplines) but also for some secondary school teachers who teach science subjects without being able to study them in depth. Often the scientific education teachers received did not allow them to develop big ideas. Instead, teachers should also have this opportunity to be able to help students understand the big ideas.

4.1.2 Benefits of big ideas

According to Harlen et al. [2015], there are two main dimensions in which the benefits of big ideas arise, the individual and the collective.

Big ideas, distilled from science to be powerful and widely applicable, can help grasp the essential properties of events or phenomena, even without knowing all the details. For students, understanding the big ideas of science can give them the satisfaction of making sense of the world (seeing patterns in different situations and connections between them) and understanding the purpose of scientific research and its impact on our lives. This can revitalize motivation for learning during and beyond formal education. Moreover, understanding the world better helps individuals realize their right to citizenship because it helps them make personal decisions that affect their health, also in relation to the environment, and career choices.

Making informed individual choices (e.g., regarding health, energy use, and environmental care) has broader implications than the personal sphere because of the long-term impact of human activities on the environment and society. By appreciating the importance of science, it is possible to recognize how the use of scientific knowledge can have both positive and negative effects. In this sense, science education has a crucial role: fostering an understanding of the problems that lead to inequality in the world and a willingness to address them by taking an active role.

4.1.3 How to distil big ideas

Harlen et al. [2015] report that the approach to science education through the development of big ideas described in the first book 'Principles and Big Ideas of Science Education' has been

widely accepted and embraced by the educational community. For the second book, they question the principles and criteria for selecting and distilling big ideas, ultimately confirming their validity.

Firstly, two dimensions must be considered when deciding whether an idea (a scientific fact, concept, or theory) is a *big idea*.

> Range – whether to include scientific attitudes and dispositions towards science and what are variously called skills, practices, competences or capabilities as well as core scientific ideas.
>
> Size – how broad a compass of phenomena the ideas should explain, recognising that the larger the idea, the more distant it is from particular phenomena and the more abstract it therefore appears to be. [Harlen et al., 2015, p. 11]

The *size* dimension resembles the *horizontal applicability* of Schwill [1994][1]. Discussing how to develop the fundamental ideas of informatics, he argues that fundamental ideas should have broad applicability.

In addition, we briefly report the principles that Harlen et al. [2015, p. 14] suggest should be used to distill big ideas from the "raw" and complex scientific knowledge from which they originate.

> [B]ig ideas should:
>
> - have explanatory power in relation to a large number of objects, events and phenomena that are encountered by students in their lives during and after their school years
> - provide a basis for understanding issues, such as the use of energy, involved in making decisions that affect learners' own and others' health and wellbeing and the environment
> - lead to enjoyment and satisfaction in being able to answer or find answers to the kinds of questions that people ask about themselves and the natural world
> - have cultural significance – for instance in affecting views of the human condition – reflecting achievements in the history of science, inspiration from the study of nature and the impacts of human activity on the environment.

4.1.4 Progression to teach big ideas

According to Harlen et al. [2015], the best way to teach and learn a big idea is through a progression. Each progression moves toward its target big idea, always starting with a "small idea" (the core of the target big idea in a short and easy-to-understand form). It is a progression of four stages, each with its version of the same big idea, more and more complete and general.

[1] We will discuss Schwill's work later (4.2) as it concerns the fundamental ideas of informatics.

> [W]e begin with the small and contextualised ideas that children in the primary or elementary school, through appropriate activities and with support, will be able to grasp. These are followed by ideas that lower secondary school students can develop as their increasing capacity for abstract thinking enables them to see connection between events or phenomena. As exploration of the natural world extends in later secondary education, continuation of this creation of patterns and links enables students to understand relationships and models that can be used in making sense of a wide range of new and previous experiences. [Harlen et al., 2015, p. 19]

They adopt a narrative approach to describe the progression of the idea evolving from small to big. However, what the authors call *the small idea* (and the same applies to all the ideas in the progression) could be described as *the target big idea* at a different abstraction level, chosen to be suited for students' knowledge and abilities. From this perspective, progressions consist of increasing levels of abstraction.

Progressions to be effective must be based on a solid understanding of each idea at each stage. The stages are designed to be reached at different school levels and according to students' age. However, many contextual factors (e.g., the pedagogy used) can influence the speed of the progression and the potential destination.

Progressions realize the *vertical applicability* of Schwill [1994]. Considering how to develop the fundamental ideas of informatics, he argues that fundamental ideas should always make sense to students at the various levels of their education.

4.2 Big ideas of informatics

In a global landscape where teaching informatics (sometimes under the name of *computational thinking*) increasingly enters curriculums even from the early years of schooling, the tendency found among teachers to focus on the details while losing sight of the bigger picture prompted Bell et al. [2018] to formulate the big ideas of informatics for pre-tertiary (K-12) education. This tendency is furthered by the fact that the informatics reference curricula are overflowing with topics [see, e.g., Brown et al., 2014; Falkner et al., 2014; Bell et al., 2014] and that the technology side of informatics is running, giving the impression that new content must always be chased. In this scenario, the tendency to focus on marginal and technological aspects is exacerbated by the lack of specific preparation of a large proportion of teachers asked to teach informatics without having studied it (especially in primary school).

Bell and colleagues are not the first to try to distil the big ideas of informatics education [see, e.g., Schwill, 1994; Denning and Martell, 2015]. However, they are the first to do so for young learners in pre-tertiary education, with a dissemination intent and narrative approach that are close to the spirit of the big ideas of science by Harlen et al. [2015], as they explicitly affirm.

> Computer science is full of paradoxes and surprises that provide opportunities for students to understand, enjoy and marvel at the digital world, and we are particularly interested in ensuring these are captured so that the ideas are in line

with the intention of the "Big ideas of science education". [Bell et al., 2018, sect. 1]

4.2.1 Features

The big ideas of informatics by Bell et al. [2018] are not intended as general principles, disciplinary areas or even curriculum topics but as powerful and general ideas that capture the essence of informatics. The big ideas are what young students should retain, regardless of whether they will specialize in informatics or use its core knowledge to exercise their right to citizenship; the focus is on concepts that are fundamental in informatics, but not obvious for the layman to recognize.

The big ideas of informatics are formulated so that teachers can use them with students to connect the various topics in the bigger picture of a long-standing discipline. To do so, they do not focus on specific technologies so that they can stay the same through technological advances. The following is an example that Bell himself gives about the big idea of programming versus its technological aspects, namely programming languages.

> Some might argue that students should use a language used in industry, such as Java or C#, but if we focus on the big ideas in programming, writing programs is about getting a computational device to follow the steps in an algorithm, and we know that any algorithm can be programmed using just six key elements: input, output, storage, sequenced instructions, iteration (loops) and conditionals (if and case statements). These six elements make a computing system *Turing Complete* [...] fully capable of computing anything that a conventional computer could compute [...]. If students learn about all six of these elements and how to combine them to achieve a desired result, then they have been exposed to the full power of computation [...]. [Bell and Duncan, 2018, sect. 10.7]

Indeed, the big ideas of informatics also meet Schwill's *vertical* and *horizontal applicability* [Schwill, 1994], which we have already mentioned regarding the big ideas of science. Also according to Bell et al. [2018], horizontally, a big idea must be stable over time and general enough to observe its manifestations in multiple phenomena of reality. Vertically, a big idea must make sense (at different levels of detail/abstraction) to students at various levels and throughout their careers.

4.2.2 Benefits

Knowing and understanding the major landmarks of informatics can help to see its bigger picture. In particular, the big ideas aim to inform curriculum design and, more importantly, to enable (particularly non-specialist) teachers to understand informatics in its fundamentals.

Indeed, without a general perspective, teachers may perceive informatics as an imposition on their limited classroom time; non-specialist teachers may feel overwhelmed by a series of extraneous and complicated topics. Focusing on the big ideas and goals of teaching informatics can prevent it from becoming a disjointed set of topics to be taught.

> For example, many primary school students learn a programming language such as Scratch, but what are the real concepts that we want them to take away from this? In ten years' time it is unlikely that they will need to know the colour of an arithmetic operator or the name of the command for storing a value into a variable. Likewise, many curricula include converting binary numbers to decimal, but in practice few people may ever have to do that. Yet there is value in working with both Scratch programming and binary numbers. Looking for the big picture [...] we need to appreciate the overarching goals to be able to make sense of the small details that appear in the classroom. [Bell and Duncan, 2018, sect. 10.1]

In addition, teachers can use big ideas to discern core knowledge from skills. "*For example, programming is just one of our ten big ideas, but in many curricula considerable time is spent on this, partly because it is a skill that can require some time to acquire. Conversely, there will be other topics around learning to use or configure computers that appear in curricula but not the big ideas because students need to learn to do these things, even though they aren't necessarily a fundamental concept from computer science.*" [Bell et al., 2018, sect. 1] Skills, like programming, are acquired over time, while core knowledge enables students to gain a broad understanding of informatics that is useful to support learning the proposed topics.

As for curriculum design, the big ideas are intended to help curriculum designers focus on core knowledge and topics so that teachers can recognize the long-term value of informatics in such curricula. Referring to stable, non-technology-dependent big ideas helps build curricula made up of relevant topics that will not soon be obsolete, rather than curricula that must be updated every time a new technology is developed. Also, a big picture helps to ensure that curricula are not built of not-so-relevant topics chosen just because their teaching resources are available in textbooks or online.

4.2.3 A collaborative process

The big ideas of informatics were developed in a collaborative process, pursuing as much objectivity as possible through inter-subjectivity. The process involved informatics education experts, curriculum designers, and informatics scientists worldwide.

Bell et al. [2018] developed the first list of candidate big ideas by having in-depth discussions with several (university and pretertiary) informatics education researchers. Other early versions were created by showing other fellow researchers the big ideas of science and asking them what an analogous list for informatics should include. The suggestions received and their first list were skimmed of repetitions and synthesized into a single list. The list was shared with teachers reasonably new to the subject to ensure that those big ideas were clear and meaningful to educators and curriculum designers.

Bell et al. [2018] report that the big ideas they propose result from several iterations of the process. They also caution against considering the list as definitive and open up to feedback and input from the entire community of researchers and educators.

Chapter 5

Teaching Informatics Concepts

From Lodi's perspective [Lodi et al., 2017; Lodi, 2020a; Lodi and Martini, 2021], the popularity of computational thinking and the push for its teaching in primary education suggest introducing the teaching of fundamental informatics concepts. We know informatics is difficult to access through programming (see 1). Therefore, it may be appropriate not to teach programming as a first thing, especially in the early grades, or at least to do so within a pathway that includes other topics and has cultural rather than specialistic goals.

Informatics fosters the development of other skills and competencies beyond programming. It is also a challenging academic subject with a rich conceptual framework, similar to physics or history. The core skill set that students acquire through studying informatics can be called computational thinking, and it encompasses concepts such as abstraction, decomposition, and generalization. It is crucial to comprehend them to be able to think computationally. [Curzon et al., 2018, sect. 8.1].

In the chapter titled *Teaching Computing in Primary Schools*, Bell and Duncan [2018, sect. 10.8] warn that we need to keep in mind that the main goal is not to push students with always new concepts but rather to provide an understanding of the subject and ignite students' passion for it. Above all, we need to be aware of why informatics is being included in a curriculum and keep the focus on the big picture of lasting big ideas, especially when teaching the details of a topic.

> For example, many primary school students learn a programming language such as Scratch, but what are the real concepts that we want them to take away from this? In ten years' time it is unlikely that they will need to know the colour of an arithmetic operator or the name of the command for storing a value into a variable. Likewise, many curricula include converting binary numbers to decimal, but in practice few people may ever have to do that. Yet there is value in working with both Scratch programming and binary numbers. Looking for the big picture [...] we need to appreciate the overarching goals to be able to make sense of the small details that appear in the classroom. This will raise ideas such as the concept of teaching computational thinking, and how the ideas can be made suitable for this age group. [Bell and Duncan, 2018, sect. 10.2]

In this scenario, big ideas are a framework that helps avoid new subjects at primary school coming across as seemingly unrelated collections of topics. Such a framework is helpful to be aware of what the core concepts are that we want students to take away from their learning. Some informatics big ideas, in a reinterpretation of Bell himself thought for the primary school, could be:

- programming involves combine a few key elements to create powerful applications;
- digital systems should be designed with the user in mind, and programs should be written with the next programmer in mind;
- there are many ways to represent data using two symbols, but the precision of the representation will be a trade-off between the physical cost of high accuracy against the human costs of insufficient accuracy, such as low quality images or a limited range of text characters that can't express their language properly;
- computation is an interaction between algorithms and data, and that this can be made to happen by writing programs.

> These are just some (important) examples of the general ideas that students can get to grips with as they engage in learning based around computational thinking. At the same time they will be gaining a familiarity with jargon (such as 'algorithm' and 'binary') by using it in practice. [Bell and Duncan, 2018, sect. 10.8]

The concepts and topics chosen to pursue these big ideas must be chosen with the broad perspective and goals of pretertiary, particularly primary education, always in mind.

> A key purpose of education is to prepare students for their future, and enable them to contribute positively to society. The particular programming language that they learn at six or eight years old is unlikely to be one that they use later in life (if they program at all in their careers) but the principles that they exercise by exploring computational thinking will be used later. [Bell and Duncan, 2018, sect. 10.8]

That of big ideas is a general approach to education, a "guidance system" in which big ideas can be seen as beacons guiding educators and students (and curriculum designers and even legislators) to safe destinations in the vast sea of disciplinary knowledge. However, despite its relevance and benefits, such a guidance system is not enough in daily teaching practice. Supporting students in understanding knowledge and acquiring skills and competencies requires deploying specific educational methodologies and strategies.

In the following section, we will review some of the most relevant methodologies for teaching informatics concepts from this cultural and citizenship perspective that points to big ideas rather than specialized knowledge.

The subsequent chapters report some of the most relevant educational literature about two main topics that have been the focuses of our research work, cryptography, and interdisciplinarity between informatics and mathematics (that we pursued through cryptography).

Indeed, we believe that these topics are emblematic examples of this cultural perspective of informatics for all. When approached within the premises just described, they are an excellent way to introduce informatics and ignite students' general interest in it while bringing valuable elements for interpreting today's society.

5.1 Approaches

As seen about introductory programming in the previous part (see 2.4), it is widely accepted that the more traditional forms of teaching (e.g., direct instruction) are no longer sufficient. The reasons include the increasingly complex environmental and societal challenges and the ever-expanding scientific and technological knowledge, particularly in the informatics domain. As for informatics education, the widespread difficulties of access to informatics (which pose problems of participation and citizenship, as seen in 1.1), show how our current teaching approaches are ineffective. Curzon et al. [2018, sect. 8.1] state that experiencing and understanding concepts like abstraction, decomposition, and generalization is crucial to learning informatics and developing computational thinking. It is necessary to use appropriate pedagogical techniques that give students a clear understanding of the various core concepts of informatics, their relationships, and how they fit into the subject's larger contexts.

5.1.1 Discovery Learning

Discovery learning is a method where students are given a question, problem, or set of observations to investigate, and then work independently to find the answers. Through this process, students "discover" the factual and conceptual knowledge they are seeking. It is an *inquiry-based* approach[1] where students have a high level of autonomy in their learning [Bruner, 1961].

This approach is thought to be effective as it allows students to build their own understanding of the material rather than simply memorizing information. Discovery learning can also help prepare students for real-world problem-solving and experimentation. It allows them to learn how to find and use resources, work collaboratively, and think critically and creatively. Additionally, discovery learning can help students develop the ability to learn on their own.

Baldwin [1996] argues that using discovery learning to teach informatics concepts can be effective because informatics is also a hands-on field involving problem solving, experimentation, and exploration. Since discovery learning can help students develop the ability to learn on their own, its use could be critical in informatics, where technology is always changing, and new skills need to be constantly learned.

However, the same author states that discovery learning in informatics education is rare. He claims that informatics courses can be systematically designed and taught for discovery learning, producing notable benefits over the conventional lecture-and-lab format. Discovery learning has been utilized in other disciplines, often in conjunction with group

[1] Inquiry learning is a general approach in which students are given questions, problems, or observations to investigate. Through this process, students are encouraged to find answers and solutions independently [Bateman, 1990].

learning [Davidson, 1990], and has also been applied in elementary and secondary science education, sometimes under the name "inquiry training" [Joyce et al., 1992]. However, according to Baldwin [1996], it is not commonly used in informatics education. At the same time, opportunities for discovery learning exist in many activities used to teach informatics (e.g., labs and projects), but they are often not fully exploited. Laboratories, especially those involving scientific experiments, are often seen as opportunities for students to discover and learn about informatics independently. However, these labs are not typically designed for discovery learning [Baldwin and Koomen, 1992; Moore, 1993]. Similarly, projects are often used in informatics education, and whole courses may be based on a single project or related set of projects. Although projects can be viewed as opportunities for students to discover new concepts, they are frequently used to reinforce information that was previously introduced through lectures or textbooks [Etlinger, 1990].

Based on two discovery learning courses, one a junior/senior course on computer graphics and the other a sophomore/junior introduction to C and UNIX, [Baldwin, 1996] describes some of those benefits and pitfalls. The traditional lecture-and-lab format assigns students a passive role in learning. Students attend classes, do assigned exercises and readings (but only the assigned ones), and take exams. Exams are an incentive for studying, but this "studying" is primarily memorization, done immediately before exams, and only of material expected to be on the exam. Students hardly ever identify or solve problems for themselves – they simply use the ideas in the text or lectures. They do not know what learning resources are available besides textbooks, and they try to read technical books like novels – they take read to mean "look at the words" rather than "understand the ideas". He claims that the worst problem with conventional instruction is that it does not prepare students for the learning they will need to do to keep up with their field after school [Baldwin, 1996]. On the other hand, discovery learning has its own set of pitfalls, such as students not learning the core knowledge, failure to make progress, students expecting to be told what to learn, and students becoming too focused on a specific project or discovery. To overcome these issues, instructors can design projects that require students to learn a common core of knowledge while still allowing for individual creativity, create mechanisms to keep students continually progressing, remind students of their role in the course and encourage self-discipline, and balance pure discovery learning with more traditional lectures or readings [Baldwin, 1996].

About the balance between exploration and creativity on one side and the necessary learnings, Prince and Felder [2006] state that the purest form of discovery learning, where teachers provide problems and give feedback on student's solutions but do not guide their efforts, is rarely used in higher education. The main reason is that instructors fear they will not be able to cover enough content if students are required to discover everything independently. To address this concern, evidence that discovery learning improves learning outcomes without sacrificing content is needed, yet, again according to Prince and Felder [2006], currently such evidence does not exist. Therefore, instructors are more likely to use a variation of discovery learning, also known as "guided discovery", which involves providing some level of guidance (the optimal guidance we discussed; see 2.5.3.6) during the learning process [Spencer and Jordan, 1999]. With this approach, the differences between discovery and problem-based learning (see 2.6.2 in part I) become less distinct [Prince and Felder, 2006].

5.1.2 Unplugged approach

Unplugged activities [Bell et al., 2009] to teach informatics have become increasingly popular in recent years. The unplugged approach is a form of teaching that focuses on teaching informatics concepts by engaging students in hands-on activities without computers (or other digital devices) by using physical objects to illustrate and experience abstract concepts. This approach is often used as an alternative to traditional computer-based instruction and activities. It engages students in the learning process, aiming to increase their understanding and retention of the content. According to Lodi [2020b, p. 115], unplugged activities offer:

- a constructivist environment: indeed
  - by manipulating real objects or dramatising processes, pupils can observe what happens, formulate hypotheses, validate them through experiments, i.e. develop a scientific approach to the construction of their knowledge;
  - by working in a group, pupils are encouraged to participate, share ideas, verbalize and uphold their deductions;
- inexpensive set up: they usually require very basic and inexpensive materials, so they can be easily proposed in different contexts;
- no technological hurdles: they allow students (and teachers) to have meaningful experiences related to important CS concepts (like algorithms) without having to wait until they get some technology and programming fluency.

There are a variety of approaches to do this centered around kinaesthetic approaches, such as role-playing computation, puzzles, art activities, games of all kinds, and magic tricks(see [Curzon et al., 2018, sect. 8.3] and Curzon and McOwan [2017]), to help develop computational thinking in its broadest sense. Stories can be used to explain informatics core concepts, such as decomposition, generalization, and abstraction, and unplugged activities can be used to illustrate these ideas by having students experience them. It is essential to make the links to the concepts being demonstrated clear; otherwise, students can be left understanding the concrete version of the activity but not the informatics concept itself [Curzon et al., 2018, sect. 8.3].

These activities have been successfully used at all levels [Curzon et al., 2019, sect. 17.3.1], from primary to master's levels, as well as when teaching adult teachers, as a way to teach programming and computing concepts more generally in a constructivist way [Papert and Harel, 1991]. Indeed, the unplugged approach to teaching requires minimal instructions and encourages students to construct knowledge for themselves; it then connects students' work to the context of informatics and physical computational devices. This approach has also been used as part of continuous professional development for teachers with positive evaluation results [Curzon, 2015; Smith et al., 2015; Meagher, 2017].

Generally, when using the unplugged approach, students are given a problem and then encouraged to explore potential algorithms for themselves. As the size of the problem increases, students must use more efficient and rigorous strategies. One of the objectives of using a constructivist approach is to break down stereotypes about what makes a good

informatician rather than teaching students specific algorithms and techniques. Students can learn that complex problems can also be solved in these unplugged contexts, and they can feel empowered when they discover informatics concepts on their own. Indeed, unplugged activities are also helpful for communicating to students, teachers, and education officials that there is depth to informatics and computational thinking that goes beyond computers and stereotypes.

The most popular unplugged activities come from the New Zealand-based *CS Unplugged* project[2] by Bell et al. [2009]. However, many "unplugged" activities are not necessarily based on CS Unplugged today; instead, they are developed by a large international community of educators. In any case, the key elements of all the activities and approaches we are exploring are that no computer is required – although all concepts come from informatics, that students are engaged in kinesthetic activities, and that the necessary equipment is readily available.

Tim Bell, the creator of CS Unplugged, has made it clear that his approach is not meant to replace the opportunity for students to write programs on digital devices but rather to serve as an adjunct pedagogy to enable learners to become aware of informatics big ideas [Bell et al., 2009] without the overhead of learning to program first [Curzon et al., 2014]. Indeed today, in the many classrooms where digital devices are available, the CS Unplugged activities can be explicitly linked to programming through a "plugging it in" follow-up [Bell and Vahrenhold, 2018].

Finally, it should come as no surprise that Tim Bell, informatics researcher and educator, is also the one who developed the big ideas of informatics (see 4.2). Indeed, his unplugged activities are a concrete way to experience and understand core informatics principles rather than learn specialized (and fragmented) knowledge.

5.1.2.1 Potential benefits and actual implementation

There are numerous potential benefits of using unplugged methodology and activities to teach informatics concepts since they provide "quick wins" that help build confidence in the topic. Also, teachers have reported inspiring confidence-building experiences without the overhead of having to learn to program first [Curzon et al., 2014]. According to CS Unplugged [[n.d.]], the main potential benefits are those listed below.

- Increased engagement. Unplugged activities can be engaging and hands-on, helping students to stay focused and motivated.

- Improved problem-solving skills. Unplugged activities help students develop their problem-solving skills by allowing them to explore and experiment with different approaches to solving a problem.

- Increased collaboration. Unplugged activities often involve small groups or the entire class working together to solve a problem, which encourages collaboration and communication.

[2] https://www.csunplugged.org/

- Improved creativity. Unplugged activities allow students to explore and use their creativity to come up with original solutions to problems.
- Improved conceptual understanding. Unplugged activities can help students develop a deeper understanding of concepts by letting them interact with the material in a more physical way.
- Improved computational literacy. Unplugged activities can help students develop their computational literacy by having them think through algorithmic problems without relying on technology.
- Increased accessibility. Unplugged activities provide students with the opportunity to learn informatics without requiring access to a computer.

As said, a key feature of the unplugged approach is using minimal instructions to enable students to build knowledge independently. After they have done this, it is crucial to connect their work to the larger context of informatics and what occurs on physical computational devices. Indeed, that later connection is indispensable. Without it, Feaster et al. [2011] found that "*the program* [based on CS Unplugged] *had no statistically significant impact on student attitudes toward computer science or perceived content understanding*". In the same vein, Taub et al. [2012] report that "*the students' attitudes and intentions regarding CS did not change in the desired direction*". Compared with more traditional approaches, Thies and Vahrenhold [2013] found that "*it is indeed possible to weave Computer Science Unplugged activities into lower secondary computer science classes without a negative effect on factual, procedural, or conceptual knowledge*", and that it could have some benefit in that "*the Computer Science Unplugged materials can prove helpful for ability grouping within a class, since, on average, more students are enabled to reach a higher operational stage*".

By examining these diverse contexts, we recognize that CS Unplugged performs best when combined with later "plugged-in" activities. This should not come as a surprise because getting a program to run correctly is an excellent way for students to demonstrate that they have grasped the informatics concepts they are working with since the computational agent (i.e., the computer running the program) will do exactly what the program says to do. Additionally, this will enable students to experience the results of their instructions in a tangible environment that gives immediate and ordinary feedback (rather than waiting for human and potentially vague feedback from the teacher). Summing up, to actually express the potential benefits above, the unplugged approach must be followed by plugged work, linking the tangible experience of informatics concepts with their computational manifestation. That is why the "original" CS Unplugged activities by Bell et al. [2009] are today provided with their related "plugging-in" follow-up work [Bell and Vahrenhold, 2018]. The current version of CS Unplugged also provides guidance on scaffolding [Wood et al., 1976] and Socratic-style questioning to ensure that students explore the ideas and construct their own knowledge [Wells, 1999].

5.1.3 Task-specific programming languages

Guzdial and Naimipour [2019] proposes task-specific programming languages as a possible

solution to the fact that few students choose informatics in the U.S. [Guzdial, 2019a], which is a problem both from a cultural and citizenship perspective and because of the continuing shortage of informaticians (as discussed in 1.1). The vision behind task-specific programming languages is that for most people, the best way to access informatics is not the classical way, i.e., through learning programming (the difficulty of accessing informatics through programming and related analyses and strategies are discussed in the part I of the literature review), but rather by engaging more and more students in gaining computational literacy.

> Computing education research has supported the hypothesis that context can make computer science learning more successful. We see a historical argument for using other STEM subjects as that context [Knuth, 1972; Perlis, 1962]. The goal for integration is to make a positive feedback loop. Consider physics, as an example. Learning computer science in the context of physics can lead to better learning of physics (e.g., as seen in Guzdial [2019a] and Sherin [2001]) and can also make the computer science more relevant and useful [Guzdial, 2013]. [Guzdial and Naimipour, 2019, p. 1]

As anticipated at the end of 2.7, task-specific programming languages, initially introduced by Guzdial et al. [1997], have two main goals within this vision. The first is to support the learning of non-programming tasks that teachers (typically non-informatics) want students to accomplish by leveraging computers for their automaticity and computational power and because they can provide meaningful real-world scenarios. This vision embodies one of the interdisciplinary souls of informatics[3], already theorized and discussed by Papert [see, e.g. Papert, 1980].

Second, they aim to embody computational thinking "à la Papert" (see 3.1). Students using task-specific programming languages are exposed to the fundamental ideas of informatics while pursuing other learning goals.

> Even before students learn ideas like variables and iteration, there are more fundamental ideas of computing that students need to understand. These include the ideas that programs are assembled out of basic elements, and different orderings of elements can sometimes have the same result, and even that the program determines the computer's behavior (there's no magic). [Yadav and Berthelsen, 2021, p. 174]

Thus, using task-specific programming languages alone does not allow one to learn how to program; however, it does allow students to experience in concrete and active contexts (thus potentially constructivist; see 2.5.3) some of the concepts and ways of thinking of informatics.

Following the initial definition by Guzdial et al. [1997], Task-specific programming languages are designed to be used for specific tasks and typically have a limited set of features and capabilities. They are optimized for the task at hand and may lack features that are not required for that task. They may also have a simplified syntax, making them easier to use for non-programmers. According to Guzdial [2021], task-specific programming languages:

[3] Interdisciplinarity, particularly between informatics and mathematics, is discussed extensively in the remainder of this part II, particularly in 7

- Support learning tasks that teachers (typically non-CS teachers) want students to achieve;
- Are programming languages, in that they specify computational processes for a computational agent to execute; and
- Are learnable in less than 10 minutes, so that they can be learned and used in a one hour lesson. If the language is never used again, it wasn't a significant learning cost and still provided the benefit of a computational lesson.

In contrast to the classification of languages proposed by Hermans [2020] (and reported in 2.7), Guzial believes that a language such as LOGO, although it fits within the languages designed for learning to program, is not a "mini" language at all (nor "toy" according to a previous version of the same classifications [Gilsing and Hermans, 2021]), since it is a general purpose and Turing-complete programming language, even though it was conceived (long ago now) with a simplified syntax for educational purposes. According to Guzdial and Naimipour [2019], task-specific programming languages, on the other hand, genuinely are mini-languages. In them, the task is very specific, circumscribed, and generally unrelated to learning programming. In addition, they argue that task-specific programming languages are more akin to Papert's *microworlds* rather than the full LOGO. For example, the original LOGO module *Turtle Graphics* is LOGO's microworld for plane geometry. If students use only the constructs of the Turtle module within LOGO, they are indeed using a mini-language or a task-specific programming language. Just as is the case with task-specific languages, the goal of microworlds is not to teach programming but to teach something else (i.e., the task at hand), albeit through the programming of a computational agent.

These are the premises that lead Guzdial and Naimipour [2019] to explore task-specific programming within precalculus, a U.S. teaching subject chosen because of its possible applications in multiple domains, thus critical for success in STEM studies [Sadler and Sonnert, 2018], and because they claim it has the potential to be taught more effectively through the use of computing. In this case, the task-specific programming language takes the form of application software for image editing, specifically "*[a]n image filter builder for learning basic matrix arithmetic (addition and subtraction) and matrix multiplication by a scalar*" [Guzdial and Naimipour, 2019, p. 1].

Again according to Guzdial and Naimipour [2019], task-specific programming languages offer a promising possibility for integrating informatics into other disciplines, which is impossible with our general-purpose programming languages, even graphical block-based ones. They suggest that by using task-specific programming languages, students could learn the causal and repeatable nature of informatics programs. So the authors raise the research question of how much students could learn about informatics if they used more task-specific programming languages before they started their first informatics-specific course. In other words, they invite exploration of how much informatics is really needed to help students learn with and about informatics rather than just assuming that informatics is taught only through learning programming with a Turing-complete language.

5.1.3.1 Teaspoon languages

More recently, Guzial (in Guzdial [2021] and, more lengthy, in Yadav and Berthelsen [2021]) pushes the vision of task-specific programming languages even further beyond STEM disciplines (naturally akin to informatics) to propose small programming activities in most school courses, even humanities one. The main characteristics and goals of teaspoon languages do not differ substantially from those of task-specific programming languages, already described above. Teaspoon languages aim to be even simpler and more limited and support learning also in non-STEM domains. Indeed, Guzdial [2021] suggests that the relationship between the two is "$TSP >= Teaspoon$". Actually, the term *teaspoon* is meant to emphasize the small amount of computing brought into a non-informatics activity.

The vision for teaspoon languages of Guzdial [2021] and Yadav and Berthelsen [2021] contrasts with the "Hour of Code"[4] system, whose core idea is to increase familiarity and confidence with programming by exposing students to real informatics every year. On the other hand, the teaspoon language approach suggests including few and small programming activities in every social study, language, art, and math class every year. These teaspoon languages are all distinct and tiny. The objective is for students to feel comfortable with programming by the time they take an informatics course (typically in high school or university) by having had much experience with it and having seen a variety of programming languages.

[4] https://hourofcode.com/

Chapter 6

Cryptography

6.1 Importance of cryptography today

Cryptography is essential to many activities and tools in our contemporary digital and connected society. Instant messaging and social networking, online purchases of goods and services, online trading, digital identity and e-governance are just some of the current services and applications that are only possible because of modern cryptography. Various European frameworks (such as DigComp [European Commission, 2017]) and international curricula (such as the American education standards [K-12 CS Framework, 2016] and the English informatics curriculum [UK Department of Education, 2013]) include skills related to informatics security (i.e., cybersecurity). Some of these skills are more oriented toward using informatics security for specific purposes (personal or business), while others are about understanding its foundational principles. In any case, these documents recognize that skills related to informatics security are essential for students to become active citizens of the digital society.

Cryptography is one of the pillars of informatics security. Moreover, novice students recognize cryptography as an appealing context for informatics classes [Lindmeier and Mühling, 2020, p. 3]. Pre-college education is not intended to train professionals but to help students understand the world we live in so that they can take an active part in it, pursuing their own goals. Therefore, we believe that students must know and understand the principles of cryptography and their relevance to the activities and tools of digital society.

6.2 Cryptography education

6.2.1 International frameworks

In 2017, the leading U.S. and international associations of informaticians and informatics engineers (including ACM, *Association for Computing Machinery*, and *IEEE Computer Society* develop the *Cybersecurity Curricula 2017* [Joint Task Force on Cybersecurity Education, 2018], which gathers guidelines for higher-education courses in cybersecurity. The document, clearly aimed at specialized education, includes cryptography in the first learning unit, deemed

necessary to lay the foundation for subsequent learning [Joint Task Force on Cybersecurity Education, 2018, p. 24].

Among the cryptography content included in the document, we report:

- essential concepts such as, for example, encryption and decryption functions, authentication, symmetric vs. asymmetric key, perfect security (e.g., *One-time pad*), computational security, and more;
- mathematical background such as, for example, modular arithmetic, primitive roots and discrete logarithm, primality test vs. factorization of large integers, and more;
- historical ciphers such as, for example, transposition ciphers, substitution ciphers (e.g., Vigenère, ROT-13), Enigma machine, and more;
- symmetric ciphers such as, for example, block ciphers including DES, AES, and more;
- asymmetric ciphers and concepts such as computational complexity, *one-way* functions, Diffie-Hellman protocol, RSA's notion, and more.

Concerning specifically pre-college (also K-12) education, the American Computer Science Teachers Association (CSTA), and some of the associations mentioned above, include cybersecurity as an essential topic at all school levels. Specifically, in 'Level 2,' the one related to students between the ages of 11 and 14 (grades 6-8), the 2-NI-06 standard indicates that students should:

> [a]pply multiple methods of encryption to model the secure transmission of information. [...] Encryption can be as simple as letter substitution or as complicated as modern methods used to secure networks and the Internet. Students should encode and decode messages using a variety of encryption methods, and they should understand the different levels of complexity used to hide or secure information. [Students could use] methods such as Caesar ciphers [...]. They can also model more complicated methods, such as public key encryption, through unplugged activities [CSTA, 2017, 2-NI-06]

6.2.2 Cryptography education in IEdR conferences

In a review of publications of cybersecurity education in the two largest ACM conferences (i.e., ITiCSE and SIGCSE) between 2010 and 2019, Švábenský et al. [2020] find that the research "*predominantly focus[es] on tertiary education in the USA*". Most of the reviewed papers deal with the broader perspective of cybersecurity. Cryptography is just one of many topics considered [see, e.g., Sommers, 2010; Turner et al., 2011; Brown et al., 2012; Deshpande et al., 2019], and it is often seen as a technical and instrumental topic rather than being treated from the perspective of its founding principles. In addition, very few papers [e.g., Buchele, 2013; Hsin, 2005] specifically address cryptography: only 14 of the 71 papers reviewed include cryptography topics.

Given the importance and novelty of the topic in pre-university education, we decided to conduct ourselves an extensive review of papers dealing with the teaching of cryptography

in the school context. We considered both works focusing only on cryptography and works dealing more extensively with cybersecurity only when cryptography was included in a relevant way.

In the following sections, we report the most significant trends that emerged from our review.

6.2.3 Hands-on and inquiry-based activities

The educational proposals we found often include hands-on activities, most times situated in motivating and realistic contexts; some relevant examples follow.

Simulations of encryption, decryption and attacks on the Caesar and Vigenère ciphers and the RSA cryptosystem to understand how a secure email exchange works [Gramm et al., 2012]. Summer camps to teach cybersecurity and computational thinking through robotics and block-based programming languages, with Caesar and Vigènere encryptions implemented in robots for secure communications [Lédeczi et al., 2019; Yett et al., 2020]. A "toy" social networking platform that includes simple activities and tools for implementing the Caesar cipher with the goal of learning essential cryptography and informatics security concepts in the process [Zinkus et al., 2019]. Purpose-built interactive web tools that simulate cryptography systems at a high level and thus enable students to engage in cryptographic challenges [Schweitzer and Boleng, 2009].

In general, in the context of cryptography and cybersecurity, Konak [2018] argues that "experiential learning" activities (i.e., either active and participatory, hands-on, or inquiry-based) can improve students' self-efficacy and problem-solving skills.

6.2.4 Visualization tools and high-level programming

Various visualization and interactive simulation tools have been developed over time to teach cryptography algorithms and systems, such as Caesar and Vigenère ciphers, DES, AES, RSA, and SHA. Usually, these simulations show how ciphers work, their weaknesses, and thus how they can be attacked [see, e.g., Simms and Chi, 2011; Schweitzer and Brown, 2009; Ma et al., 2016; Anane and Alshammari, 2020]. Often these simulations are accurate but too rich in technical details for high school students and, in general, for young learners who are novices in cryptography. Moreover, interactivity is often limited to entering the message to be encrypted, proceeding step-by-step, and receiving brief explanations. At most, it is possible to change a few simulation parameters.

A less technical and more student-oriented approach is that proposed by McAndrew [2008]. Students can use open-source computer algebra systems to implement classic and modern cryptography algorithms so that, by programming them at a high level, they can better understand how they work and how they can be attacked.

The idea of high-level programming also emerges outside the educational context. van der Linden et al. [2018] suggest using the metaphor of blocks, taken from visual programming languages, to represent cryptographic functionality. This way, the cognitive load is reduced, and possible errors are limited; as a result, software developers can implement cryptographic features in systems more securely and effectively.

6.2.5 Unplugged activities

Some authors propose and implement *unplugged* activities to teach cryptography to smooth its learning curve and make it educational and also enjoyable for young students. In these activities, learners experience encryption and decryption algorithms, protocols and attacks at a high level, with hands-on activities and without using computers (a description of the unplugged approach can be found in 5.1.2).

For example, Bell et al. [2003][1] propose simulating a *one-way function* (a fundamental concept in public-key cryptography) by searching for a *perfect dominating set* (PDS)[2] on a graph. Young learners perform elementary arithmetic calculations on the graph nodes to exchange a secret number.

Konak [2014] proposes some simple unplugged activities to explain classical and modern cryptosystems with paper, pen, boxes and locks to K-12 students. For example, learners use paper strips to create variations of Caesar cipher and a paper with four holes cut out as a grille cipher. Boxes and locks are used "*to introduce the concept of key exchange and discuss the problems of using symmetric algorithms over a public network*".

Fees et al. [2018] make concrete the color metaphor often used to explain the paramount *Diffie-Hellman key agreement* by having students mix food colorants to generate a shared secret key.

> Alice and Bob are often regarded as two people with their own private paint colors (or private keys) with access to a public paint color (public key). The two exchange paint in such a way that the final result is Alice creating a paint color that is the same color as Bobs[...] The exchange was public, so an eavesdropper, often known as Eve, has access to much of the information but what she is missing is the private colors possessed by Alice and Bob. Therefore, Eve cannot arrive at the same color (decrypt the message)[...] [Fees et al., 2018, sect. 4.2]

Greenlaw et al. [2015] adopt an unplugged approach albeit mediated by the use of an informatics tool, which in this case is not part of the learnings (as, for example, an IDE is when learning to program) but only a virtual context (as an alternative to a concrete one) where the activity is happening. In an introductory cybersecurity course, they use the virtual message board as a teaching tool to demonstrate how a *person-in-the-middle attack* on a public key cryptosystem works.

[1] Tim Bell was one of the first to propose and implement *unplugged* activities to teach core informatics concepts.

[2] A vertex v of an undirected graph dominates vertex u if there is an edge from v to u. The vertex v also dominates itself. A set of vertices is a *perfect dominating set* if each vertex of the graph is dominated by exactly one vertex in the set.

Chapter 7

Interdisciplinarity and Non-Informatics Methodologies

The importance of introducing informatics into pre-university education has been strongly advocated over the past decade. As stated on many occasions in this thesis, informatics should be recognized as a fundamental and independent scientific discipline. It should be taught to students so they can understand the digital world in which we are immersed, act as active and informed citizens, and become potential workers in the ever-growing digital job market.

However, in an increasingly complex and rapidly changing world, many criticize the traditional compartmentalized teaching of disciplines in schools. More integrated and interdisciplinary teaching is advocated, particularly for the core STEM fields (science, technology, engineering and mathematics).

A review of interdisciplinarity follows, focusing on its impact and use in education and framed in the context of a European Erasmus+ project in which we participate. Then we review two methodologies from mathematics education that we used – following an interdisciplinarity principle – to design an interdisciplinary educational intervention (between mathematics and informatics) on cryptography.

7.1 The context of the IDENTITIES project

IDENTITIES is an Erasmus+ project started in 2019 and coordinated by the University of Bologna. Universities from four different countries are partners in the project: the University of Bologna and Parma (Italy), the University of Barcelona (Spain), the University of Crete (Greece), and the University of Montpellier (France). The researchers involved are experts in various fields, from informatics education to linguistics, from physics education to mathematics education. IDENTITIES focuses not on student education but on in-service and pre-service teacher education. Interdisciplinarity in the STEM (science, technology, engineering, mathematics) domain is the core of the project. The project addresses STEM topics (such as climate change, nanotechnology and coronavirus) and interdisciplinary curricular topics (such as cryptography, parabola and parabolic motion). IDENTITIES is still ongoing. The project

is developing new teaching approaches to interdisciplinarity in science to innovate teacher education for complex contemporary challenges (e.g., climate change). It is also exploring some of the STEM topics emerging from these challenges and curricular interdisciplinary topics (e.g., cryptography) with a twofold purpose. These topics are both contexts for investigating new forms of knowledge organization for interdisciplinarity and an opportunity for the interdisciplinary design of classroom activities and novel models of co-teaching.

7.2 The necessity of interdisciplinarity

The importance of interdisciplinary knowledge is a central focus of many educational and institutional research projects today [OECD, 2019][1]. Today's emergencies (such as the pandemic and climate change), which we face as a society, increasingly highlight that a monodisciplinary approach is no longer adequate, just as traditional monodisciplinary education is not [Baptista and Klein, 2022; Pharo et al., 2012; Brown et al., 2010; Schmitz et al., 2010].

Although interdisciplinarity is nowadays widely promulgated as a necessary and positive goal in the academic comunity [Brown, 2020; MacLeod, 2016; Grüne-Yanoff, 2016; Khilji, 2014; Frodeman, 2014; Huutoniemi et al., 2010], disciplines are organized into subjects at school and university [Ortiz-Revilla et al., 2020]. In other words, disciplines are the key unit of knowledge and social organization in education [Jacobs, 2014]. Indeed, the current disciplines define and shape research, other kinds of publications, careers and professional associations. In this scenario, interdisciplinarity challenges the very structure of school and academia [Jacobs, 2014]. While the current knowledge organization has been (and still is) motivated by the need to develop competence in the respective disciplinary fields, it can create boundaries and barriers [e.g., Russell, 2022; Pharo et al., 2012]. Such barriers hinder collaboration, awareness of the similarities and differences between disciplines, and the development of shared languages needed to understand and collaborate.

There is a need for a third way to maintain the advantages of organizing knowledge and teaching into disciplines while profitably synergizing the different disciplinary viewpoints [Baptista and Klein, 2022; Barelli et al., 2022]. According to the 'Education at a Glance' report [OECD, 2019], *interdisciplinary knowledge* is as important and valuable as *disciplinary knowledge*. The IDENTITIES project aims to overcome two forms of banalization about these two types of knowledge. The first sees interdisciplinarity as a-disciplinarity or non-disciplinarity as if knowledge and education could be based only on transversal themes. The second narrows interdisciplinarity to the instrumental use of concepts and methods from one discipline (e.g., informatics) to solve problems formulated in another (e.g., mathematics).

7.3 Defining interdisciplinarity

Much research has attempted to make a definition of interdisciplinarity. According to Brewer [1999], "*[i]nterdisciplinarity generally refers to the appropriate combination of knowledge*

[1] *Education at a Glance* is the authoritative and most comprehensive source of information on the state of education worldwide; it provides data on the education systems of OECD (*Organization for Economic Co-operation and Development*) countries and some partner nations.

from many different specialities – especially as a means to shed new light on an actual problem" [Brewer, 1999, p. 328].

Choi and Pak [2007], more precisely, make the following proposal, which is now widely used and accepted.

> - *Multidisciplinarity* draws on knowledge from different disciplines but stays within the boundaries of those fields.
> - *Interdisciplinarity* analyzes, synthesizes, and harmonizes links between disciplines into a coordinated and coherent whole.
> - *Transdisciplinarity* integrates the natural, social and health sciences in a humanities context, and in doing so transcends each of their traditional boundaries.
>
> [Choi and Pak, 2007, p. 359]

Klein [2010], too, does not provide a single definition but a taxonomy of interdisciplinarity, similar to Choi and Pak [2007]. In short, multidisciplinarity (which Klein calls *pseudo-interdisciplinarity*) involves encyclopedic and additive juxtaposition and, at best, some form of coordination. Pseudo-interdisciplinarity lacks intercommunication, and disciplines stay separate. In contrast, interdisciplinarity is integration, interaction, and connection.

7.3.1 The boundaries perspective

Akkerman and Bakker [2011] give interdisciplinarity a new perspective with the influential work 'Boundary crossing and boundary objects'. According to them, *boundaries* exist and should be considered in both learning and work.

> Whether we speak of learning as the change from novice to expert in a particular domain or as the development from legitimate peripheral participation to being a full member of a particular community [...], the boundary of the domain or community is constitutive of what counts as expertise or as central participation. When we consider learning in terms of identity development, a key question is the distinction between what is part of me versus what is not (yet) part of me. Boundaries are becoming more explicit because of increasing specialization; people, therefore, search for ways to connect and mobilize themselves across social and cultural practices to avoid fragmentation. [Akkerman and Bakker, 2011, p. 132]

Thus, the (educational and professional) challenge is creating opportunities for collaboration across boundaries between a variety of different fields.

Based on a body of research investigating the concept and nature of *boundary* [e.g., Suchman, 1993; Engeström et al., 1995], they define it as "*sociocultural difference leading to a discontinuity in action or interaction. Boundaries simultaneously suggest a sameness and continuity in the sense that within discontinuity two or more sites are relevant to one another in a particular way.*" [Akkerman and Bakker, 2011, p. 133]

Their theory introduces three other concepts built around the boundary one. The first is that of *boundary objects*. Boundary objects "*enact the boundary by addressing and articulating meanings and perspectives of various intersecting worlds or that move beyond the boundary in that they have an unspecified quality of their own*" [Akkerman and Bakker, 2011, p. 150]. They have an ambiguous nature, as they belong "*to both one world and another*", and "*neither one nor the other world*" simultaneously. Thus, boundary objects have the potential to divide two worlds as much as to connect them.

> We contend that it is precisely this ambiguous nature that explains the interest in boundaries and boundary crossing as phenomena of investigation for education scholars. Both the enactment of multivoicedness (both–and) and the unspecified quality (neither–nor) of boundaries create a need for dialogue, in which meanings have to be negotiated and from which something new may emerge. [Akkerman and Bakker, 2011, p. 142]

The second concept is *boundary people*, "*marginal strangers who sort of belong and sort of don't*" [Akkerman and Bakker, 2011, p. 150]. They are those "bridge" people who make the difficult (yet necessary) experience of alterity in dealing with two or more communities whose identities are strongly defined and "defended" by their respective members. As boundary people deal with these communities from the outside and the inside, their identity is determined by a continuous negotiation of sense and meanings. This negotiation process allows them to develop new knowledge and a new language, enabling them to talk about and to all the communities.

The third concept is *boundary-crossing learning mechanisms*. Boundary crossing is a process in which demarcation lines between disciplines or practices are uncertain or destabilized because of increasing similarities or overlaps. Suchman first defines boundary crossing regarding professionals who "*enter into territory in which we are unfamiliar and, to some significant extent therefore unqualified*" [Suchman, 1993, p. 25]. In doing so, they "*face the challenge of negotiating and combining ingredients from different contexts to achieve hybrid situations*" [Engeström et al., 1995, p. 319]. This challenge is complex and achieving its goal (i.e., making such "hybrid situations" happen) requires those who face the challenge to put specific boundary-crossing learning mechanisms in place [Akkerman and Bakker, 2011].

7.3.2 Learning through boundary crossing

According to Akkerman and Bakker [2011], such learning mechanisms are four and ideally describe a progression toward the full realization of interdisciplinarity: *identification*, *coordination*, *reflection*, and *transformation*.

Identification entails questioning the core identity of each intersecting discipline or practice; questioning leads to a new understanding of what the different practices are about. The boundaries between disciplines or practices are encountered and reconstructed without necessarily overcoming discontinuities. The learning potential resides in a renewed sense-making of different disciplines or practices and related identities.

Learning at the boundary is a matter of *coordination*, where boundary objects have the role of mediating artifacts. Effective means and procedures could allow diverse disciplines or

practices to cooperate efficiently; a dialogue between the different parts is established only to maintain the flow. Coordination is very different from identification. Learning potential is not in reconstructing but in overcoming the boundary: continuity is established, facilitating future and effortless movements between different disciplines or practices.

Reflection occurs by realizing and making explicit the differences between disciplines or practices and learning something new about oneself and others. It involves both perspective-making and perspective-taking, which are dialogical and creative processes and generate something new. Reflection looks similar to identification, yet it is different in focus. While identification produces a renewed sense of disciplines or practices and an identity reconstruction, reflection results in an expanded set of perspectives and, thus, in constructing a new identity that will inform future practice.

Transformation constitutes a further step. Collaboration and co-development lead to profound changes in disciplines or practices, potentially creating new, in-between disciplines or practices. The new boundary disciplines or practices are meaningful in both worlds and are evolutions of the original ones from which they emerged. Transformation is the most difficult to achieve. Most of the research aims for this fourth type of dialogical learning mechanism.

7.4 Interdisciplinarity in education

Kapon and Erduran [2021] use the boundaries framework to show three cases in different education contexts – each exhibiting one or more learning mechanisms at work – in which boundary crossing promotes interdisciplinary learning.

Schvartzer et al. [2019] discuss student engagement when learning by using disciplinary knowledge in different interdisciplinary contexts and problems.

Levy et al. [2019] focus on how discipline-based problems can be better understood using the explanatory potential of interdisciplinarity.

Levrini et al. [2019] emphasized the inherent interdisciplinarity of STEM disciplines and topics (including cryptography) in discipline-based educational systems from a curriculum development perspective.

7.4.1 Interidiscipinarity and disciplines

This renewed sensitivity to interdisciplinarity in education highlights its nature as a process of "*integrating, interacting, linking, and focusing*" [Klein, 2010] different disciplines, that is, disciplinary domains and their epistemic cores. On the subject of disciplines, Alvargonzález describes how interdisciplinarity can change them.

> Interdisciplinarity would arise in a near symmetrical way when two or more disciplines converge in a given field, as they would, for example, in biochemistry, bioinformatics or geophysics. This convergence can lead to practical and theoretical integration of the disciplines involved, which would be unified. Paradoxically, these convergences, on many occasions, give rise to newly independent and sovereign disciplines, at least when they are considered in terms of their academic institutionalization. [Alvargonzález, 2011, pp. 392,393]

Interactions between disciplines "*may range from simple communication of ideas to the mutual integration of organizing concepts, methodology, procedures, epistemology, terminology, data, and organization of research and education in a fairly large field*" [Apostel et al., 1972, p. 25]. Therefore, it is also a matter of the epistemology of disciplines.

IDENTITIES project reflects on the epistemic nature of interdisciplinarity from an educational perspective. It aims to answer how to discuss what we mean by discipline non-stereotypically and how to decode (and possibly deconstruct) the notion of interdisciplinarity. One of the project's most ambitious goals is to interpret and articulate the complex intertwining between disciplinarity and interdisciplinarity in STEM. The etymology of the word *discipline* is based on the Latin verb *discere*, which means 'to learn'. Disciplines are a set of knowledge and skills rooted in the educational need to (re)organize knowledge to teach, learn and communicate (Alvarogonzález, 2011). The (re)organization must be so that learners, while constructing their knowledge, can also develop epistemic skills, such as explanation, sharing, argumentation, problem solving, verification, representation, and modelling [Levrini et al., 2019].

The scope of the IDENTITIES project goes beyond the analysis of interdisciplinarity from a theoretical and epistemic perspective. However, the description of the project stops here, because this is where our active involvement in it stops. Likewise, the review of the literature on interdisciplinarity also ends here, because that is enough to frame and interpret our research on interdisciplinarity in cryptography.

7.5 Theory of Didactical Situations and Didactical Engineering

The *Theory of Didactical Situations* (TDS) is a theoretical framework that focuses on understanding the learning and teaching process in the classroom. It provides a framework for analyzing the didactical situation, which includes the teacher, the students, the curriculum, and the learning environment. TDS aims to identify the obstacles that students face in learning and provides solutions to overcome them.

Didactical Engineering (DE), on the other hand, is a methodology that uses the principles of TDS to design teaching sequences and learning environments. DE aims to create optimal learning situations by designing teaching sequences that are coherent, dynamic, and effective.

The relationship between TDS and DE can be seen as follows: TDS provides the theoretical foundation for DE, and DE applies the principles of TDS to design teaching sequences and learning environments. The principles of TDS guide the design process in DE, ensuring that the teaching sequences effectively overcome the obstacles students face in the learning process. In other words, DE is a practical application of TDS.

7.5.1 Theory of Didactical Situations

The *Theory of Didactical Situations* [Brousseau and Balacheff, 1997], first developed in France by Brousseau between 1970 and 1990, has become prominent in mathematics education in French-speaking countries. This theory is used as an educational methodology; it can be employed to design a learning situation (e.g., a task or problem) within the DE framework

(acting as a research methodology to investigate the effectiveness of the whole educational proposal).

TDS is based on concepts such as *didactical situation*, *learning obstacle*, and *didactical contract*. According to this theory, the knowledge development process in students includes several consecutive steps (i.e., different types of situations), for example, *action*, *communication*, *validation*, and *institutionalization*. Some of constitutive concepts of TDS are particularly relevant to our research on interdisciplinarity in cryptography; we briefly describe them.

- A *didactical variable* is a variable of the situation's task or problem that is parameterized by teachers (or researchers). Its possible values influence the potential (hierarchy of) strategies students implement to tackle the task or solve the problem. Identifying instructional variables, foreseeing their effects, and choosing their values according to learning objectives is crucial to any *a priori* analysis of a didactical situation.

- In the context of the Theory of Didactical Situations, the *milieu* is the set of elements of a situation with which students can interact. In response to students' actions, the milieu produces retroactions (i.e., feedback), enabling students to adjust their behavior, modify their understanding of the task or problem and adapt, ultimately enabling them to learn. The analysis and organization of a milieu – mainly in terms of possible students' actions and retroactions that the milieu allows – is crucial for the design of a didactical situation and for its *a priori* analysis.

- *Adidacticity* in a didactical situation is the potential of the situation and its milieu to enable students' learning independently of teacher interventions (as in the case of learning outside didactical contexts). In TDS, adidacticity in learning is crucial, mainly because it prevents specific side effects of the *didactical contract* (i.e., the implicit pact between students and teachers [Brousseau et al., 2020]).

- To realize the learning potential of a didactical situation, teachers must transfer to students some responsibility for completing the task or solving the problem. *Devolution* enables this transfer of responsibility [Brousseau and Warfield, 2020] and must be considered when designing a didactical situation. Specularly, after students have completed the task or solved the problem, *institutionalization* is the teachers' action that allows students to structure what they have learned (by facing the task or problem) into more formal knowledge.

7.5.2 Didactical Engineering

Didactical Engineering (or *Didactic Engineering*) is an educational research methodology that involves designing and analyzing (from an epistemological and an educational perspective) a didactical situation that is then tested [Artigue, 1994; 2014; Barquero and Bosch, 2015; Artigue, 2020]. According to Barquero and Bosch [2015], Didactical Engineering "*emerged in the middle of the 1970s with the works of the French researcher Guy Brousseau [Brousseau and Balacheff, 1997] [...] to define the relationships between the theoretical developments*

of didactics and the empirical reality of the classrooms". It has been successfully practiced for carrying out *qualitative research* in mathematics education for several decades [Artigue, 2020].

This methodology focuses on the conception, design, organization, realization, observation, and analysis of classroom implementations. It was developed to address theoretical and practical aspects of mathematics education that existing methodologies (e.g., questionnaires, interviews, and test comparisons) – adapted from other scientific fields, such as psychology – could not capture.

DE aims to build a constructive relationship between practice and research. Educational systems are indeed considered in their concrete operation, and researchers must always take into account the conditions and constraints under which teaching and learning processes occur.

DE has also been practiced outside the mathematics education community (e.g., in physics education) and has been used to train teachers and experiment with new educational approaches, methodologies, and designs.

The Didactical Engineering methodology is a process structured into four intertwined phases.

1. *Preliminary analysis.* This first phase clarifies the background for the next phase, that is, for the conception and organization of the didactical situation. It consists of studying the content (originally, mathematical content) and the conditions of teaching and learning processes. The analysis is carried out in three different dimensions.

   - An institutional analysis of the actual context in which the didactical situation will take place. Researchers identify and analyze the conditions and constraints of the specific educational context. Institutional conditions and constraints may be of different kinds: curricular characteristics, teaching practices, technological resources available, evaluation practices, characteristics of the students and teachers involved, and more.
   - An epistemological analysis of the content. Researchers identify possible epistemological issues related to the content to be taught. Epistemological analysis helps researchers take the necessary reflective stance and distance from the educational context in which they are embedded and thus build a reference.
   - A didactical analysis of the content. Researchers survey the existing educational literature about the teaching and learning of the content to be taught.

   According to Artigue [2014], these three dimensions of the preliminary analysis reflect the systemic perspective underlying DE as a research methodology. Each phase has its methodological specificities and needs. The epistemological analysis often involves using historical sources; the institutional analysis also generally includes a historical dimension. As made clear by Bosch et al. [2005], curricular organizations and choices result from a long-term historical process; they cannot be understood just by analyzing current curricula, official documents, and textbooks. Such understanding is needed to clarify the strength of the constraints faced and how some of these can be moved in

the design. The didactical analysis generally has a substantial cognitive dimension, but this cognitive dimension is only one part of the global picture. It should also be noted that, depending on the specific objectives of each research, what is investigated in each of these dimensions (and its respective importance) can vary substantially.

2. *Conception and a priori analysis.* This second stage involves modeling the didactical situation and analyzing its content and organization within an educational theoretical framework (for example, the *Theory of Didactical Situations*, discussed in the following 7.5.1, or the *Anthropological Theory of Didactics* [Chevallard and Bosch, 2020]). The conception of the situation and analysis are closely related, as analysis helps to revise and adapt the conception in order to achieve the targeted learning objectives.

   The conception concerns a series of choices that affect both the content to be taught and the organization of the didactical situation at different levels. These choices are made explicit during the *a priori* analysis, and their relationship with the research hypothesis and preliminary analysis is made explicit. The conception choices may relate to the situation task itself, its content, and also the resources offered to students. The *a priori* analysis also discusses how these choices influence the possible strategies students will adopt to deal with the task of the didactical situation; this helps researchers foresee the possible interactions (both among students and with teachers), relational dynamics, and cognitive and other obstacles.

   The goal of the *a priori* analysis, however, is not to predict the behavior of individual students but to create a generic and practical reference of the situation's learning potential and possible difficulties. This reference will later be used for comparison with the actual classroom implementation.

3. *Realization, observation, and data collection.* During the implementation, researchers collect data to be used in the next phase, the *a posteriori* analysis. The data collected aim to understand students' interactions with the milieu (that is, the set of elements with which students may interact; for a more in-depth explanation, see the following 7.5.1). The data are also helpful for understanding the extent to which choices made in the conception phase actually help students move from initial, naive strategies to more elaborate and complex ones that potentially involve learning.

   Usually, the data collected are observers' notes, student productions and records, and audio or video recordings. Researchers are in the position of observers during implementation.

   It is important to note that implementation often leads to some adjustments being made to the design during implementation itself, especially when DE concerns a sizeable didactical situation. In the realization phase, these adjustments must be documented along with the reasons why they were required and obviously taken into account in the following phase when the *a posteriori* analysis is carried out.

4. *A posteriori analysis.* The *a posteriori* analysis is concerned with comparing the data collected during the classroom realization with the *a priori* analysis. What have been

the convergences and divergences, and what do they reveal? What have been the unanticipated interactions? How can the contrast between the difficulties of the didactical situation and the learning potential be interpreted?

Note that there will always be differences between the implementation and the *a priori* analysis because the *a priori* analysis considers a generic and abstract student behavior that is obviously not present during the actual classroom implementation. Therefore, the validation of research hypotheses does not require an exact correspondence between *a priori* and *a posteriori* analysis.

However, understanding students' activity is made possible by the completeness and depth of the *a priori* analysis.

In summary, according to the Didactical Engineering methodology, the *a priori* analysis is compared with the *a posteriori* analysis of the classroom realization. The validation of research hypotheses is internal: it results from the epistemological conformity of the *a posteriori* analysis with the *a priori* analysis. Moreover, the *a priori* and *a posteriori* analyses develop in a circular process, in which each implementation can enrich and refine the *a priori* analysis.

It should be emphasized that there is a continuous interaction between the results of the different phases. The results of the *a posteriori* analysis may suggest not only the introduction of changes in the educational process design but also a new characterization of the content involved (going back to the preliminary analysis). From a broader perspective, the results obtained and the emerging open problems may also contribute to educational science, leading to new theoretical or methodological developments. In this sense, DE is not a developmental practice in which the results of already established research are transformed into educational proposals. In fact, it is a way to empirically challenge assumptions about the possibilities of knowledge diffusion and the phenomena that hinder it.

7.5.3 TDS, DE and participatory design

Relationships can be identified across Theory of Didactical Situations, Didactical Engineering, and participatory design approaches as educational research methodologies, especially when it comes to designing learning environments and activities that could meet teachers' and students' needs and expectations.

As reported in 2.9, participatory design is a collaborative approach involving users in the design process to ensure that the final product or solution meets their needs and expectations. Similarly, DE involves designing teaching sequences and learning environments in collaboration with teachers and students, considering their needs, expectations, and feedback.

The principles of TDS can inform the design process in both participatory design and DE by providing a theoretical framework for analyzing and understanding the learning and teaching process. By applying the principles of TDS to the design process, designers can ensure that the learning environment and teaching sequences are designed to promote learning and understanding.

Participatory design can also be used as a research methodology to study the learning and teaching process in the classroom. This approach involves engaging with teachers and

students as co-researchers, thereby providing insights into the learning experience from the perspectives of those involved. Similarly, DE can be used as a research methodology to investigate the effectiveness of teaching sequences and learning environments.

In summary, TDS provides a theoretical foundation for understanding the learning and teaching process, which can inform the design process in participatory design and DE. DE involves designing teaching sequences and learning environments in collaboration with teachers and students. Participatory design involves engaging with users in the design process to ensure that the final product or solution meets their needs and expectations. Additionally, participatory design and DE can be used as research methodologies to study the learning and teaching process in the classroom.

Part III

Original Contributions – Introductory Programming

Introduction to part III

In this first original part of the thesis, we present contributions from our research about introductory programming, a traditional (scientific and technical) mode of access to informatics. In an ideal scan (ideal since the reader will also be able to grasp intertwined references), this production is situated within the literature reviewed in part I. In particular, we sought to realize the vision described in the review, specifically the search for *optimal guidance* in active constructivist-inspired approaches (thus far from any orthodoxy of "minimally-guided constructivism"). In our work, we tried to balance as active and student-centered activities as possible with a variable amount of guidance. We aimed to provide scaffolding to activate Vygotsky's *zone of proximal development*, and it to be adaptive with respect to content, learners, and the particular moment in the "CS1-like" learning path.

We present our original learning design proposed to support novice students learning new programming concepts after reporting on the broader research in which the learning design was conceived. Next, we report and discuss the design and implementation of experimentation of our learning design in a real high-school context of learning C++ programming.

The last two chapters recount two more research initiatives, but always in the context of introductory programming. We report and discuss our experience teaching and researching the use of information technologies in emergency remote teaching, attempting to mimic an in-person experience for a highly-participated CS1 course for math undergraduates during the 2020 COVID-19 pandemic.

Finally, we report and discuss a participatory process where we collaborated with primary school teachers to co-design a learning module using the *Use-Modify-Create* methodology to teach iteration to second graders through a visual programming environment.

Chapter 8

Necessity of a Progression of Notional Machines

In this chapter, we report the preliminary results that emerged from an early attempt to answer the following research question: *what theoretical model for teaching programming can support novice students in learning the fundamental concepts of informatics?*

This question is relevant given the problems with teaching and learning introductory programming (described in 1 and 2) – including, in particular, the difficulty of developing the strategic knowledge that enables students to grasp also the core informatics concepts of programming (see 2.3.1) – and their disruptive consequences.

To tackle this challenge, we investigated two ideas. Foremost, the model could be based on a progression of *notional machines*, given the potential of such educational devices and their increasing relevance in programming education (see 2.3.4). A progression of notional machines might also be a more agile answer to the challenge of finding the optimal level of guidance by choosing the "right" programming language to learn to program (see 2.7).

Second, progressions could be driven by a mechanism of necessity: transitions from one notional machine to the next would be designed to make students "feel the necessity" to learn what is new in it. This latter inspiration comes from the *Learning Edge Momentum* hypothesis. According to it, the initial moments in learning are the most critical (see 1.4), leading us to seek ways (e.g., the mechanism of necessity) to sustain students' motivation and understanding in transitions in the learning path of introductory programming.

This work is also presented in the extended abstract "*A Necessity-driven Learning Design for Computer Science*" [Sbaraglia, 2021], published in 2021 in 'Proceedings of the 26th ACM Conference on Innovation & Technology in Computer Science Education' (ITiCSE '21).

8.1 Context and motivation

It is an accepted hypothesis that students without proper scaffolding (see 2) find programming concepts difficult to grasp, do not understand their programs' fundamental properties, and do not know how to control those properties by writing code. To provide that scaffolding, Du Boulay [1986, p. 57] first introduced the concept of *notional machine* to refer to

"*the general properties of the machine that one is learning to control*". Informatics and programming education research agree on the importance of explicitly teaching notional machines to foster the learning of programming [Sorva, 2013]. Section 2.3.4 contains more about notional machines, their characteristics, and their potential.

Also, earlier in 1977, Shneiderman described a "spiral approach" to learning programming that should begin with essential syntax, accompanied by simple semantics, to adapt to students' cognitive limits [Shneiderman, 1977]. The spiral approach idea has inspired different kinds of programming languages for learning, for example, mini-languages and incremental languages. Section 2.7 delves deeper programming languages for learning.

Science education research has proved PS-I learning approaches (where problem-solving activities precede formal instruction) effective [Sinha et al., 2021]. In the problem-solving phase, *failure-driven scaffolding* can improve PS-I efficacy further [Sinha et al., 2021]. Besides, research shows that *Productive Failure* (that intentionally designs for and uses failure in preparatory problem-solving) is most effective within a PS-I design and also suggests the possibility of developing improved variants of it [Sinha and Kapur, 2019]. An extensive review of PS-I approaches and Productive Failure is reported in 2.6.3.

These three elements (i.e., I. scaffolding through notional machines, II. incremental approaches to programming, and III. Productive Failure and PS-I approaches) form the framework within which we seek to answer the research question: *what theoretical model for teaching programming can support novice students in learning the fundamental concepts of informatics?*

8.2 Problem statement

Despite research raising the issue already decades ago [Shneiderman, 1977; Du Boulay, 1986], novice students still struggle to learn programming.

Notional machines – understood as abstract machines whose role is not in software architecture design but education architecture design – could aid learning, aiming for an "education of attention" [Fincher et al., 2020] to relevant and often not immediately visible aspects of programming education. Many teachers already use notional machines on several levels (e.g., they teach them explicitly; they use them as a reference in designing activities and courses) [Fincher et al., 2020] and with various purposes, even along the same learning path. Nonetheless, there are no established learning designs using notional machines to teach programming, and much research – theoretical and empirical – has yet to be done [Guzdial et al., 2020].

Besides, learning to program seems to be facilitated by incremental languages, like *Hedy* [Hermans, 2020]. The progression of language subsets (expanding in syntax and functionalities) suggests (and could be mapped onto) a progression of notional machines. Developing notional machines' progressions tailored to the specific needs of a learning context could be a way to avoid adapting the teaching to a chosen language or developing a specific one. Indeed, a progression of notional machines might be easier to realize or adapt than developing or adapting programming languages (for learning; see 2.7) to pursue the optimal level of guidance (see 2.5.3.6). In any case, we need appropriate environments and representations to

use notional machines for teaching and learning effectively.

Students encounter a bigger subset of syntactic rules and functionalities moving from one notional machine to the next. Sometimes, this transition passage also requires a change in the level of abstraction. Suppose, as an example, that after introducing a "*for-each*" iteration construct over simple sequences (such as strings or tuples), it is time to introduce a classic *for loop* on the indexes of those sequences, unveiling the possibility of accessing elements by index. This transition can be seen as a step downwards the abstraction ladder. The new, expanded language defines a new, less abstract notional machine. Students will need to use the new construct to express more sophisticated computations while seeing more detail of that machine (i.e., the index access to the sequence elements). Changing the level of abstraction can be difficult for students [Curzon et al., 2019, p. 533]. Therefore, recognizing this change is critical for teachers and educators and impacts learning.

8.3 Research Goals

We aim to develop a theoretical model for teaching programming to novice learners that could also support them in learning informatics core concepts. The model could be based on a progression of notional machines and embodied by a learning design specific to programming and informatics.

Many informatics educators already use sequences of notional machines [Fincher et al., 2020]. The model envisioned – inspired by research on spiral approaches to teaching and learning programming [Shneiderman, 1977], and particularly on incremental languages, such as *Hedy* [Hermans, 2020] – relies on carefully designed progressions of notional machines. The learning trajectory must start from simple and essential notional machines (whose syntax is simplified and semantics limited to a subset of commands and constructs), following a progression developed explicitly for the target notional machine. Moreover, progressions should vary according to the learning context.

When moving from one notional machine to the next in the progression, students will face a more extended notional machine, which, as mentioned, may also require some abstraction level drops. To stimulate students' motivation to use the next notional machine and learn what is new in it, we suggest using a *learning by necessity* design (more on this in the following chapter 9). Such design – inspired by Productive Failure [Kapur and Bielaczyc, 2012] and PS-I failure-driven approaches [Loibl et al., 2017; Sinha et al., 2021] – must realize the idea of learning something that is not yet known through the "feeling of necessity" experienced when dealing with a problem that stimulates such necessity.

Picking up on the previous example, having students struggle for a while on the task of swapping pairs of adjacent elements in a sequence (e.g., as in the first round of a *Bubble sort*) when they only know how to iterate over the entire sequence with a "*for-each*," will hopefully bring out the *necessity* of accessing more advanced tools for iterating over sequences.

8.4 Research methods

Designing and testing one (or more) progression(s) of notional machines to teach introductory programming is necessary for developing the theoretical model. The literature on incremental languages will help define the scope and characteristics of the notional machines of such progression. It will also help identify the main and most common obstacles in learning to program to design the *learning by necessity* transitions accordingly.

We aim to design and implement feasible experimentation, even in the short time of a Ph.D. program. Therefore, we plan to extrapolate from that first progression a module for learning a core programming concept (for example, the "indefinite iteration" with the *while loop*), starting from students' knowledge and experience of the previous concept in the progression (picking up the example, the "definite iteration" with the *for loop*).

Pandemic situation permitting, we intend to experiment with the module in an actual school setting. It seems ideal to test it in a specific Italian high school, that is, a *technical* (strand) *technological* (track) institute for *informatics* (sub-track). Specifically, we plan to test the module with one class in the high-school third grade (students about 15 years old), which is the first school year when students start systematically learning to program. We plan to compare results and student satisfaction with another third-grade class (from the same school year and possibly with the same teacher), which instead follows a more traditional approach to learning the "indefinite iteration".

Moreover, since notional machines are related to the programming language used, the potential for their visual representations should be factored in when choosing the language. *Snap!*[1], a block-based graphical programming language for learning, seems to provide an ideal environment to realize progressive language subsets. Indeed, since it is highly customizable, it allows new commands and constructs to be created and others to be hidden. That said, both our learning design and the theoretical model aim to be language-independent.

8.5 Early contributions

We designed prototypes of the first notional machines of a possible progression for learning introductory programming. These notional machines represent minimal Python and Snap! subsets made only by carefully selected or customized commands and constructs.

Moreover, in a local high school with a group of 14 years old students, we tested a series of activities to teach core cryptography systems and concepts (such as Caesar's cipher, One-time pad, and Diffie-Hellman protocol; more about this course and its implementation in chapter 14 of the following part IV). We developed a short progression of notional machines using Snap! as a language for cryptographic primitives (implemented with *ad hoc* blocks that were realized from scratch) and as an environment for teachers to build such notional machines and students to experience them hands-on. In addition, we designed a series of "necessity-driven" learning transitions to stimulate students' motivation to use and learn the new concepts encountered along the cryptography progression.

[1] https://snap.berkeley.edu/about

Furthermore, we have an initial collection of *learning by necessity* scenarios, some related to programming, others to cryptography. We plan to develop new ones: the more comprehensive the collection will be, the easier it will be to find the best transitions to drive any progression of notional machines.

One final observation: the fact that the first notional machines and progressions realized relate to both programming and cryptography demonstrates the potential of the envisioned learning design (and the related theoretical model) for teaching other informatics concepts other than programming.

8.6 Conclusions

The work described in this chapter was the first research effort to formalize our vision for learning introductory programming and, more generally, the fundamental concepts of informatics. This vision grew out of observations made in our experience as informatics educators and teachers. In particular, recognizing in students a "feeling of necessity" for concepts, and leveraging it as a possible driver for motivation and learning, an initial review of the literature, bringing out the elements described in 8.1. Using those "ingredients" made it possible to better describe our vision and the problem we aimed to be addressed with that vision. This was the basis that allowed us to formulate a more concrete - though still not fully focused nor sufficiently detailed - research plan and to describe the first contributions of our research activity. In general, this part of the work was an essential step in the definition and development of our learning design, *Necessity Learning Design*, described thoroughly in the next chapter.

Chapter 9

Necessity Learning Design

In section 1, we saw how introductory programming courses (often called 'CS1') are too difficult for many novices, making access to informatics a barrier to higher and necessary participation.

Considering that the LEM hypothesis (which we believe is the most credible explanation for difficulties in learning introductory programming; see 1.4) holds that the initial moments are the most critical in learning, we investigate *what kind of learning design can support novice students when introducing a new programming concept*.

Inspired by PS-I approaches (*Problem solving followed by instruction*) and *Productive Failure* learning design, we define an original "necessity-driven" learning design for introductory programming. Students are put in a situation that seems familiar to them; however, this time, they are missing an essential ingredient (i.e., the target concept) to solve the problem. Then, struggling to solve it, they experience the *necessity* of that concept. A direct instruction phase follows. Finally, students return to the problem with the necessary knowledge to solve it.

In a typical CS1 learning path, we recognize a challenging "abstraction rollercoaster". We provide examples of learning sequences designed with our approach to support students when the abstraction changes (either upward or downward) within a programming language, that is, when a new construct (and related syntactical, conceptual, and strategic knowledge) is introduced. We also discuss the benefits of our learning design in light of informatics education literature.

This work is also presented in the full article "*A Necessity-Driven Ride on the Abstraction Rollercoaster of CS1 Programming*" [Sbaraglia et al., 2021a], published in 2021 in the journal 'Informatics in Education' and available for open access.

9.1 Introduction and motivations

Today, most education research agrees that active methodologies – whereby learners actively explore and construct knowledge – are helpful for learning [Prince, 2004; Freeman et al., 2014]. On the other hand, educators often have to teach specific introductory or technical concepts that students are unlikely to learn or even discover through free exploration. Informatics also

faces this issue since it is a discipline with many technical aspects, especially in introductory programming [Guzdial, 2017]. As a result, in introductory programming courses, a common approach remains directly teaching language elements, usually followed by their application in programming assignments. This approach is called *direct instruction*, which indicates the explicit teaching of content and skills through lectures or demonstrations to students. It is based on explicit instruction, in contrast to exploratory models such as *inquiry-based learning*. However, direct instruction of technical concepts is not ideal for novice learners. Teaching a concept through a direct instruction approach is likely to bore students – also because they may not grasp the significance of that concept from their point of view – leading them to low motivation and poor learning outcomes. More about this is in 2.6.3.

We describe how we tried to tackle these challenges by investigating *what kind of learning design can support novice students when introducing a new programming concept*. We focused specifically on the introduction phase of a concept because the LEM hypothesis – which seeks to explain the bimodal nature of results in introductory programming courses such as CS1[1] (see 1.4) – argues that the initial moments are the most critical in learning. We answered this question by developing an original learning design specific to introductory programming. To stimulate students' motivation and support their understanding, we suggest fostering "necessity-driven" learning, that is, challenging students with a problem that makes them "feel the necessity" of something they do not know yet.

To this end, we first investigated what we later called the *necessity mechanism*. It is a particular disposition of students to be ready to learn a new concept after being in an instructional situation that requires it. In our experience as informatics educators and teachers (both in pre-university and university education), we have observed this phenomenon numerous times. We initially recognized a similarity to what is described in *A Time For Telling* [Schwartz and Bransford, 1998] from mathematics education (see 2.6.3), but without finding a more specific definition for it that was more aligned with our observations. In order to try to understand the nature of that learning disposition better and thus answer the question of *what that particular mechanism is* – and *how to foster it in students*, we defined a general mechanism that we called the "necessity mechanism".

Our pedagogical inspiration lies in approaches in which problem solving, as a preparatory activity, precedes instruction (PS-I) since this kind of approach can increase learners' motivation and improve understanding [Kapur, 2016; Loibl et al., 2017]. Moreover, we draw on *Productive Failure* learning design [Kapur and Bielaczyc, 2012], which shows that failing in the preparatory problem-solving phase favors students even more in learning from the following instruction phase.

In addition, we provide and discuss some concrete examples of how our learning design can be used to support learning to program in CS1, specifically the "bootstrap" of a new concept (e.g., introducing the *indefinite iteration*). In order to do this, we propose concrete examples of "necessity-driven" learning sequences and contextualize them within a CS1 learning path.

Another significant contribution of this research is an in-depth analysis of abstraction swings (i.e., a rollercoaster) from the learner's point of view. We tried to answer the

[1] As mentioned, with CS1, we indicate "a first course in informatics", in which students usually learn basic programming skills.

question of *what impact abstraction has on learning introductory programming* and, thus, *how abstraction influences a learning design to introduce programming concepts*. Indeed, already in CS1, we recognize that abstraction – a fundamental idea of informatics[2] – heavily comes into play. Therefore, the examples provided support precisely learning moments in a CS1 course when abstraction changes. These changes are challenging for novices because they require more effort and increase cognitive load [Curzon et al., 2019, p. 533]; it should also be considered that both upward and downward movements can be difficult for students. Furthermore, because upward and downward movements determine different learning scenarios, teachers and educators must consider the direction of abstraction movement when developing teaching activities with our learning design.

9.1.1 Outline

Section 9.2 presents a learning design for teaching introductory programming to novices, which we call "Necessity Learning Design" (NLD) for introductory programming. Subsection 9.2.1 describes the "necessity mechanism", which is the core element of our learning design. Subsection 9.2.2 describes our learning design, features, and phases in detail.

Section 9.3 proposes a concrete example of the application of our learning design in CS1 by using *necessity* sequences to support learning when abstraction changes (within the programming language of choice). Subsection 9.3.1 discusses how the choice of the learning path (i.e., which contents and their order) determines these movements and their direction (increase or decrease in abstraction). Subsection 9.3.2 shows how both directions pose challenges (albeit different) for learners and can lead to different teaching strategies. Subsection 9.3.3 proposes a CS1 learning path, mainly as an example to show where to place the *necessity* sequences we discuss. Subsection 9.3.4 shows four examples of learning sequences designed with NLD and presents the general structure of the examples as a tool for designing other *necessity* sequences.

Finally, section 9.4 reports the outcome of this research work and wraps up its key points. It also explains the limitations of the *necessity* mechanism and NLD (9.4.1), describes what did not go as we expected during the research (9.4.2), and presents its future developments (9.4.3).

Please note that the literature relevant to this research work focuses on introductory (or CS1) programming (see 1 and 2), Problem-based learning (2.6.2), Productive Failure (2.6.3.1), PS-I approaches (2.6.3.2), and abstraction in programming languages (mainly from the learner perspective; see 2.3.6). For the reader's convenience, we briefly recap such literature in the following 9.1.2.

9.1.2 Summary of relevant literature

Our literature review reports on the challenges of teaching introductory programming to novice students, which is the typical CS1 scenario (see 1 and 2). This is the scenario in which

[2] For a historical and epistemological review and a discussion on the fundamental elements of an "informatical way of thinking", see Lodi and Martini [2021] and Lodi [2020a].

the problem lies, and that motivates our general research question, that is, *what kind of learning design can support novice students in learning to program?*

Also, the literature review highlights the different types of knowledge involved in learning to program (i.e., syntactic, conceptual, and strategic; see 2.3.1) and how using well-designed examples can help students learn effectively (2.6.1). It also discusses the role of *cognitive load theory* and the *Learning Edge Momentum* (LEM) hypothesis in understanding how students learn to program (1.4). Most importantly, the LEM hypothesis suggests that teachers and educators should pay particular attention in the introduction phase of a new concept (see 1.4 and 2.6.1). This leads to a more refined research question, that is, *what kind of learning design can support novice students when introducing a new programming concept?*

Problem-based learning (PBL) is a learning methodology that presents students with problems for which they must autonomously study the material needed to understand and solve them; it focuses on learning goals rather than solving the problem itself. Problem-based learning has been used in many fields of education, including informatics. The literature review shows that it is effective for teaching programming to novices (see 2.6.2). Elements of project-based learning have been integrated into PBL for programming due to the nature of informatics and software development. While sharing with PBL the essential feature of using an exercise for which students need to acquire more knowledge to motivate learning, our proposed learning design (described in 9.2) has significant differences (that will be clearer in the following). For example, our exercises are small and surgically designed around a specific target concept; there is no teacher scaffolding because failure is at the core of the learning design; the target concept is taught later with traditional instruction.

As the literature review in section 2.6.3 reports, some science education research suggests that it is better to prepare students for instruction by engaging them in activities, and even allowing them to fail, rather than starting with direct instruction. These preparatory activities can include analyzing contrasting cases, generating answers, and encountering impasses in problem-solving. These challenges can help students develop the knowledge structures and deep understanding necessary to learn from direct instruction. The teaching approaches combining problem-solving and direct formal instruction in this specific order are called PS-I, which stands for *problem solving before instruction* (see 2.6.3.2). The literature review in 2.6.3.1 reports that *Productive Failure learning design* (PF) is the most popular among these. PF's *problem solving* is a generative exploration phase: students engage in a complex problem and generate multiple representations and solution methods (RSMs). PF's *instruction* is a consolidation phase: teachers organize and assemble relevant students' RSMs into canonical RSMs. Three core principles guide PF learning design: creating problem-solving contexts that involve working on complex problems, providing opportunities for explanation and elaboration, and comparing and contrasting failed or sub-optimal RSMs with canonical RSMs. It fosters the activation of three cognitive mechanisms: prior knowledge activation, awareness of knowledge gaps, and recognition of deep problem features. As we will discuss later, our design is inspired by PS-I approaches and especially by *Productive Failure* (see 9.2.1, while having its strong peculiarities (9.2.2.1).

9.2 Necessity Learning Design

In light of the reviewed literature (a brief recollection in the previous 9.1.2), we propose a *learning design* to support novices in learning introductory programming (often CS1), which we call "Necessity Learning Design" (NLD).

The core element of our design – which sits in the domain of PS-I approaches – is the *necessity mechanism*. What we call *necessity mechanism* is a learning mechanism originally defined from various sources of inspiration (see 9.2.1) that shapes the learning experience to support motivation, engagement and better understanding, by putting students in a situation that can stimulate in them the necessity of the concept that will be introduced afterwards.

In the following subsection, we analyse the *necessity mechanism*. Then, in subsection 9.2.2, we detail and discuss how we leverage this mechanism in our Necessity Learning Design for introductory programming.

9.2.1 Necessity mechanism

The *necessity mechanism* is prompted by assigning students a carefully designed *problem*; note that we use the term 'problem' accordingly to the PS-I literature, although the problem could be a simpler task, such as a programming task in our learning design. The problem is built so that, on the one hand, students feel like they can solve it with the knowledge they already have. On the other hand, however, *necessity* problems are constructed to be unsolvable except with a particular concept not yet taught (to which we refer to as the *target concept*, following PF and PS-I literature), making learners experience the need for it. When learners realize that, "surprisingly", they cannot solve the problem, it is the *time for telling* [Schwartz and Bransford, 1998]. That is when the feeling of strong necessity – generated by failing to solve the problem (or by struggling to find sub-optimal solutions) – can be leveraged to introduce the target concept.

For example, after students have learned the two main forms of definite iteration – i.e., sequential scanning (`foreach` loop) and loop with explicit but automatic index handling (`for` loop) – and after they have applied them to solve problems, a new problem requiring indefinite iteration is proposed. This new problem might be to count how many pseudo-random integers between 1 and 1000 a program generates before getting the number 42 (see necessity example 2). The problem is posed similarly to those faced when learning definite iteration so as to inspire confidence in students that they can solve it with the loops they have learned so far. However, since they cannot meet the problem request – at least not easily nor optimally (e.g., they might use the `for` loop with a very large number of repetitions) – students will hopefully feel a necessity (even an abstract one) of a construct that allows repeating until something occurs, rather than repeating a given number of times. The *desirable difficulty* of "*having to resolve the interference among the different things under study [i.e., the interference between what worked for the previous known problems and what is not enough now] forces learners to notice similarities and differences among them, resulting in the encoding of higher-order representations, which then foster both retention and transfer*" [Bjork and Bjork, 2011, p. 61] of the following instruction phase.

In designing *necessity* problems, we follow some of the PF principles on problem solving

to "*[c]reate problem-solving contexts that involve working on [...] problems that challenge but do not frustrate, [and] rely on prior [... students'] resources*" [Kapur and Bielaczyc, 2012, p. 49]. We aim to activate two of the Productive Failure cognitive mechanisms (reported in subsection 2.6.3.2): *prior knowledge activation* and *attention to the target concept's critical features*.

Also, in order to achieve the Productive Failure "sweet-spot calibration" of problems, we design them to be as similar as possible (i.e., using almost the same words, phrasings, scenarios, and requests) to the previous well-known problems students faced developing mastery of the previous target concept. Carrying on with the example, the last problem before the *necessity* problem (see example 2) might be to count how many times a program that generates 100 (or any fixed number of) pseudo-random integers between 1 and 1000 produces the number 42.

Most importantly, *necessity* problems are solved precisely by applying the new target concept, that is, the *smallest possible addition* to students' prior knowledge. As already recognized by Shneiderman [1977, p. 195], "*[a]t each step the new material [...] should be a minimal addition to previous knowledge, should be related to previous knowledge, should be immediately shown in relevant, meaningful examples and should be utilized in the student's next assignment*". In other words – following the PF principle of relying on learners' resources – a *necessity* problem must be finely tuned to students' prior knowledge so that they can fully understand its request, identify its significant features, and devise strategies for solving it. However, none of these strategies will lead to the canonical solution, as it requires applying the target concept. According to Bjork and Bjork [2011, p. 62] on the *generation effect*, problems "*can potentiate the effectiveness of subsequent study opportunities even under conditions that insure learners will be incorrect*".

As can be seen from this description, which (net of the programming example) illustrates a general mechanism, we believe that the necessity mechanism is exploitable in learning contexts beyond introductory programming. While we developed our learning design specifically for introductory programming (as described in the following 9.2.2), the necessity mechanism on which it is based can be used more generally in science education, as a tool immediately available to teachers and educators, or as a part of other methodologies.

9.2.2 Necessity Learning Design for introductory programming

In this section, we present and detail the *Necessity Learning Design* (NLD) that leverages the more general *necessity mechanism* to support novice students when introducing a new programming concept.

9.2.2.1 A necessity sequence: P!S-I-PS

Assuming the logic of PS-I approaches [Loibl et al., 2017], both to sustain motivation and because novices generally struggle to understand the significance of new concepts when they are introduced by direct instruction – instead of planning for the use of the target concept "in the student's next assignment" (see Shneiderman's quote 9.2.1) – we create a "meaningful

example" (i.e., the necessity problem), in which the target concept is needed immediately *before* the instruction phase.

Necessity Learning Design is greatly inspired by unscaffolded PS-I approaches (which resemble straightforward Productive Failure [Sinha et al., 2021]), drawing on the idea that problem solving best prepares learners for the instruction phase.

On the other hand, NLD deviates from PF in how it introduces the target concept after the problem-solving phase, specifically by differing in the purpose and implementation of the instruction phase. In the instruction phase of Productive Failure, the teacher introduces the target concept to students (building on their RSMs, which are their representations and solution methods; see 2.6.3.1) and uses the target concept to build the canonical solution to the same problem of the previous phase. By contrast, the instruction phase in Necessity Learning Design only introduces the target concept in general. The target concept is presented in its essential features, through the easiest examples possible, and most importantly, *without applying it to solve the problem*.

After the instruction phase, NLD requires students to return to the problem of the first phase (whose purpose was for them to experience the necessity of the target concept; see 9.2.1) that is precisely structured to be solved using the target concept just presented.

Inspired by the **PS-I** notation adopted by Sinha and Kapur [2019] to formalize the approaches where problem solving precedes instruction, we describe our learning design as a *P!S-I-PS* approach. *P!S* emphasizes that *necessity* problems are unsolvable (*!S*, with the exclamation mark used as the 'not' operator, as is the case in many programming languages) before the instruction phase. The *PS* at the end indicates a second problem-solving phase, in which students return to the same problem after the instruction phase.

P!S phase: unsolvable problem solving The "problem" posed in our P!S phase diverges from Productive Failure in the "complexity of the problem" design principle. PF designs problems to be complex and information-rich (providing even unnecessary data) so that it is natural and inevitable for students to generate multiple RSMs. That is because of the essential role of students' RSMs in the "consolidation phase", in which PF builds *direct instruction* on "*organizing and assembling the relevant student-generated RSMs into canonical RSMs*" [Kapur and Bielaczyc, 2012, p. 49]. While also programming tasks admit various RSMs (always sub-optimal without the target concept, as discussed later in 9.2.2.3), the Necessity Learning Design does not rely on students' RSMs to introduce the target concept, nor does it build the canonical solution together with students in the instruction phase.

Among the reasons for this different sequence is the nature of the "problems" – better said *tasks* – that can be proposed to students learning introductory programming. The solution to a programming task is a program, and a program is an interactive object. The student who develops a program can immediately check whether or not her solution can be executed. A non-executable program is a first obvious indication of an error. On the other hand, if a program is executable, there are often many possibilities for the student to verify whether it works correctly. In other words, the student's program is an interactive solution attempt, which returns information helpful for solving the problem. This unique characteristic of programming problems constitutes a source of motivation for students to

experience first-hand the use of the target concept to solve the *necessity* problem.

Moreover, a specific consideration can be made for introductory programming. Since it involves introductory and mostly technical knowledge, there is inevitably less room for various RSMs and debating. On the one hand, confronting and discussing different solution strategies (before instruction and after the second PS phase) is undoubtedly valuable. However, on the other hand, the actual learning trigger of our learning design is not the discussion on students' RSMs but the feeling of necessity induced by the *necessity* problem.

While it would be unnatural (because of the introductory and mostly technical knowledge) to design complex tasks with multiple RSMs to teach introductory programming, we focus on developing meaningful but minimal problems that are "surgically" precise for the target concept. *Necessity* problems for introductory programming are built on students' prior knowledge but in such a way as to be solved only with the target concept as if the programming task were an encrypted message and the target concept the key.

I phase: (general) direct instruction As anticipated, in the instruction phase of Productive Failure (and, more generally, of PS-I approaches), the teacher illustrates the target concept and then applies it to solve the problem of the PS phase. In our scenario, presenting the solution to students would waste a precious *learning potential*, that is, the feeling of necessity (generated by the *necessity mechanism*, see 9.2.1) students have experienced in the P!S phase.

Our choice is backed by the "optimal tutoring strategy" formulated by VanLehn et al. [2003] in their study of what actions of human tutors cause learning within intelligent tutoring systems. Instructors, after students reach an *impasse* (as it happens in our **P!S** phase), "*prompt them to find the right step [...], and [...] provide an explanation only if they have tried and failed to provide their own*" [VanLehn et al., 2003, p. 245]. Thus – if we equate 'providing an explanation' with 'writing a program' – not revealing the solution already in the instruction phase to allow students to try autonomously to apply the target concept to the PS task is an "optimal strategy".

PS phase: informed problem solving In addition, all the reasoning students do in trying to solve the problem before knowing the target concept can be valuable not only to understand the concept's significance but also to realise how to apply it to solve the problem. Indeed, applying a concept is a further and more challenging step than simply understanding it. From the learning perspective, understanding a programming concept corresponds mainly to the conceptual level, knowing how to apply it to the strategic level (on the different kinds of knowledge in learning to program, see 2.3.1).

More complex to try to frame the expected learning outcomes at the end of a necessity sequence within the SOLO taxonomy. We hypothesize that the educational success of a necessity sequence can lead novices to the *multistructural* level (3 out of 5). We believe that reaching the more advanced levels (*relational* and *extended abstract*) requires developing mastery (more in 9.2.2.2). In any case, further theoretical and experimental research efforts would be needed to test this hypothesis

For a summary comparison between Necessity Learning Design and Productive Failure, see Table 9.1.

9.2.2.2 Learning between Necessity Sequences

Necessity Learning Design only aims to structure some specific moments of an introductory programming learning path for novices, precisely when a new concept is introduced (more about such moments in 9.3.1 and 9.3.2). In other words, the sequence (i.e., P!S-I-PS) of *necessity* activities is not in itself sufficient for students to fully develop the syntactic, conceptual, and strategic knowledge (see 2.3.1) of the target concept or to reach. Similarly, in terms of SOLO taxonomy (2.3.2), the necessity sequence alone does not enable students to reach the highest levels of knowledge (*relational* and *extended abstract*). For this to happen, it is essential that, after every occurrence of a *necessity* sequence, students are exposed to significant examples of the related target concept and other problem-solving situations[3] and have enough time to develop mastery. More generally, teachers and educators should adopt all the best practices that informatics education research suggests to support students in fully developing the knowledge and skills needed to learn to program.

Necessity Learning Design's characteristic of focusing on crucial learning moments (i.e., introductions of new concepts) does not undermine its general purpose (i.e., addressing the challenge of introductory programming) or its usefulness. Indeed, precisely because NLD immediately stimulates the use of the target concept in a problem-solving context – in which it is "the right thing to use" – our learning design aims to be an ideal starting point for developing especially the strategic knowledge (or the highest level of the SOLO taxonomy) of that concept.

9.2.2.3 Hard vs. Soft Necessity

The *necessity* of a programming concept (and the related construct) can be considered from two different perspectives.

From the programming languages perspective, there is only one genuinely *hard* necessity – that of the brute-force ability to program a specific function. It shows up only when the (subset of the) language in use is not Turing complete. If we allow only (true) definite iteration, then, for example, we cannot write programs that may loop forever for specific input values. However, modern languages are all Turing complete (at least in their "standard model" [Martini, 2020] where arbitrary resources are allowed). After all, besides the ability to memorize data of potentially unbounded size, only selection and indefinite iteration (or recursion) are needed for Turing completeness.

On the other hand, Turing completeness does not tell the whole story. The brute-force ability to program any computable function usually involves unnatural coding of data and processes. Lists, for instance, are not needed for such completeness. We may always encode a generic list of integers $[x_0, x_1, x_2]$ with *a single* integer $2^{x_0} * 3^{x_1} * 5^{x_2}$ (*Gödelization*) and implement all list operations through prime factor decomposition. However, this possibility

[3] Please, recall that the programming tasks students face in this phase are similar to the *necessity* problem of the following sequence, see 9.2.1

Table 9.1: Comparison between Productive Failure (unscaffolded PS-I) and Necessity Learning Design (P!S-I-PS)

Productive Failure (PF)	**Necessity Learning Design (NLD)**
PS	**P!S**
Problem built on students' prior knowledge	
Problem sweet-spot calibration: challenging but not frustrating	
Problem: complex, information-rich and ill structured (what-if scenarios)	Problem: simple, well structured, very similar to those the students are already used to
Problem stimulates multiple RSMs	Problem surgically designed to stimulate the necessity of the target concept
No teacher scaffolding	
Students discuss to confront different RSMs	RSMs are programs, tinkerable objects (self assessment and trial & error)
Students present their work and compare (guided by teachers) affordances and constraints of failed or sub-optimal RSMs to assemble the canonical RSM	–
I	
Teachers introduce the target concept in the context of the problem and solve it	Teachers introduce the target concept without referring to the problem (showing the solution would waste the "necessity learning potential")
Students practice on well-structured problems on the target concept	–
Teachers formally introduce the concept and explain it through simple examples	
Students practice on isomorphic problems	–
–	**PS**
–	Students immediately apply a concept (strategic knwl) after learning it (conceptual knwl) in a meaningful and well-known context (P!S problem)

(and others of this kind) is irrelevant for novice students in an introductory programming learning context.

Therefore, we better consider an educational perspective and identify an "educational" *hardness* of *necessity*, which shows up exactly when the available programming tools express the new concept (i.e., the target concept) only through unnatural or too complex coding. As a matter of fact, it is very unlikely for novice students to be able to produce such advanced – yet unnatural and complex – solutions at the point where they are in the programming learning path. This educational perspective is bound to the chosen programming language and the order of topics in the learning pathway.

Therefore, the *educationally hard necessity* of a target concept appears when students, drawing only on what has been taught in the course so far – hence without the target concept – can produce only sub-optimal solutions to a given problem or task. Indeed, most of the time, using the constructs learned until that moment in its "standard way"[4], students will only be able to produce partial solutions, i.e., solutions that do not solve the problem in all possible cases. However, recall that – apart from the exceptional cases of a Turing-incomplete language (subsets) – complete and general solutions (i.e., solutions that work for all cases) are always technically available, although through more or less fancy hacks. Nevertheless, as said, it is very unlikely (though not impossible) that a novice student could circumvent an *educationally hard* necessity problem. For example, students might "force" a definite loop construct they know to express an indefinite iteration, e.g., in Python, using a `for` loop over a list and increasing the list length in the repetition body[5]. Solutions such as this would demonstrate a student's remarkable mastery of the concepts preceding that moment of *necessity*. In these (rare) cases, teachers should make these students realize that such solutions, though workable and clever, are nonetheless sub-optimal hacks (often inelegant, inefficient, unnecessarily complicated) and prone to introduce issues as the complexity of tasks increases. Students who can circumvent a *hard necessity* problem are also likely to be receptive to such clarifications.

By contrast, in cases of *educationally soft necessity*, sub-optimal solutions formulated without the target concept can, quite easily, solve the problem completely, just relying on students' previous standard knowledge. In such cases, it is not unlikely that students will be able to circumvent a *soft necessity* problem since they could "blindly" use what they already know. For example, with reference to the turtle geometry sequence (see example 1), drawing a polygon of 20 sides can be done by blindly replicating the same code 20 times. In general, in *soft necessity* cases, sub-optimal solutions may be fully functional (though often not general). However, they are always one or more of the following: less compact (e.g., because of repeated code) – and therefore less maintainable and more prone to errors, less modular (e.g., not using functions, not adopting OOP), less clear (e.g., intricate solutions), and also possibly less efficient (i.e., more computationally costly in time, space or both).

From the educators' perspective, being aware of whether the *necessity* is *educationally*

[4]The canonical usages of a construct, as taught to students.

[5]Typically, in most modern languages, constructs designed for definite iteration (e.g., `for`, `foreach`) can be forced to express indefinite iteration, sometimes in more natural ways (e.g., in C), sometimes in unorthodox and inelegant ways (such as in the Python example just mentioned).

hard or *soft* helps design the related *necessity* problem. If it is *educationally soft*, the *necessity* problem should be structured in such a way as to make it as difficult as possible to circumvent the problem by blindly using prior knowledge. Conversely, if it is *educationally hard*, the *necessity* problem should require exactly what is (educationally) impossible to do without the target concept. In summary, the goal is always to stimulate the necessity of the target concept, without which it must be (almost) impossible (i.e., *educationally hard necessity*) or hindered (i.e., *educationally soft necessity*) to develop a complete solution.

To clarify what was said in a concrete context, we refer to a scenario where the problem is to input a list of students' names and randomly choose the first to take the oral exam.

1. If the number of students is known[...]

   ... and it is very small.
   No *necessity* is stimulated since it should be fairly simple even for inexperienced students to use separate variables to input the names, extract a random number, and use a simple selection (with few `elseif` branches) to print the corresponding student.

   ... and it is large.
   It is still conceptually easy to solve the problem with unrelated variables and a (long) selection with many `elseif` branches. However, the solution code would be long and repetitive, and errors would likely be made. Hence, even though it is possible to solve the problem completely, writing such a solution is prone to errors and could be frustrating. This scenario can stimulate in students a *soft necessity* of some kind of collection with index access (like arrays).

2. If the number of students is unknown.
   An *educationally hard necessity* is stimulated. Students can produce an *incomplete* solution assuming a large maximum number of students, leading back to the solution of the previous point. Of course, this solution would be highly sub-optimal. In this case, we are dealing with an *educationally hard necessity* (this time, of some kind of dynamically extendable collection with index access, such as Python lists) because students have no "standard" way[6] to produce a *complete solution* (i.e., to memorize an arbitrary, unknown number of students only using simple variables).

In conclusion, the *hardness* (and the effectiveness) of any *necessity* sequence is determined by both the chosen programming language and specific topics order in the learning path. This explains why the information in each of the presented *necessity* examples always includes "What students already know" and, when relevant, a warning about other concepts that students should not know at that time.

[6] See higher up in this discussion how it would be possible to simulate a list only using a single integer variable (*Gödelization*).

9.3 A use of NLD in the CS1 abstraction rollercoaster

In section 9.2, we introduced and described Necessity Learning Design. Here we present a possible application of NLD in some moments that we consider crucial in a CS1 course, which are those in which the abstraction level changes in relation to the constructs of the chosen language (see 2.3.6).

From the literature review on abstraction (2.3.6), we briefly recall that abstraction in informatics and programming languages is used to hide information and manage system complexity. Even an entire programming language can be seen as an abstraction of the underlying physical machine. Abstraction can be used to simplify the implementation of a system without affecting its behavior and applies both to control and data. Control abstraction involves hiding procedural data to define new control structures in a programming language; data abstraction involves hiding data structures to define new data types. Whether on control or data, students must follow a change in abstraction level when learning a new construct (of the same language), either downward when it is less abstract (usually more powerful and detailed) or upward when it is more abstract (more concise and evocative).

Simply put, abstraction from the learners' perspective is a lens through which we analyzed the introduction of new programming constructs. Ultimately, this lens helps in the design of *necessity* sequences.

We begin by discussing these movements of abstraction (9.3.1) and the relevance of their direction (9.3.2). Then we present a possible learning path for CS1 (9.3.3). Finally, within this path, we describe four *necessity* sequences – also detailing their general structure – that we developed as concrete examples of how to support learners in introducing four core programming concepts (9.3.4).

9.3.1 Abstraction movements in introductory programming

The history of programming languages, from low level to high level, clearly shows an abstraction process, with the introduction of more and more abstract constructs (for the notion of *type*, see Martini [2016a;b]), or with the creation of more and more abstract languages, sometimes maintaining low-level functionalities (even when not strictly necessary for expressiveness), other times sacrificing them for cleanness.

Creating these abstractions in programming languages is a complex process driven by experiments and semantics. In Visser [2015], we find a discussion about this abstraction creation process. First, a *programming pattern* – a "recipe" for solving a re-occurring problem, which the programmer applies manually in any instance (e.g., calling and returning sequences in assembly language using the return stack) – is identified. Then, a linguistic *abstraction* – a construct providing a "black-box" for that pattern (e.g., functions and the passing mechanism of their parameters) – is devised and realized. The essential point is that a "good" abstraction, once created, gets autonomous life because it captures an important concept of software development.

Professionals have welcomed this movement upward the abstraction ladder – because it enables them to express more and more complex computations in a simpler, more evocative,

and more concise way – and resort to lower-level constructs (or languages) only when necessary (e.g., for efficiency's sake).

By taking an expert perspective, it could be tempting to *always* follow the same upward direction (usually called a *bottom-up approach* [Caspersen, 2018, p. 113]) and follow an ascendant abstraction path when teaching programming to students. That is, starting from lower-level constructs so that, when a higher-level construct (abstracting the former ones) is introduced, students can fully understand its underlying details. However, as already noted by Shneiderman [1977, pp. 195,196] – and in line with the studies on cognitive load theory (see in particular 1.4.1) – students "*would be overwhelmed if the general form were presented first, [while they] can absorb complex forms in a step by step process*", therefore "*[n]on-essentials must be stripped away so as to provide students with a minimum useful subset of the language which can be expanded gradually*". Indeed, a small subset of abstract constructs (e.g., `print`, `if`, `foreach` loop) is enough to produce functioning and meaningful early programs (e.g., given a sequence of strings representing the XML code of a social network posts, print out only those mentioning your name), and thus sustain motivation in novice learners [Caspersen, 2018, p. 113]. Furthermore, some studies show that implicit looping is more natural for novices than explicit looping [Guzdial, 2008], hinting at the possibility of teaching more abstract constructs before the less abstract ones (i.e., focusing more on "what" than on "how", following Statter and Armoni [2020]).

Hence, in some cases, educational purposes can take the abstraction ladder downward in the opposite direction of the one followed to introduce more abstract constructs in programming languages. For example, instead of going upward from using the `while` loop for *definite* iterations (i.e., by manually handling an index/counter) to the `for` loop with explicit but automatically handled index, up to the `foreach` loop with implicit (and hidden) index handling, we can take the opposite route by starting to teach the more abstract loop (i.e., the `foreach` loop) and going down from there. Of course, learners in introductory programming courses will soon face situations in which the most abstract construct is not sufficient anymore (or it is less elegant, simple, efficient). When this happens, students can feel the *necessity* (see 9.2.1) of "opening the black box" to have a less abstract but more powerful mechanism.

On the other hand, during a typical introductory programming/CS1 course, adding more constructs often raises the abstraction level. For example, in our proposal, this happens when introducing a dictionary-like data type (i.e., a mapping between two finite collections of arbitrary types; example 4). Since it helps solve problems handled before with parallel arrays (one containing the indexes and one the values), introducing dictionaries can be seen as an increase (compared to using parallel arrays) in data abstraction.

We observe that the order in which concepts are introduced determines the direction of the movement of abstraction. In the previous example on loops (i.e., moving down towards the less abstract `while` loop), the abstraction movement would be upward if the general form (i.e., `while`) were presented before the more abstract one (i.e., `foreach`).

As mentioned in 2.1, there is no agreement in the literature on the best topic order to follow in introductory programming/CS1 courses. Consequently, introducing the same concepts in different courses may correspond to opposite directions in the abstraction movement due to

different choices on the order of topics. Moreover, we observe that a typical introductory programming course usually goes up and down the abstraction ladder (even more than once) along its learning path.

Hence, considering the multiple changes of abstraction level across the typical abstraction mechanisms of any CS1 programming language, we recognize what we call a *rollercoaster of abstraction*.

In 9.3.3, we present a comprehensive CS1 proposal and discuss the abstraction movements (and their directions) within its learning path; we also discuss the possibility of alternative paths and, thus, different abstraction movements and directions.

9.3.2 Abstraction ups and downs: different and difficult

As anticipated at the end of 2.3.6, movements between levels of abstraction pose specific learning challenges.

Moving downward the abstraction ladder is difficult because novices face the challenge of dealing with more details than before (see 1.4.1 on cognitive load). Indeed, a drop in the level of abstraction requires students to consider additional information and determines a less straightforward way to do things; however, it allows for more sophisticated computations. A metaphor that speaks well of this difficulty is that of the car transmission. It is intuitive to understand how a person who has learned to drive automatic cars finds it challenging to drive a manual car because it requires knowing and maneuvering more things.

However, moving upward the abstraction ladder can be, more surprisingly, difficult as well. A study conducted by Alexandron et al. [2012, p. 157] on *live sequence charts*

> showed that some [students] felt that the high abstraction level does not give them enough control, though the goals of their program were achieved without getting into lower level details. [... T]his subjective feeling is strongly related to [...] students' previous programming experience. Since the students were used to working on lower abstraction levels, it determined their perception of what the 'right abstraction level' is. When moving to a higher abstraction level, they could not control things that they used to control before, and thus felt that they lose power.

Picking up on the car transmission metaphor, it is quite common for people who are used to driving manual cars to report difficulties (at least at first) in switching to an automatic transmission. They usually complain of less control – it is impossible to control the engine brake in the same way as in a manual car – and disorientation.

Therefore, since riding the abstraction rollercoaster of learning introductory programming seems challenging both upward and downward, we propose examples of how to use Necessity Learning Design to support learning at these critical abstraction movements. We think NLD can support student learning in these critical moments. This is also because of the multiple roots on which we developed it (recalled in 9.1.2), for example, Productive Failure to best prepare for instruction and the constructivist idea of "generation effect" that can favor a positive learning edge momentum and solid and deeper learning.

Moreover, recognizing the direction of the abstraction movements when introducing a new construct (and the related concept) helps design its *necessity* sequence. Indeed, whether abstraction goes up or down, the nature of the *necessity* problems changes as scenarios characterized by different *necessities* arise.

Reducing the abstraction level causes students difficulties because it requires them to see and deal with previously hidden details. We observe that in these cases, teachers and educators must stimulate in students a *hard necessity*, which is usually more straightforward to seek when abstraction drops (see 9.2.2.3). This means proposing to students a *necessity* problem (educationally) impossible to solve without those additional details.

Differently, increasing abstraction also causes difficulties for students since it usually takes away details they learned to control. In these cases, it is difficult to find *hard necessity* scenarios. Indeed, more abstract tools (i.e., commands and constructs) are intended to facilitate tasks (compared to using more trivial tools) rather than enable new functionalities (see 2.3.6). Consequently, when abstraction increases, teacher and educators must stimulate in students a *soft necessity* (see 9.2.2.3) to use the more abstract tool. This can be done by developing programming tasks that make it complicated (i.e., time-consuming, tiring, prone to errors) to "blindly" use the less abstract tools that students are already familiar with, effectively creating the opportunity to use the more abstract ones.

As informatics teachers and educators with many years of experience (both in tertiary and pretertiary education), we found that developing programming tasks to foster learning when abstraction increases is more challenging than when abstraction drops. Therefore we provide just one example of Necessity Learning Design used to support a downward movement (example 2) and three examples of NLD supporting upward movements (examples 1, 3, 4). All examples are framed into a (proposal of) CS1 learning path, which we describe in the following 9.3.3.

Abstraction movements in learning programming and program design

As a final reflection, it might be intriguing to investigate possible correspondences between program design strategies (top-down and bottom-up; 2.3.5) and the ups and downs of introductory programming (9.3.1). While remaining in the domain of programming education, the rollercoaster concerns a different, more fundamental aspect of the learning phenomenon. Educators who want to use NLD need to understand that in learning programming, novices are required to move up or down in abstraction every time a new construct is introduced. It is a dimension that does not seem to intersect with that of planning, and the two abstractions (ups&downs vs. top-down&bottom-up) do not seem to be linked. These are just two of the many declinations of the concept of abstraction in informatics.

It may be added that a new programming concept is introduced in NLD by an example (P!S phase). In this sense, we are using a bottom-up approach as far as the planning is concerned. This bottom-up approach will be consolidated in the NLD later phases (i.e., alignment phase and consolidation exercises) by seeing more examples and delving more into the theory than during the quick introduction of the essentials (I phase). Despite this interpretation of NLD from the planning perspective, we still do not identify correspondences between the two abstractions that could be valuable from an educational standpoint.

9.3.3 A possibile CS1 learning path

In the following, we report the contents of a possible CS1 learning path where we contextualized the examples provided in subsection 9.3.4.

The course we present here follows a rather classical approach to introductory programming. It is based on the CS1 for math majors we have successfully experimented with over the last ten years, both as a traditional in-person course and as a fully online (yet synchronous) one (for two years because of the COVID pandemic; see 11 and also [Lodi et al., 2021b]).

Differently to an entirely classical progression (as proposed in some textbooks, e.g., Guttag [2021], Downey [2015]), we propose to begin with a `turtle` module. We use it as a tool for a playful and creative introduction to programming, influenced by Papert's constructionist approach with Logo turtle geometry [Papert, 1980] (of which the `turtle` module is a Python implementation). Indeed, as in renowned courses (e.g., Harvard's CS50[7] or Berkeley's CS10[8]), this introductory part can be approached with visual languages such as *Scratch* or *Snap!*, which provide turtle primitives and simple repetition constructs such as `repeat N`.

Our proposal of a CS1 pathway follows, instantiated with Python language but easily adaptable to any other high-level imperative language. The learning path includes the main programming concepts and constructs, along with Python language features realizing them, elementary patterns, notable algorithms, and theoretical aspects. Please note that the target concepts of the *necessity* examples provided in 9.3.4 are indicated in bold italics.

Example of CS1 learning path for non-majors

- Importing constants and functions from a module
- Basics of the `turtle` module
  - `forward` (`backward`) function
  - `left` (`right`) function
- `for` loop to express a simple `repeat N` with `N` being a fixed integer value (e.g., `for i in range(10)`) – **example 1 target concept**
- Built-in data types
  - Integers, floats
  - Strings and their basic operations (including selection of a character)
- Variables and assignment
- Functions (calling them, passing paramenters, defining custom functions)
- Input and output (`input` and `print` functions)

continued on next page

[7] https://cs50.harvard.edu/college/2021/fall/syllabus/#lectures
[8] https://cs10.org/su21/5syllabus/

- Type conversions (e.g., `int()`, `str()`)
- `random.randint` to generate a random integer in an interval
- Boolean type and boolean expressions
- Conditional commands (`if`, `if...else`, `if...elif...else`)
- `for` loop to iterate over sequence elements ("`foreach`" loop)
- Elementary pattern: linear scan (for each element of a sequence, do some operations)
- Tuples (and generalised assignment)
- Slices on sequences (`[start:stop:step]`)
- Elementary pattern: linear scan with a gatherer
- Idea of object and methods
  - `is` vs. `==`
  - Methods on built-in types (e.g., `str`, `float`)
- `range` and `for` over a `range` to express the definite iteration with explicit but automatically handled index (e.g. `for i in range(1,10,2)`)
- Elementary pattern: linear search
- `while` loop to express indefinite iteration – **example 2 target concept**
- Algorithm: Euclidean algorithm to compute GCD
- Pattern/algorithm: binary search
- Computational complexity: linear vs. logarithmic
- `matplotlib.pyplot.bar` to plot a simple bar chart
- Arrays (or lists) with index access – **example 3 target concept**
- Lists (mutability, dynamic insertion and deletion of elements, list methods)
- Recursion
- Computational complexity
  - resources, computational steps, cost of elementary/non-elementary steps, big-Os

continued on next page

  - complexity of binary search
  - complexity of numeric algorithms
- Sorting algorithms: insertion sort, quicksort, merge sort
  - their computational complexity
- Dictionaries – **example 4 target concept**
- List comprehension and dictionary comprehension
- Generators
- Object-oriented programming (OOP)
  - Classes and objects, attributes and methods
  - "Magic" methods
  - Private attributes, instance attributes
  - Subclassing and inheritance
- Binary tree abstract data type (implemented with OOP)
- The Halting Problem

This pathway is just a proposal, and we provided it mostly to place the *necessity* examples in a concrete context. Also, the examples we developed using Necessity Learning Design can fit into other introductory programming learning paths, provided that the prerequisites are met and the topics students should not know (to stimulate the various *necessities* of the examples) have yet to be addressed.

With the same warning about dependencies between topics, it is worth mentioning that NLD can be used to support the introduction of other concepts. Indeed, the examples presented here are just a few of the many possible *necessity* sequences that can be developed. For example, to introduce object-oriented programming, teachers and educators might use NLD to stimulate the necessity of a coherent representation of an object instead of relying on unrelated data structures and functions. This transition corresponds to an upward movement of (data and control) abstraction, which can be set in a *soft necessity* scenario.

9.3.3.1 Ups and downs in our and in other paths

From the viewpoint of abstraction movements, our learning path immediately shows ups and downs. In the beginning, we intuitively introduce the `repeat N` loop (with `N` being a fixed integer known at compile time). The `repeat N` loop is realised in Python with a `for i in range(N)`. Leaving initially unexplained what `i` and `range` are, we teach a simplified use of the construct that, in this case, serves only to repeat `N` times a block of instructions. At this point, the abstraction goes up a bit because the first "real" Python loop we teach after the

`repeat N` loop is the `for` on sequences (`foreach`). After that, the abstraction goes down as we explain what a `range` is and how to use the `for` construct to iterate over a `range`, to have (in Python, to simulate) a `for` loop with explicit but automatically handled numeric index. The goal is to access sequence elements by index (and not just *repeat N times* a block of instructions). A *hard necessity* sequence can be constructed around the need to use indexes in a non-trivial way, e.g., to access the previous/next element or only specific parametric positions. Abstraction goes down further when the `while` loop is introduced to express indefinite iterations (see example 2).

The rollercoaster described so far is experienced by students not only at the crucial moment of learning those new constructs/concepts. Indeed, students, in order to solve future problems, will always have to move between the different abstraction levels encountered to choose the construct at the most suitable level for the problem at hand. Clarified this, we observe how abstraction tends to increase or remain constant from this point onwards in the proposed learning path.

However, since the path in 9.3.3 is only a proposal[9], it is easy to imagine other learning paths with different movements, showing that the rollercoaster is inherent in introductory/CS1 programming.

For example, Python provides a powerful tool for "inline" list construction, *list comprehension*[10]. This construct is more abstract than the classic elementary pattern that uses a `for` loop and an empty list as a gatherer. In addition, it features a syntax inspired by the mathematical notation of set construction (i.e., note the similarity of $\{x^3 \mid x \in [0..4]\}$ with `[x**3 for x in range(5)]`). It has been found that non-programmers intuitively prefer constructions such as list comprehensions [Pane et al., 2001, p. 258]. It is possible to introduce this list-building construct first, a choice that makes particular sense, for example, for math majors, given the similarity of list comprehension with the intensional mathematical representation of sets. The necessity of learning the `for` loop – therefore going down in abstraction – can then be stimulated when creating the list is not the only thing that needs to be done but when other operations also need to be performed at each iteration.

Generally, an approach that starts from more abstract constructs – thus deviating from the more established tradition of teaching programming from the basics and abstracting from there – might be beneficial in domain-specific contexts other than informatics. Suppose researchers or professionals outside informatics need to learn to program. In that case, the level of abstraction of the tools they might be interested in is likely the one closest to the type of data they have and the elaboration they need to do.

According to this perspective, for instance, it would be possible to learn the *tree abstract data type* first and only afterwards, when the necessity of modelling more sophisticated (than trees) data structures, learn how trees can be implemented through lists. Similarly, as with list comprehension, it is possible to teach a powerful construct such as *generators* first in

[9] As already reported in 2.1, there is no "right" order of topics universally accepted by the informatics education community.

[10] According to Python documentation, list comprehensions "*provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.*" https://docs.python.org/3/tutorial/datastructures.html

their most abstract and intensional form provided by Python. Afterwards, it will be possible to descend the abstraction ladder to teach generators' explicit construction and the more sophisticated commands (e.g., `yield`) when necessary (e.g., when a mathematician needs to handle a large list of primes without allocating it all in memory).

9.3.4 Examples of NLD use in abstraction movements

The following illustrates four examples of Necessity Learning Design used to introduce core programming concepts. The proposed examples are designed for Python, but – unless there are specific limitations – they can also be adapted to introduce the same concepts in other imperative languages.

The examples presented below are *necessity* sequences. A *necessity* sequence is the succession of three phases: **P!S**, **I** and **PS**. Each sequence, designed according to NLD, aims to teach a concept that determines a change in the abstraction level in the use of the programming language.

Each example is presented in a box divided into three parts: *necessity* scenario, *necessity* sequence, and example characteristics.

The *necessity* scenario provides the following information.

- **Title**. It evokes the necessity stimulated by the sequence.
- **Problem in a nutshell**. A concise description of the problem (i.e., programming task) that aims to stimulate the necessity for the target concept.
- **What students already know**. It lists the knowledge students should have to deal with the sequence, expressed at the syntactic, conceptual, and strategic levels (see 2.3.1).
- **Target concept**. The new minimal addition to students' previous knowledge we want them to learn, expressed at the syntactic, conceptual, and strategic levels (see 2.3.1).

After the first dashed line, the second part describes the *necessity* sequence in detail.

- **Before the necessity sequence**. It reports an example of a programming task with which students have become familiar in developing mastery of the previous concept. This task is the last before the P!S phase and is structurally and linguistically very similar to the *necessity* problem of the current sequence (see why in 9.2.1).
- **P!S – Problem-solving phase (unsolvable problem)**.
  - *Problem*. The text (as proposed to students) of the *necessity* problem (see 9.2.2), a programming task that requires the target concept in order to be solved. It may consist of a sequence of tasks.
  - *Necessity trigger(s)*. The obstacles students encounter when trying to solve the problem without the target concept. Not being able to overcome these obstacles should stimulate in them the feeling of the necessity of the target concept.

  - **Necessity**. The core mechanism of our learning design describes the feeling of the necessity of the target concept. That is, what students experience by not being able to solve the proposed problem.

  - *Sub-optimal solution(s)*. The solutions students might develop in trying to solve the problem without the target concept. A student solution in this phase is usually incomplete; however, it also may be complete (i.e., a solution that works for all cases but is still sub-optimal for other reasons; see 9.2.2.3). The solutions reported do not necessarily have to emerge during the sequence implementation, nor is it required that all students can formulate them.

- **I – Instruction phase**. The target concept, and its related knowledge, that the teacher directly instructs students (but without applying it to solve the necessity problem).

- **PS – Second problem-solving phase**. It reports the optimal target code students should be able to develop by engaging again in the necessity problem after being instructed on the target concept.

After the second dashed line, the third and last part reports some characteristics of the example and, when present, implementation warnings.

- **Characteristics**.

  - *Abstraction mechanism*. It indicates whether the sequence is about *control* abstraction or *data* abstraction (see 2.3.6).

  - *Abstraction movement*. It indicates whether the target concept represents an *increase* or *decrease* in the abstraction level (compared to what students can already do with the tools they know so far).

  - *Educational hardness*. It indicates whether the sequence stimulates an educationally *hard* or *soft* necessity; it depends on whether or not it is possible to develop complete solutions without the target concept (see 9.2.2.3).

- **Warning(s)** (optional). It lists, if any, the knowledge that would enable students to solve the problem easily. It is crucial to be aware of such knowledge: if students have it, they will not experience the necessity of the target concept.

Each example is followed by a brief final discussion, which reports relevant considerations on the *necessity* sequence presented.

9.3.4.1 The necessity of definite iteration

Necessity example 1

The necessity of definite iteration (with turtle geometry)

Problem in a nutshell

Drawing a polygon with a given (high) number of sides. Then, changing the program to modify the side length. Finally, changing the program to modify the number of sides.

What students already know

- *Syntactic level + viable conceptual model*
  - Importing functions from libraries
  - Basics of the `turtle` module:
    - `forward` (`backward`) function
    - `left` (`right`) function
- *What students already know how to do (strategic level)*
  - Using `forward` (`backward`), `left` (`right`) functions from `turtle` module to draw contiguous segments

Target concept

- `repeat N` loop, with `N` being a fixed integer known at compile time (syntactic + conceptual)
- repeat instructions a fixed, pre-determined number of times (strategic)

Before the necessity sequence

Example of a previous task not needing the target concept Draw a square of side 50.

continued on next page

P!S – Problem-solving phase (unsolvable problem)

Problem The problem consists of three sequential tasks. The next task is given when students have finished the current one.

A. Draw a 20-side regular polygon of side 50.

B. Edit the previous program so that it now draws a polygon (still 20-side) with sides length equal to 45.

C. Edit the previous program so that it now draws a 18-side polygon (with sides length equal to 45).

Necessity triggers

A. Students need to repeat 20 times a two-instruction block.

```
forward(50)
left(18)
```

B. Students need to update 20 times the side length.

```
forward(45)
left(18)
```

C. Students need to remove two blocks and update 18 times the angle.

```
forward(45)
left(20)
```

Necessity I need a way to repeat a block of code a fixed number of times.

Sub-optimal solution

- (complete) The straightforward solutions with the replicated code are completely correct, but tiring and likely subject to errors or oversights.

I – Instruction phase

Illustrating how to *repeat N times* a block of instructions, with *N* being a fixed integer value, realised in Python.

```
for i in range(N):
    <block of code>
```

continued on next page

PS – Second problem-solving phase

Target code A possibile solution to task C (solutions to taks A and B can now be easily obtained by modifying this code as well).

```
from turtle import forward, left
for i in range(18):
    forward(45)
    left(20)
```

Characteristics

- *Control* abstraction
- *Upward* abstraction movement (compared to replicating the same block of instructions many times)
- *Soft* necessity (complete, though sub-optimal, solutions are possible without the target concept)

Implementation warning

Other concepts that students should not know yet for the mechanism to work:

- Variables and assignments

Example discussion Task A would be sufficient to introduce the necessity of definite iteration. However, we believe that tasks B and C can reinforce this necessity. Indeed, the request for multiple changes aims to provoke a little frustration in students so they can feel the necessity of a repetition construct, especially in a scenario of soft necessity like this (see 9.2.2.3).

Note that this would not be the case if variables and assignments were already known to students. In that scenario, tasks B and C are not useful. Therefore, task A should be designed in such a way as to stimulate the necessity of definite iteration even more (e.g., asking for more polygons, for polygons with more sides). Otherwise, simply using two variables (i.e., one for the side length and one for the number of sides), the updates required by tasks B and C would be trivial and fast.

9.3.4.2 The necessity of indefinite iteration

Necessity example 2

The necessity of indefinite iteration

Problem in a nutshell

Counting how many random numbers are generated before getting a certain value.

What students already know

- *Syntactic level + viable conceptual model*
  - Importing functions from libraries
  - Variables and assignments
  - Boolean expressions
  - `if` statement
  - `for` loop with explicit, automatically handled index ("true" definite iteration; e.g., Python `for i in range(1,11)`, Snap! (and Pascal) `for i :=1 to 10`, but not the 3-clause `for` of C and Java)
  - Function to generate a random integer in an interval
- *What students already know how to do (strategic level)*
  - Use a variable as a counter
  - Combine definite iteration with selection to iterate over a collection and perform an action on the basis of a condition

Target concept

- `while` loop (syntactic + conceptual)
- Repeating until a condition is met (without knowing how many iterations will take) (strategic)

Before the necessity sequence

Example of a previous task not needing the target concept Count how many times a program that generates K pseudo-random integers between 1 and 1000 produces the number 42.

continued on next page

P!S – Problem-solving phase (unsolvable problem)

Problem Count how many pseudo-random integers between 1 and 1000 a program generates before getting the number 42.

Necessity trigger Students do not know a-priori how many calls the program needs to get 42.

Necessity I need a way to repeat "until something happens" without knowing in advance when (or even if) it will happen.

Sub-optimal solutions

- (incomplete) Using a definite iteration to repeat a very high number of times (ideally to the MAXINT, if available in the language), hoping that 42 will be generated before.
- (complete) Using the (often not simple nor elegant) possibility of modern languages' `for`-like loops to realise an indefinite iteration.

I – Instruction phase

Illustrating the concept of indefinite repetition using the construct that repeats a block of code "while a condition ¡C¿ is True" realised in Python.

```
while <C>:
    <block of code>
```

PS – Second problem-solving phase

Target code

```
from random import randint
count = 1
while randint(1,1000) != 42
    count = count + 1
print(count)
```

Characteristics

- *Control* abstraction
- *Downward* abstraction movement (compared to the for loop with explicit, automatically handled index)

continued on next page

- *Hard* necessity (without the target concept, using (true) definite iteration, only incomplete solutions are possible; complete solutions are possible by using—often inelegantly hence sub-optimally—`for`-like loops to realise indefinite iteration)

Example discussion The necessity that this sequence aims to stimulate would also be satisfied by using the recursive mechanism. Consequently, the same sequence can be used to introduce recursion.

Another essential clarification concerns a limitation of this *necessity* sequence. It can only be used with languages (such as Scratch and Python) in which the `for` loop realises a "true" definite iteration[11]. For obvious reasons, it cannot be used with languages in which the `for` construct (such as the 3-clause `for` of C and Java) can express indefinite iterations and thus be used instead of the `while` loop.

An alternative but equally viable scenario to stimulate this necessity is the well-known *Rainfall problem* [Soloway, 1986]. The condition on which the program waits is not related to the extraction of a pseudo-random number but to the user's input of a termination value. Only a few prerequisites would change: students need to know how to use basic input functionalities, and they do not need to know how to import modules or generate pseudo-random integers.

Another "less straightforward" way to stimulate the necessity of indefinite iteration is asking to terminate the scan before the end of the sequence (without using `break`, `return`, or auxiliary functions) when a specific condition is satisfied. For example, the problem might ask students to determine whether a very long sequence contains at least one particular integer and to do so in the shortest possible time. Consequently, since the sequence is very long, it is critical to stop as soon as the number is found. We consider the latter a more artificial request and, consequently, less effective in stimulating the necessity of indefinite iteration: this is not a task that can only be solved with the target concept but rather a task that can be solved more efficiently with it.

9.3.4.3 The necessity of arrays

Necessity example 3

The necessity of arrays

Problem in a nutshell

Keeping track of how many times each number is drawn, with many possible numbers and many extractions.

continued on next page

[11] Please, recall the discussion about this matter in 9.2.2.3.

What students already know

- *Syntactic level + viable conceptual model*
  - Variables and assignments
  - Boolean expressions
  - `for` statement
  - `if` statement
  - Function to generate a random integer in an interval
  - Function to plot a bar chart
- *What students already know how to do (strategic level)*
  - Using a variable as a counter

Target concept

- Arrays/lists with index access (syntactic + conceptual)
- Using arrays/lists to store, read and modify values accessing them by index (strategic)

Before the necessity sequence

Examples of previous tasks not needing the target concept

A. Given the possibility of tossing a coin (i.e., generating a random integer between 0 and 1), check whether the number of heads and tails is approximately equal after a high number of tosses (e.g., 10000).

B. Given the possibility of throwing a die (i.e., generating a random integer between 0 and 5), check whether the results of the throws are evenly distributed after a very high number of throws (e.g., 1 million).

P!S – Problem-solving phase (unsolvable problem)

Problem Given a simplified version of the Bingo game (i.e., the possibility of drawing a number between 0 and 89), check whether the distribution of the results of the extractions is uniformly distributed through a very high number of extractions (e.g., 1 million).

continued on next page

Necessity trigger Students need to create an unnaturally high number of unrelated variables and handle them with a very long sequence of selection statements, resulting in a program that is hard to manage without errors.

Necessity I need a data structure to collect the frequency of extraction of each number and access (and modify) the values in that structure in a programmatic way based on runtime informations (i.e., the current number extracted).

Sub-optimal solution

- (complete) Create a variable for each number (90 variables) and, after every extraction, increment the corresponding variable through a multiple selection statement.

I – Instruction phase

Illustrating:

- how to create a (fixed length) ordered structure (array/list) and initialize it with constant values;
- how to access elements of the structure by integer index;
- how to modify an element by using index access on the left hand side of the assignment.

PS – Second problem-solving phase

Target code An example, using Python lists as arrays, i.e., with fixed length.

```
from matplotlib.pyplot import *
from random import randint
L=[0]*90
for k in range(10**6):
    L[randint(0,89)] += 1

bar(range(90),L)
show()
```

Characteristics

- *Data* abstraction[a]
- *Upward* abstraction movement (compared to using many unrelated variables)

continued on next page

- *Soft* necessity (complete, though sub-optimal, solutions are possible without the target concept)

Implementation warning

Other concepts that students should not know yet for the mechanism to work:

- Dictionary-like data structures

[a]This data abstraction allows also for a powerful control abstraction. See the related discussion after the example.

Example discussion This sequence concerns *data* abstraction because a set of unrelated variables is gathered in a single data structure. However, accessing by index to the elements of this structure allows a powerful *control* abstraction since it makes a long cascade of `if` statements (previously used to decide which variable to modify) disappear in one fell swoop.

Moreover, although the example uses Python's lists (i.e., dynamically extendable structures), the core necessity here is accessing by index. Hence, this sequence is more suitable for use with languages that provide array-like structures.

Lastly, this example scenario could also be used to introduce dictionaries instead of arrays. However, we present below example 4 that we consider more effective for introducing dictionaries since its *necessity* problem requires the use of non-numeric indexes.

9.3.4.4 The necessity of dictionaries

Necessity example 4

The necessity of dictionaries

Problem in a nutshell

Counting the frequency of each character in an input string.

What students already know

- *Syntactic level + viable conceptual model*
  - Variables and assignments
  - `if` statement
  - Taking string input
  - Type conversion from `char` to `int`

continued on next page

  - `foreach` over a string
  - Arrays/lists with index access
- *What students already know how to do (strategic level)*
  - Check if a character is in a string
  - Use a variable as a counter
  - Iterating over values of a sequence
  - Combine definite iteration with selection to iterate over a collection and perform an action on the basis of a condition
  - Using arrays/lists to store, read and modify values accessing them by index

Target concept

- Dictionaries (syntactic + conceptual)
- Using dictionaries to store, read and modify values accessing them by key (strategic)

Before the necessity sequence

Example of a previous task not needing the target concept Given a string taken from user input, determine the frequency of each of the ten digits in the string.

P!S – Problem-solving phase (unsolvable problem)

Problem Given a string taken from user input in an unknown (potentially very long) alphabet, determine the frequency of each character in the string.

Necessity trigger Students need to keep correspondence between each character and its frequency. However, arrays allow accessing values only by integer indexes.

Necessity I need a way to associate a non-integer element (i.e., a character) with its frequency and use that element as a *key* to access and modify such frequency.

Sub-optimal solution (complete) Creating two "parallel" arrays/lists. The first one stores each character encountered, and the second one stores the frequency of that character in the corresponding position.
Note that a way to dynamically increase the structures' length is needed because it is unknown beforehand how many different characters are in the string.
Note also that looking up for characters (and their corresponding integer index) in the first array/list introduces a high computational overhead[a].

continued on next page

I – Instruction phase

Illustrating:

- how to create a dictionary[b] data structure that keeps a collection of (key,value) pairs, i.e., a correspondence between unique keys and arbitrary values;
- check if a key is already associated with a value in a dictionary;
- how to access dictionary elements by key;
- how to modify a dictionary element by using key access on the left-hand side of the assignment.

PS – Second problem-solving phase

Target code We propose a solution using Python dictionaries.

```
text = input()
freq = {}
for c in text:
    if c not in freq:
        freq[c] = 1
    else:
        freq[c] += 1
```

Characteristics

- *Data* abstraction
- *Upward* abstraction movement (compared to using structures with only integer index access)
- *Hard* necessity (although complete but still heavily sub-optimal solutions are possible without the target concept, they require advanced concepts such as parallel arrays and dynamically increasing structures)

[a] For each character c of the input string, $O(\texttt{len(L\_CAR)})$ to determine if c is already present in `L_CAR` and, if present, to determine its integer index.

[b] Often called *associative array*.

Example discussion Although the sub-optimal solution presented (using parallel lists) is complete, we still think this sequence stimulates an *educationally hard necessity*. Indeed, if students are unaware of the parallel arrays/lists pattern, they are unlikely to conceive the

solution strategy reported as a possible sub-optimal (yet complete) solution.

Furthermore, we believe it is best if students are not exposed to the use of arrays/parallel lists (as a way to keep correspondence between different sets of elements) to increase the efficacy of this *necessity* sequence. Because this programming pattern allows the problem to be solved completely and quite easily (though less straightforwardly than using dictionaries), the necessity perceived by students would be much weaker. Besides, it would become a *soft necessity* scenario (since students could resort to the tools they already know, see 9.2.2.3). In that case, the necessity feeling would concern not the possibility of solving the problem but just the efficiency of the solution (observable and significant only with very long inputs) and the conciseness of the code.

This observation is worth making because programming courses traditionally teach this pattern. However, in languages that include more advanced data abstractions (such as dictionaries or OOP), we think it is not advisable to teach it, as it is less efficient, concise and elegant than its more specialized and modern alternatives.

9.4 Conclusions

Two are the main contributions of this work.

1. The proposal of *Necessity Learning Design* (NLD), in particular:

   - The definition of the *necessity mechanism*, observed in our experience of teaching programming and inspired by some science education literature, but formalized in an original way; the mechanism was conceived for teaching programming but turned out to be more general.
   - The development and analysis of *Necessity Learning Design*, a learning design specific to introductory/CS1 programming that leverages the necessity mechanism; NLD is designed to support the introduction of the core programming concepts.

   Necessity Learning Design requires students to engage with programming tasks very similar to those they already successfully faced, but this time they miss an essential ingredient (the target concept). Hence, struggling without success to solve the task, they will experience the *necessity* of that concept.

   This is how we try to answer our research question: *what kind of learning design can support novice students when introducing a new programming concept?*

2. A series of *necessity* sequences – framed in a concrete CS1 path – as examples of NLD use at learning moments when abstraction changes because of the introduction of a new programming concept.

   These moments are some of the ups and downs of what we call the *rollercoaster of abstraction*. Research shows that moving between the different abstraction levels of the constructs of a programming language is difficult for novices, both going upward and downward (albeit for different reasons).

Abstraction from the learners' perspective is a lens through which we analyzed the introduction of new programming constructs. Ultimately, this lens helps in the design of *necessity* sequences.

Necessity Learning Design for introductory/CS1 programming From an educational point of view, our design is inspired by PS-I approaches and, in particular, by Productive Failure learning design. However, NLD is *domain-specific* [Nelson and Ko, 2018] since it leverages inherent aspects of programming problems, like the interactivity of a program and the possibility of having objective feedback from the machine to check whether the problem is solved or not. Therefore, we developed a three-phase approach synthetically described as P!S-I-PS.

i. In the **P!S** phase, students are not able to solve (or optimally solve) a given programming problem, experiencing the *necessity* of the target concept.

ii. In the **I** phase, unlike in Productive Failure, students are not given the solution. The target concept and its general usage are directly taught.

iii. In the final **PS** phase, students go back to the problem with the necessary knowledge to solve it, building on their previous failed attempts.

Moreover, NLD approach seems more generally in line with the vast body of research on CS1 courses for various reasons.

- It falls within the domain of active learning approaches (see 2.4), as students are actively involved in solving a problem and return to it, compelled to reflect on their previous attempt.

- It can help reduce cognitive load (see 1.4.1) because it allows for a very gradual path, in which a new concept is proposed as a *minimal addition* to students' prior knowledge (see 9.2.1) and introduced in an isolated situation.

- Necessity problems (i.e., programming tasks) are carefully designed to capture the essence of the target concept; they are meaningful and prototypical examples of the use of that concept.

- Problems are constructed so that the target concept is (at least at that specific point in the learning path) the optimal one to use to solve that problem. Therefore, NLD helps students not only focus on syntactical and conceptual knowledge but is especially useful in fostering strategic knowledge. Indeed, a *necessity* sequence puts students immediately in a situation where the target concept is essential to satisfy the problem's request.

- The examples of *necessity* sequences we presented are designed to support students when a new concept is introduced; therefore, they stand at the edges of core programming concepts.

This choice aligns with the *Learning Edge Momentum* hypothesis (see 1.4): teachers and educators should pay particular attention when introducing the first programming concepts (and their connections). By supporting students to understand those core concepts in the earliest stage, Necessity Learning Design can prevent building negative momentum from the start. Indeed, we believe our learning design can benefit particularly those students who cannot keep up with the pace of introductory programming courses (such as CS1) from the early steps.

NLD examples in the abstraction rollercoaster The necessity sequences we proposed (as examples of NLD use in a CS1 learning path) support learning moments in which the abstraction level changes because of the introduction of the target concept. Programming education literature acknowledges that these moments are critical for learners. Moreover, we recognize an "abstraction rollercoaster" in a typical CS1 learning path because the abstraction level goes up and down (more than once and in many different possible orders) within the same programming language (whatever it is) chosen for the course. As teachers and educators with years of experience teaching introductory programming, recognizing that both directions are challenging for novices was a little surprise. Therefore, all the abstraction movements in a programming learning path require instructors' awareness because ups and downs demand different kinds of attention and present different scenarios for the use of Necessity Learning Design.

- Going down in abstraction is challenging – and it is easy to understand why – because it adds details (which must also be handled correctly), thus increasing learners' cognitive load. Usually, to stimulate the right necessity in students, they have to be put in a situation where the extra details are necessary to solve the problem (*educationally hard necessity*).

- Going up is also (differently) challenging because details that students have learned to control are taken away. Instructors may think this is easy because, from their perspective, the new construct is more convenient. Instead, they need to convince students that the new construct makes their life easier. A well-designed *necessity* problem can help exemplify how the new construct is either simpler to use, efficient, elegant, expressive, or a combination of these (*educationally soft necessity*).

While acknowledging that there is no agreement among researchers and educators on the right path to follow in introductory programming courses such as CS1 (see 2.1), we presented a concrete learning path (based on the CS1 for math we successfully tested for ten years, also online). Along that path, we placed the four *necessity* sequences we developed as examples of NLD use. As extensively discussed, the learning path (particularly the order of topics) determines the abstraction movements within it and thus influences the choice and design of the *necessity* sequences.

9.4.1 Limitations

As already pointed out and discussed, Necessity Learning Design is designed to support students only *in introductions* of new concepts (and it is particularly suitable when the new concept/construct determines a change of abstraction).

Two other significant limitations are related more to the necessity mechanism than the learning design.

First, none of the students of a class group must know the target concept. If even one student knew it, she would not experience the necessity feeling in the first stage (P!S) and, by simply communicating with classmates, would make the *necessity* sequence worthless for her entire class. This condition (i.e., that none of the students knows the target concept) may not occur as often as instructors deem using NLD appropriate. Moreover, establishing such a condition could not be straightforward, particularly without stimulating students' curiosity.

The other limitation concerns students' perceptions. If instructors leverage the necessity mechanism too often, two scenarios (which unfortunately are not mutually exclusive) may occur as a negative consequence. Students might "bite the bullet". Having already experienced at least a necessity sequence, they might remain inactive in the first phase (P!S), passively waiting for the instruction phase to have the target concept at their disposal. In this scenario, despite instructors' efforts to implement the sequence, there is no benefit to motivation. In the other negative scenario, students might feel "cheated" by their instructors, thus contesting the method altogether. In any case, there would be no learning benefits but, on the contrary, potential relational and trust issues to manage.

This latter limitation, particularly its two possible (and not mutually exclusive) negative consequences, invites sparing use of Necessity Learning Design. Therefore, we remark (again in line with the LEM hypothesis) to use NLD particularly early in the learning path (when students are most likely not to know the potential target concepts) and only to introduce new core concepts. In the same vein, we invite using NLD when the target concept requires students to move (either up or down) in abstraction so as to "spend" NLD use at the most critical stage of the learning path.

9.4.2 Accidents on the road

Since we started from the observation and analysis of the (later called) *necessity mechanism* in our teaching situations, we did not know what the exact outcome of our research would be. However, it was clear to us from the beginning that this learning mechanism could be leveraged to support novices in learning introductory programming (being aware, also from IEdR literature, of the difficulties of introductory programming and access to informatics; see 1).

On the one hand, we did not expect the necessity mechanism to be more general than just programming (9.2.1). Most notably, even when the development of Necessity Learning Design was in an advanced stage, we had yet to realize that its use was precisely for introducing concepts and not teaching them in general.

An in-depth analysis of abstraction – prompted by a publication opportunity and not initially planned in the research project – allowed us to recognize the difficulty of abstraction

movements (even when abstraction increases) from the learners' perspective. This insight led us to realize that NLD's potential is in supporting the *introduction* of core programming concepts (and that it is especially beneficial when abstraction changes). This awareness allowed us to define our research question more precisely and better connect our research to the literature on introductory programming, particularly the LEM hypothesis (1.4) and its implication on cognitive load (1.4.1).

Finally, in the original project (as anticipated in the earlier research work described in chapter 8), there was the idea of including notional machines as part of our learning design in an attempt to develop a theoretical model expressed in terms of progressions of notional machines (educational devices) rather than language constructs (concrete artifacts).

Finally, this research project also had the ambition to consider notional machines (as seen in the preliminary work described in chapter rifnece0), including them as an integral element of our learning design. The more general and ambitious goal was to develop (alongside the development of the learning design) a theoretical model for teaching introductory programming, defined in terms of progressions of notional machines (educational devices; see 2.3.4) rather than programming constructs (concrete artifacts).

This idea, which also arose from the example of learning languages (see 2.7), still remains to be investigated and transformed into a more concrete proposal than one reported in 8. Indeed, our research took a more applicative turn and focused on developing the learning design along with concrete examples and general indications on how to use it. Our spirit as teachers and educators emerged in wanting to create something immediately helpful to mitigate the problem of teaching programming to novices. Therefore, it is not surprising that the following research work was designing a school implementation and experimentation (see the following 9.4.3 and particularly chapter 10).

9.4.3 Future works

Our proposal is built on education research literature (mostly from science education), and we believe it is a sound *necessity-driven* learning design. We informally experimented with it in several editions of a CS1 for math majors (at the undergraduate level) with good results and positive feedback from the students.

However, we are aware that Necessity Learning Design must be tested in more rigorous ways and more controlled environments. We need to gather quantitative and qualitative data to evaluate its impact on the different types of programming knowledge, learning edge momentum, cognitive load, and students' perception. Ultimately, we need such data to evaluate its effectiveness in supporting novices' learning when introducing a new programming concept. The following chapter (10) details the design of an NLD experiment in a real school setting, its implementation, and reports a preliminary analysis of our observations (as external researchers) and the information collected from the students and teachers involved.

Besides that, we welcome teachers and educators to let us know examples of any *necessity* situations they recognise in their teaching (better if contextualised in a learning path to identify the direction of the abstraction movement). Such information can help us build other relevant *necessity* sequences to cover more and more core programming concepts in different paths and thus continue refining our learning design. We also welcome anyone willing to try

any of the *necessity* sequences we proposed, and we are happy to provide any clarification and support.

Chapter 10

Necessity School Experimentation

We knew that a necessary step was to evaluate the effectiveness of *Necessity Learning Design* (see chapter 9) in an authentic learning context. We were aware that an initial trial, however challenging to design and implement, would not bring definitive results on the effectiveness of our learning design. However, it would provide valuable insights into possible improvements and limitations, highlighting design flaws or neglected aspects. With this in mind, we sought to measure the learning assessment dimension more quantitatively and the student motivation and student and teacher perceptions in a mixed (both quantitative and qualitative) way. Therefore, we organized an experiment in a local high school, choosing among the few school strands where informatics and, in particular, programming are taught. We used NLD to introduce arrays as a data structure in C++ (unfortunately, among the languages most taught in Italian schools) and, more generally, an essential concept of introductory programming. A first analysis of the collected data gave us a substantially positive picture of the use of NLD to support the introduction of a new concept while also confirming the limitations already expected at the design stage, highlighting some errors in the development of the sequence of necessity investigated.

This chapter reports on the first school-based experimentation of NLD in a local high school. The experimentation is described by following its phases. First, there was a **preliminary design** phase (described in 10.1.2) that involved only us researchers and mainly set the objectives and framed the general ways of experimentation. Then the **concrete design** was defined with the teacher involved in the experimentation. The specific choices made are described and discussed in 10.1.3. Section 10.2 reports everything significant that emerged during the **implementation** of the experimentation in the experimental class and control classes. Finally, section 10.3 discusses a **preliminary analysis** of the data collected.

The most relevant material built and used for the experimentation is attached in its original form (in Italian, untranslated) in appendix 16.2.

10.1 Experimentation design

10.1.1 Non-interference principle

Before moving on to a detailed description of the various design phases of the experimentation, we want to make explicit a principle that guided the entire design (and the implementation, too). We tried to disrupt the school context as little as possible so that the only new element was the use of Necessity Learning Design. This resolution concerned, for example, deciding to interfere as little as possible with the modes of instruction, exercise and verification (including materials) routinely used by the teacher involved in the experiment. At the same time, this general principle advised us to limit the number and duration of self-perception questionnaires administered to students. Also, with the same intention, we decided not to play an active role during the implementation phases of the experimentation. In other words, we decided not to support the students nor advise them during the steps of the *necessity sequence* (see 9.2.2.1), which would have been natural given our experience of teaching informatics and programming, also in high school. On the contrary, we tried to train the teacher beforehand and offer him constant support, especially outside of school activities, for any requests or doubts (while trying not to change his habits and strategies). Thus, during the activities, we would just observe (keeping track of the observations in a private journal), trying to make our presence as unobtrusive as possible so that the activities could unfold as naturally and similarly as usual.

As far as possible, we wanted the use of NLD to be the only difference from the learning contexts and processes to which the classes involved in the experimentation were accustomed. In the following sections, we describe all design choices, and when they were motivated (also) by this principle, we highlight it.

10.1.2 Preliminary design

10.1.2.1 What to measure

Before actually organizing the experimentation, we defined what we wanted (and could) measure relative to the use of Necessity Learning Design in a concrete learning context. Although we did not yet refer to a specific context, we already tried at this stage to take into account the typical limitations we would encounter in an Italian high school. This context is particularly familiar to the author of this thesis, as he has been teaching informatics in high school since 2013 and holding a chair since 2016. For the quantitative dimension, we wanted to consider the learning assessment grades for each student and the whole class to register a possible discontinuity introduced by using NLD. Specifically, we intended to compare the average of the grades up to the time of the intervention with NLD with the grade of the first assessment test on the topic introduced with NLD. Also, from a quantitative point of view, we wanted to analyze students' self-perceptions on learning arrays (and informatics programming in general) through purpose-built pre and post questionnaires mainly realized by Likert-scales. Similarly, we wanted to evaluate possible changes in students' motivation. Qualitatively, we wanted to analyze students' dispositions and impressions, and teachers'

opinions, through open-ended questionnaires, field observations conducted by us researchers, and the writing and analysis of our private journal of the experimentation.

10.1.2.2 How to evaluate NLD efficacy in learning

We thought of proposing a necessity sequence of necessities to introduce a new programming construct/concept to a third-grade class of an "Istituto Tecnico Tecnologico" (i.e. the *technological* track of the technical strand of Italian upper-secondary school), following the informatic sub-track. In this type of school (one of the very few where informatics is taught), students (about 15-16 years old) start programming in the third grade. Thus, these are novice students who face an introductory programming learning path. According to the national guidelines for technical high schools and specifically for the technological-informatics track, this path leads students to master programming with at least one imperative language, including the object-oriented paradigm. Within three years (i.e., the end of high school), they should know and use the most relevant software engineering techniques for software modularity and reusability. In the meantime, they are supposed to have learned the main informatics concepts and ways of thinking. In this context, we thought of organizing a quasi-experiment. We would involve two third-grade classes, one using NLD and the other acting as a control group, where the programming concept being experimented on was introduced in a "traditional" way (we clarify later what is meant by traditional). The quasi-experiment, that is, having an experimental group and a control group (i.e., the two classes) but without being able to assign the participating students to these two groups freely[1], is an unavoidable necessity dictated by the actual school context. Indeed, interrupting normal classroom activities to organize proper experimental groups is impossible. It is already complicated enough to organize experimentation involving teachers and students – who are already amply engaged and stressed by the regular school routine. However, we tried to minimize the uncontrollable variables given these institutional limitations. We thus involved two third-grade classes at the same point in the programming learning path. Another critical point, to have the two student groups comparable as much as possible, was the intention to involve two classes with the same informatics teacher. As anticipated, we planned to propose our intervention in the first part of the school year, when the students are still basically programming novices (and thus have the characteristics common to most of those undertaking introductory programming paths). At the same time, we wanted to avoid NLD being the first mode students encountered to learn a new programming concept. First, NLD does not allow it: the *necessity mechanism* requires building mastery over preparatory programming exercises (see 9.2.1). Second, we wanted to give students a chance to express an opinion about the mode they had previously experienced (thus being able to compare it with NLD). Also, from a quantitative point of view, we needed a history (albeit short) of previous grades to compare against the learning assessment results for the topic introduced with NLD. In particular, we thought that two classes (the experimental class receiving the mode with NLD and the control class), albeit in a very similar situation, are difficult to compare, as each class often has very different starting

[1] Complying with the characteristic of the randomness of the sample is one of the necessary characteristics of a "real" *experiment*.

characteristics and educational history. In other words, we did not deem it possible to directly compare the experimental class's grades with the control class's. Instead, in each of the two classes, we would compare (both at the class level and for each student) the assessment grade for the experimentation concept (i.e., the one introduced with NLD in the experimental class) with the average of the previous grades. The *delta* (i.e., the difference) between these two measures would be nonzero in case NLD introduced a discontinuity (positive or negative) from the previous mode. This *delta* is comparable between the experimental and control class (both at the class and student levels). In addition, the presence of the control class makes it possible to factor out the impact of the new topic on the *delta*. That said, there are a great many other factors that could affect this measure (starting with the bio-psycho-social status of individual students and also of the teacher, the general mood of the class, and many others), as indeed happens in most social experiments organized in real-world settings. We cannot eliminate them or control them completely. There are indications and methodologies from statistics that help to consider and partially control these other factors. However, a more in-depth analysis in this regard will take place at a later stage of the research project, that of rigorous analysis of the data from NLD school experimentation. For now, the choices already described (such as choosing two classes with the same teacher and at the same point in the learning path, and also using a control group to compare the delta of the experimental class with a baseline) and, in general, the choices made in defining the experimental protocol (including those that concerned its actual implementation) were always aimed at making the experimentation most rigorous and controllable as possible.

10.1.2.3 How to evaluate NLD effects on students

We thought of using a mix of quantitative and qualitative measurements to evaluate the impact of NLD on students, particularly from their perceptions about the experimented topic and, more generally, about informatics programming. The perceptions we sought to capture were related to both auto-assessment of learning and various dimensions of students' experience while learning programming.

Qualitative methods . Educational research suggests resorting to qualitative measures, especially at the beginning of a research project, because they are more open-ended and, therefore, suitable for detecting elements not anticipated by researchers (manca una REF). At the same time, qualitative data can return a rich and informative picture, perhaps not generalizable, but helpful in understanding and possibly evolving a new "object" (NLD in our case) being experimented on.

Among the various tools made available by educational research, we decided to use non-participatory observations to capture the qualitative perspective. We would have been present as researchers during all phases of the experimentation implementation. Our role as NLD developers would have been unknown to students, as would our experience as informatics and programming teachers. This choice reflects the general approach already presented in 10.1.1, namely, the intention to disrupt the observed contexts as little as possible so that, in our case, the principal element of discontinuity was precisely using NLD to introduce a

new programming concept. Students obviously would have been informed in advance by the teacher of our presence as researchers for a generic university project to survey teaching in local schools and our also of "silent" role as observers of their habitual educational contexts and processes. Our observations would have been recorded in a private researchers' journal. The journal's analysis, daily during the implementation and more rigorously at the end of the experimentation, could provide that open and rich information aforementioned. Such analysis could help understand NLD better and improve it, from a foundational point of view, and for its applications and limitations.

In addition, again on the qualitative front of student perceptions, we wanted students to express their opinion and general state during the experimentation's crucial phases (i.e., the phases of and around a *necessity sequence*). We wanted this chance to be perceived by them in a light and accessible way, not as a burdensome task. Thus we decided to develop a very simple in-progress questionnaire with two open-ended questions, openly emphasizing the possibility of answering even with a single word or emoji. The questionnaire includes one question about the activity ("What do you think about the activity you have just experienced?") and one about their state ("How do you feel after the activity?"). Students are supposed to answer these two questions after every crucial activity. The questionnaire is available (untranslated, in its original form) in appendix A.4.1.1.

In addition, and only for the experimental class, we thought of asking some specific questions about the use of NLD at the end of the experiment. These are only four open-ended questions, mandatory but without minimum (nor maximum) character limit so that students can feel free to express their opinions but also to answer very briefly or even not at all. Indeed, a no-answer or a very short one is still informative. As can be seen, the questions are deliberately general and wide-ranging. They are consistent with the intent to qualitatively analyze student responses, with the broad goal of a better understanding of NLD and its use in real-world contexts. Below, the reader will notice a slight straining to the structure of this chapter, making it easy to read these questions. We anticipate the programming concept being experimented on, whose choice is described only in the following subsection (reporting the concrete design of the experimentation), specifically in 10.1.3.2. Arrays are the *target concept* of our experimentation; thus we refer to the *necessity sequence* to introduce arrays, detailed in 3.

1. *What aspects did you like about the way arrays were introduced (having you try to solve the lotto problem before explaining arrays)?*

2. *On the other hand, what aspects did you not like?*

3. *How did you feel during the time when you did not know how to solve the lotto problem (because you did not yet know arrays)?*

4. *Did it help you to learn arrays, or did it hinder you? Why?*

These question can be consulted (untranslated, in their original form) at the end of the post-experimentation questionnaire for the experimental class available in appendix A.4.2.1.

Quantitative measures. We thought of using a quantitative approach to get a more precise estimate of student motivation before and after using NLD. We looked in the education research literature for questionnaires, possibly validated, that would allow us to measure student motivation. We did not find validated questionnaires suitable to assess student motivation and, more generally, perceptions in our specific scenario.

However, we found a tool that came close to our needs. It is a series of questionnaires consisting of Likert scales, therefore fit for quantitative analyses. It is not a validated tool but has been extensively motivated and discussed by Kalish [2009] and tested in some real-world scenarios. The original objective of these questionnaires was to assess the effects of instructor immediacy and student need for cognition on student motivation and perceptions of learning. Of the three questionnaires used by Kalish [2009], we selected two to be used in combination with each other, specifically the *State Motivation Scale* (SMS) e the *Cognitive Learning/Learning Loss Measure* (CL). The SMS was developed (based on work by Beatty and Payne [1985]) and used by Richmond [1990] and also used by Christophel [1990] to measure both student state and trait motivation. The SMS version we used was slightly adapted by Kalish [2009] by removing the last five prompts (deemed to be repetitive). It consists of twelve 7-point Likert scales where the prompt is rephrased twice (positively and negatively) for the same questions. Kalish used the SMS in combination with the questionnaire for *Cognitive Learning/Learning Loss Measure*, which consists of two questions structured as 10-point Likert scales. Richmond et al. [1987] developed the *Learning Loss scale*, and Kalish used it to collect student reports on the cognitive learning amount they engaged in. Indeed, according to McCroskey et al. [1996] and Lang [2007], one of the better and accepted methods to identify student learning is using student self-report measures. The *Learning Loss* is calculated by subtracting the score of the first question from the score of the second question. *Cognitive learning* is then evaluated by reversing the *Learning Loss* score.

We thought that by adapting the questions for *Cognitive Learning/Learning Loss Measure*, we could use the questionnaire to assess the specific impact on the *State Motivation Scale* not of the instructor (as in Kalish [2009]), but of the use of NLD. Since we intended to use this tool both at the beginning of the experimentation and at the end, we readjusted the questionnaire for *Cognitive Learning/Learning Loss Measure* twice. Below, the reader will notice a slight straining to the structure of this chapter, allowing us to clarify better how this evaluation tool was built. We anticipate the programming concept being experimented on, whose choice is described only in the following subsection (reporting the concrete design of the experimentation), specifically in 10.1.3.2. Arrays are the *target concept* of our experimentation; thus, we refer to the *necessity sequence* to introduce arrays, detailed in 3.

The original questions for the *Cognitive Learning/Learning Loss Measure* are the following. It can be seen how the second question aims to evaluate the impact of the instructor on student learnings.

1. *On a scale of 1–10, how much are you learning in the course immediately preceding this course, with 1 meaning you learned nothing and 10 meaning you learned more than in any other class you've had?*

2. *On a scale of 1–10, how much do you think you could have learned in the course*

immediately preceding this course had you had the ideal instructor?

Our first adaptation, to be administered at the beginning of the experimentation, serve as a baseline to measure any impact of NLD use on SMS size. The first question now refers to specific learnings from the introductory programming course. The second question, rather than evaluating the impact of the instructor, evaluates more generally the impact of teaching, that is, the dimension that the use of NLD will alter.

1. *How much are you learning in the introductory programming course this year? (1: I learned nothing – 10: I learned more than I learned in any other course)*
2. *On a scale of 1–10, how much do you think you could have learned so far this year in the introductory programming course if the teaching had been optimal (clear and engaging activities and exercises, etc.)?*

Our second adaptation, to be administered at the end of the experimentation (also after the assessment test), refers specifically to the experimentation period (about two weeks of activity) related to arrays introduction. Any different results in the experimental class could show a discontinuity introduced by the use of NLD. Any different results in the control class should allow us to factor out a possible specific impact of the topic itself (the arrays).

1. *How much did you learn in the introductory programming course in this last period of introduction to arrays? (1: I learned nothing – 10: I learned more than I learned in any other course)*
2. *On a scale of 1–10, how much do you think you could have learned in the introductory programming course, in this last period of introduction to arrays, if the teaching had been optimal (clear and interesting activities and exercises, etc.)?*

We refrain that all the questionnaires (available in their original form, untranslated in appendix A.4) would be administered to both the experimental class and the control class, again to use the control class as a baseline and also to possibly exclude any factors related to the particular construct/concept of the experimentation.

10.1.3 Concrete design

10.1.3.1 Collaboration with the school teacher

First, we contacted an informatics teacher at the school where the writer of this thesis holds a chair. The teacher had the ideal characteristics for the experimentation. He is a collaborative professional who is open to experimenting with new teaching methods. As anticipated, he was the teacher of two third-grade classes at the beginning of their learning path in introductory programming.

Dialogue with him was constant and fruitful. This dialogue was necessary to concretize the experimentation protocol and, later, to manage its day-to-day implementation in the classrooms. We immediately shared with him the intent to alter the learning context as little as possible, informing him that we would try not to interfere with his teaching habits (see 10.1.1) except for the NLD part of the implementation.

10.1.3.2 NLD sharing and topic choice

First, we shared with the teacher the essential elements of our learning design. We started from NLD motivations: supporting novice students when introducing a new programming concept (see 9.1), particularly when the new construct results in a change of abstraction within the language itself (see 9.3). We also shared his methodological premises (the inspiration from PS-I approaches, and in particular Productive Failure; see 2.6.3) but also the specificities due to the different characteristics (compared to mathematics and science) of introductory programming (see 9.2.2.1). We guided him in understanding the necessity sequences we developed as examples of NLD use (see 9.3.4), clarifying any doubts as we went along.

We then consulted with him on choosing which topic to introduce with NLD. The choice was mainly dictated by reasons of expediency: what the students had learned and were dealing with, what future topics he would cover, and most importantly, what would be an ideal period to organize our experimentation at school. Gathering and cross-referencing all these constraints, which is not interesting to discuss in more detail, we agreed that *arrays* could be the construct (specifically, the data structure) to be introduced with NLD. In particular, it was fundamental to make sure that the requirements (listed in the necessity sequence for introducing arrays; see 3) had either already been addressed with students or would be addressed prior to experimentation. It was a relatively easy task. In agreement with the teacher, we gave up the skill of plotting a bar chart (an element of the developed example but unessential to the learning objective). On the other hand, using a variable as a counter was essential for the execution of the necessity sequence. This skill was already under construction in both classes (like the rest of the prerequisites). Later the ability to use a variable as a counter would have been the main subject of the approach phase to the necessity sequence with the specific approach exercises (see the following 10.1.3.3). At the same time, we found it necessary to make sure once more that the teacher had not mentioned the existence of arrays in his classes. Moreover, as an occasion to go over the essential features of NLD again with him, we recommended that from then on, he should not mention arrays among future topics. This careful reserve was also to be held in the face of possible doubts or requests from students, with respect to which the teacher would necessarily have to take his attention elsewhere. In addition, among the recommendations made to the teacher was to be extremely careful not to reveal any clues about the existence of the arrays, particularly during the crucial P!S phase. Indeed, when students realize they cannot solve the proposed exercise, the teacher likely faces insistence generated by their frustration.

Once the topic was identified and these recommendations made, we illustrated to the teacher the characteristics of this specific necessity sequence: abstraction increase (as opposed to the use of unrelated variables) on data, which results in a soft-necessity scenario (see 3). We explained to the teacher that it was a matter of stimulating the need to use something more convenient and evocative (namely, more abstract) than the tools the students were already familiar with. We explained the need to put them in a situation where that convenience became almost imperative. Indeed, in scenarios in which in new construct/concept is more abstract, it is usually not factually "impossible", not even for students, to solve the problems

with the tools already known (see 9.2.2.3)[2]. So teachers and educators need to construct situations in which using the already-available tools is concretely hindered (e.g., more time-consuming and more code-writing, more prone to errors), like having to deal with 90 variables in the necessity sequence introducing the arrays.

10.1.3.3 Necessity sequence adaptation and development of new exercises

However, we agreed that the necessity sequence for introducing arrays developed as an example (see 3) was also appropriate for our teacher's actual classes. However, there was the need to translate that sequence (originally developed in Python) into the programming language the students were learning to program, i.e., C++. Also, in addition to this, the need to prepare other approaching exercises (see 9.2.1) in addition to those already proposed for the sequence to actually enable the students to develop the mastery that the *necessity mechanism* needs to work. Also to be developed, again in C++, were simple consolidation exercises on arrays, to be proposed after the pivotal P!S-PS exercise (the one that should trigger the necessity of arrays) for the consolidation phase. Finally, it was necessary to write texts in Italian (the example necessity sequences were developed in English) that would be as similar as possible (both in wording and in the way of posing the requests) to those that the two classes were used to deal with (remember our intent to interfere as little as possible; see 10.1.1). Therefore, we agreed with the teacher that we would be the ones to adapt our exercises and their solutions in C++ and develop the new ones needed. After that, he would check their coherence with respect to the classes' habits and skills. Conversely, he would be in charge of the texts of the programming exercises, and we would ensure that they were scientifically sound and suitable for NLD in our experimentation. These two specular processes have proceeded relatively linearly and smoothly. Adjustments made to the exercises by us (at the teacher's suggestion) and the texts by the teacher (at our request) were minimal and not substantial. All the exercises developed, broken down by stages of experimentation (thus of NLD), are provided in appendix A.3.

10.1.3.4 Formalization of the instruction phase

An indispensable step was working with the teacher to formalize his instruction phase (I) and make sure it had the required content (and nothing more, since we are dealing with the introduction phase) to implement the necessity sequence that introduces the arrays. First, we clarified again to him how NLD is intended only for introducing a new programming concept and, thus, how this constraint limited our experimentation to a specific and circumstantial learning phase of arrays. The teacher would then use his usual methodologies to develop mastery over arrays and build new programming skills from them. The same goes for this data structure's implementation and theoretical aspects; this goes beyond the scope of NLD experimentation and is not discussed.

It was crucial to agree on the minimum content the teacher would present to the control class in a typical I-PS (instruction-problem solving) sequence. The same content would

[2]As explained in 9.2.2.3, when a language is Turing complete, no computable task is truly impossible, not even in hard-necessity scenarios, even if it seems so from the student's perspective.

instead be presented to the experimental class after the first unsuccessful problem-solving phase (P!S), according to the NLD sequence (P!S-I-PS). The minimum and required content involve array initialization (we have selected some, the most relevant, of the possible ways offered by C++), access by index to read and write, and how with a simple *for loop* it is possible to perform a linear scan of all elements of an array. The teacher proposed the materials he usually uses to introduce the arrays. Again, according to the intent of change as little as possible, our intervention involved only the selection and a (non-substantial) rearrangement of the content just listed. This selection was at the expense, for example, of all the implementation details concerning array memory allocation, which the teacher was willing to address at a later time outside the experimentation. At the same time, we selected emblematic and simple examples together, again leaving some of the more advanced applications of arrays for after the experimentation. Otherwise, the teacher's original materials and related content were scientifically correct and did not require any other modifications. The materials resulting from this collaborative work with the teacher were, therefore, similar in all respects to the materials proposed to the students up to that point, in wording, style and presentation. We emphasize again how the same materials were used in the instruction (I) phase in both the control class (I-PS) and the experimental class (P!S-I-PS).

10.1.3.5 Assessment and management of the activities

The teacher's mode of assessment varies according to the learning moment. Customarily, the teacher proposed a first assessment moment very soon after introducing a new topic. This verification produced a grade in students' careers, having, however, for the teacher mainly formative value. Indeed, from the indications of this first assessment, he decided how to set the following stages of the learning activity. This first assessment was typically structured in a few simple programming exercises and multiple-choice questions to assess understanding of the constructs/concepts involved and their related basic skills. We considered this mode appropriate for our measurement needs and that it did not require any restructuring. This choice was particularly relevant to make grades from this assessment as homogeneous (and thus comparable) as possible with the history of previous grades. Together with the teacher, we developed three simple programming exercises and four multiple-choice questions (provided, untranslated, in appendix A.2), requiring a basic understanding of arrays. We developed these exercises and questions to be as relevant as possible to the content presented in the Instruction phase (see 10.1.3.4 above). For the wording, style and requests, we relied on the teacher for as much mimicry as possible. For the same reason, the correction of the students' attempts was carried out entirely and autonomously by the teacher, according to his usual methods and without any supervision or intervention on our part.

Sharing the teacher's assessment methods was also an opportunity for us researchers to learn what the classes used as learning tools. The school that hosted the experimentation adopted *Google Workspace for Education*[3] as its platform for teaching and learning. The teacher with the two classes was making full and conscious use of it and of its services (e.g., Gmail, Documents, Presentations, Classroom). In particular, materials, in-class exercises

[3] https://edu.google.com/intl/en_ALL/workspace-for-education/editions/overview/

and homework (and related students' submissions) were proposed, managed and evaluated through *Google Classroom*[4]. Google Classroom was also used for the assessments and tests (both formative and summative), during which, however, the students could not freely browse the Internet, except for the C++ documentation site. The teacher advised students to use the Code::Blocks educational IDE or a simple text editor with terminal compilation and execution. In any case, each student was free to use the preferred development tools, and the teacher assisted students in any case. Keeping with the principle of perturbing the setting as little as possible, we did not make any change requests, nor would it have been helpful for the experimentation.

Finally, we briefly report on the administration of the short (2 questions) in-progress questionnaire to follow the most significant phases of the activities (see 10.1.2.3). With the teacher, we considered that however short, we could not interrupt activities too often. The reasons were to avoid too much wasted time and attention lapses (always possible in a school setting with numerous classes) and to avoid students being bothered, perceiving the questionnaire as a burdensome task rather than a chance to express themselves. Therefore, we agreed to administer the questionnaire in the experimental class only after the three phases of the necessity sequence (P!S, I and PS) and in the control class after the two "traditional" phases (I and PS). In addition, we would only propose it on another occasion, after the approach exercises, as this phase was also a novelty (though not evident to the students) introduced by the experiment.

10.1.3.6 Choice of the class for the experimentation

Several factors determined the choice of the experimental class. First, NLD characteristics that make our learning design ideal for introducing new programming concepts to novice or inexperienced students suggested choosing third-grade classes from the technological-informatics track of a technical high school. Indeed, during that third school year, students (after a general introduction to some basic informatics concepts and digital literacy) actually begin programming. Moreover, our desire to control the experimentation's conditions as much as possible prompted us to look for a teacher who had two third-grade classes in the same school year, a situation not obvious to find. As mentioned above, a colleague of the author of this thesis, who is collaborative and open to trying new teaching methodologies, was in the situation (ideal for our experimentation) of holding the informatics chair in two third-grade classes.

The choice then came down to which of these two classes should be the experimental class, where to experiment with the necessity sequence to introduce arrays, and the control class. The two classes were comparable "on paper" (same track, same school year, same point in introductory programming learning path) and also in size (23 and 21 students). However, the two classes were quite different in reality and, all the more so for that reason, not directly comparable (see 10.1.2.2). According to their teacher, one class (21 students) was quite motivated with no particular learning problems. The other one (23 students) was less motivated, more turbulent, and more fragile from a learning perspective. About the

[4] https://edu.google.com/intl/en_ALL/workspace-for-education/classroom/

latter, the teacher reported that most students were struggling to comprehend (and thus apply) the earliest programming concepts and grasp the related informatics principles. Lower grades (than the other class) from September – the start of the school year and their first programming course – to late December certified this problematic situation.

We found it more valuable and helpful to use NLD in the more fragile class. First, we decided so because among the very goals of our learning design is precisely to support fragile learners, who typically fail to catch the positive wave of the *Learning Edge Momentum* and instead get swept away by it (see 1.4). Second, since the success of a necessity sequence depends on students not knowing the target concept whose necessity the sequence aims to stimulate in them (arrays in our case), we felt it safer to use NLD in the most fragile class. Indeed, in the other class, some students were passionate about informatics and had already attended extracurricular programming initiatives and courses in which they might have already known about arrays or similar data structures.

10.1.3.7 Definition of a concrete and balanced timetable

At this point, it was necessary to set a concrete timetable. To do this, we needed to address two dimensions.

The purely organizational one, which concerned the weekly schedule of the two classes, the availability of the lab rooms where the students have the possibility of using one computer each (as opposed to the traditional classroom, where most of the theoretical lessons take place), the need to have the lab for at least 2 hours, preferably 3, in order to carry out the necessity sequence in the same morning. Other requirements involved dealing with the introduction to arrays maximum in the span of two weeks and without too many breaks between the various phases other than the sequence (approach exercises, consolidation exercises, assessment, and also pre and post questionnaires) and being able to carry out the activities aligned between the two classes, so that students would not have (too many) occasion to share with any classmates in the other class relevant information (especially the existence of arrays) that could jeopardize the whole experimentation. The two-week duration for introducing arrays is motivated by the institutional need to respect the activities' general pace and also by our intent to interfere as little as possible (see 10.1.1). Thus, even with the use of NLD, experimentation times need to be substantially in line with those customarily scheduled for introducing arrays. Such many constraints, set in a highly unflexible context such as the school (where classrooms, labs, teachers and subjects are intertwined in a complex balance that is very difficult to set up), made defining the experimentation's timetable one of the most challenging and critical phases. However, thanks to the teacher's willingness and a few other colleagues' flexibility, we managed to squeeze all the activities in two weeks, the same for both classes. Something not mentioned so far, nevertheless relevant, is that the two classes (although very different) had to be at the same point in the learning path (which also included the prerequisites for the array necessity sequence; see 10.1.3.2) to be able to proceed side by side in introducing the arrays. Again, we have to thank the teacher's commitment and cooperation.

On the other hand, the other dimension that was decisive in defining the concrete timetable imposed a balancing of the hourly load between the two classes. It was necessary that the

students, in terms of time spent, would be given the same opportunity to assimilate the arrays. This requirement is one of the most significant in ensuring that the only discontinuity between the two classes was whether or not NLD was used. The core of the activities are phases I (instruction) and PS (problem solving) for the control class, and P!S (problem not solvable), I and PS for the experimental class. As can be seen, the experimental class has an additional phase, which amounts to extra time (in which students fail to solve the exercise and should develop the necessity of arrays). We considered this time akin to what students spend on consolidation exercises. Therefore, we allocated additional time for the consolidation phase of the control class (30 min and one more exercise), roughly equivalent to the time spent by the experimental class on the P!S phase. Only roughly, because the control class was given 15 min more in the Instruction phase since it was initial and could not take advantage of the students' necessity feeling (that should prime for learning; see 9.2.1), and 15 min more to solve the programming exercise (the same of the P!S phase for the experimental class) in the PS phase since it would be new to the control class. Here discontinuities end, thus limited to the core phase of the activities (i.e., I-PS in the control class and P!S-!-PS in the experimental class). Before the core, both classes must train equally on the approach exercises to develop mastery. After the core phase, both classes engage in consolidation exercises again in the same way. Finally, after consolidation, an assessment of the learnings related to the essential elements of the arrays (see 10.1.3.4) follows. At the beginning of the experimentation, the students spend about 10 minutes filling out the questionnaire (almost exclusively based on Likert scales and therefore quantitative) surveying their perceptions of informatics; the questionnaire repeats identically at the end, with additional questions specific to the array learning experience.

Experimental class (approximately 7 hours total)

1. Approach phase (2 hours)
    a. 80 min autonomous programming exercises
    b. 40 min teacher-led correction and alignment
2. **Core experimental phase** (3 hours)
    a. 60 min P!S (problem not solvable) phase
    b. 45 min I (instruction) phase
    c. 45 min PS (problem solving) phase
    d. 30 min teacher-led correction and alignment (P!S exercise correction with explicit demonstration of how to use arrays to solve it)
3. Consolidation phase (2 hours)
    a. 80 min autonomous programming exercises
    b. 40 min teacher-led correction and alignment

Control class (approximately 5 hours total)

1. Approach phase (2 hours)
    a. 80 min autonomous programming exercises
    b. 40 min teacher-led correction and alignment
2. **Core "traditional" phase** (2:30 hours)
    a. 60 min I (instruction) phase
    b. 45 min PS (problem solving) phase
    c. 45 min teacher-led correction and alignment (PS exercise correction with explicit demonstration of how to use arrays to solve it)
3. Consolidation phase (2:30 hours)
    a. 100 min autonomous programming exercises
    b. 50 min teacher-led correction and alignment

10.1.3.8 Privacy policy and consents

Because the experimentation took place in a big and important school, we found a tried-and-true bureaucratic setup. Indeed, the school provided us with the template (prepared by their team of experts and already used on similar occasions) of information for managing the student data we would collect during the experimentation. Specifically, the data were the students' first and last names – but only to relate the various activities outputs to the same student (the data would then be anonymized)[5] – the students' outputs of the various activities, in particular, the programs they developed and their answers to the assessment test and the questionnaires, and the grades issued by the teacher. The drafting of the final privacy policy, declined for our experimentation, was also particularly agile because we considered not making audio-video recordings of the various activities. The intent was, once again, to make our presence as observers as unobtrusive as possible to make the activities unfold as naturally and similarly as usual.

The whole process was straightforward also because every educational activity (including assessment tests and questionnaires) took place on the school's Google Workspace platform, and any related data (students' schedules, questionnaire completions, test evaluations) remained on the platform. The school provided us with a temporary account (whose duration was limited to the time of the experiment and the next two weeks) with which we could access these data and create materials (such as exercises and questionnaires) and monitor student activity. The experimentation privacy policy was also provided to the students via Google Classroom and collected signed (by the parental authority holder) through the same

[5] It would have been possible to associate students with codes to avoid collecting their first and last names. However, we feared that asking students to use the code system to identify themselves in the various activities would have disrupted too much the ways they were used to (see 10.1.1).

channel. All the students consented to participate in the experimentation and to allow us to access the related data.

A final note. Before proceeding to the concrete design of the experimentation, we consulted our supervisors within the informatics department to consider whether we should convene the ethics committee and seek formal authorization to proceed. Since our experimentation would affect only a specific and limited phase of the teaching activity (essentially resulting only in the reversal between the instruction phase and the problem-solving phase), without altering the educational setting in other significant ways (e.g., the evaluation criteria), it was judged unnecessary. This decision was also made because we would not be collecting and handling sensitive data, the data collected would be anonymized, and no audio-video recordings would be made.

10.2 Implementation

Most of the observations we report here are related to the experience of the experimental class. Although our observation effort also covered the control class, the activities there took place without any particular things to report. Indeed, according to the teacher, the array introduction followed the habitual times and ways for the control class. Apart from our silent presence (as non-participating observers) and the brief moments dedicated to filling out questionnaires (pre, post and the in-progress quick ones; see 10.1.2.3), there were no other differences from the habits of the class. Even the defined and specific timetable did not alter the usual times and manner of the class, a sign that the timetable definition (in agreement with the teacher; see 10.1.3.7) was adequate for the learning objectives of arrays introduction and complied with our intent of non-interference (see 10.1.1).

10.2.1 Keeping arrays secret

The first thing we report is the difficulty, already foreseen in the design phase (we discussed this in 10.1.3.2 and 10.1.3.6), of making sure that none of the students involved in the experiment knew about arrays. We report the difficulty of knowing this information before the implementation of the necessity sequence. In particular, during the phase of the approach exercises, this doubt (and its potential negative effect) prompted us to reconsider the issue. Together with the teacher, we evaluated whether there were ways to verify it, given how critical this precondition is for NLD functioning and effectiveness. Finally, we agreed that it was not prudent to investigate it since we might have incited students' curiosity and, thus, possibly, subverted a potentially ideal situation. We agreed with the teacher on some strategies we would implement during the core phases of the experimentation if any of the students showed that they knew about the arrays. For example, the teacher would try to deflect the discussion by not showing interest in the "informed" student's remarks in front of the class. The teacher then would try to talk to her privately as soon as possible, congratulating her on her knowledge, and offering all the support and clarification she might need. He would also ask the "informed" student not to share information about the arrays for the smooth running of the activities and out of respect for her classmates.

Luckily, this situation did not occur. From this, however, it is possible to draw some considerations regarding an obvious limitation of NLD, which we discuss in 10.3.1.

Regarding the necessary secrecy to be maintained about the target concept of the experimentation (the arrays), we also report the difficulty for the teacher (already foreseen in the design phase; see 10.1.3.2) to cope adequately with students' insistent requests during the P!S phase. There were no actual problems in this first implementation: the teacher was flawless in not revealing array information during the P!S phase. However, we reiterate how important it is and point out that we repeated this same recommendation to the teacher many times before and during the implementation. Many students asked how the proposed exercise was to be solved and whether there was a specific tool for doing so. Some of these students also asked us (in our role as observers) the same questions, and even from our perspective, we corroborate the difficulty of handling students' insistence in the P!S phase. Of course, we maintained our role as silent observers in that situation, deflecting the students' inquiries and answering that we did not know how to solve the programming exercise.

10.2.2 Exercises to approach the necessity sequence

For a *necessity sequence* to work, students must develop mastery of the *approach exercises*. Approach exercises are similar in wording and requests to the P!S exercise, the core of the sequence meant to generate the necessity of the target concept in students. However, students must know very well how to solve them with the tools they already know and use (see 9.2.1). During the correction of these exercises, we observed some students (many in the experimental class) bored, distracted, and passive, and, in parallel, few students engaged and focused. However, the teacher described this attitude as rather usual when correcting the exercises.

10.2.3 P!S phase: unsuccessful problem solving

The hour devoted to solving the P!S (problem not solvable) exercise in the experimental class was eventful. The exercise required keeping track of one million lotto draws, with numbers (randomly generated by an already provided function) ranging from 0 to 89. The version proposed in the necessity sequence examples can be seen here 3, and the version adapted in C++ for the experimentation can be seen in appendix A.3.2.

Some students autonomously set out in small groups (2-4 students) to search online for the way to create parametric variables, following the intuition of creating and somehow automatically managing variables of the type `extract`*N*, where *N* was the number actually extracted. One of these groups landed on a set of pages presenting arrays in C++, complete with theoretical and implementation details and many usage examples. However, they could not select and use that information to solve the P!S exercise in the assigned hour.

The other significant behavior that emerged involved seven students, who, in pairs or individually, decided to try solving the problem with the tools they already knew, i.e., by defining and managing 90 variables and implementing a conditional structure (made of 90 interrelated if statements) for updating them. None of them succeeded in solving the proposed exercise, complicit in the length of the code and the numerous errors made in developing it.

Other students initially followed this path but abandoned the attempt when they realized there had to be other options than such a lengthy solution.

Many students insistently asked the teacher (and sometimes even us) what was the way to solve the proposed exercise but did not get helpful answers (see 10.2.1).

Some students, sensing that the exercise was not solvable with their current knowledge, jokingly complained about the situation. Among them, some confessed to feeling frustrated or very frustrated, and a couple were even angry with the teacher. In general, as the hour approached its end, we noticed an increase in frustration and agitation in many students, even among those who had set out at the beginning of the phase to work calmly.

In general, one thing was immediately clear to the teacher, who immediately shared it with us (and repeated it later on other times). The class, which generally dealt quite passively with all teaching activities, even those designed to be more active and engaging (such as moments of participatory correction and collective discussion), came alive with unusual energy.

Finally to be reported was the difficulty in getting students to stop their attempts to solve the problem, particularly those trying to use online resources and especially those who had taken the 90-variable route. The teacher had to use his authority to get all the students to stop using their computers and tune in to the instruction phase that was about to begin.

10.2.4 Instruction phase

First of all, we were thrilled to observe that the instruction phase was much more closely followed and participated in the experimental class.

In addition, we report two more significant macro-observations about the instruction phase, the first concerning the students' experiences and the second the teacher's.

Following on from what has just been said about the problematic termination of the previous P!S phase, many students (at least one-third) showed difficulty focusing on the teacher's explanation as they repeatedly returned to trying to develop a program that solved the exercise. Some students did so without considering what the teacher was explaining. Others, on the other hand, were moving from what the teacher was illustrating, however often gathering just insufficient information and thus still being unable to solve the exercise. Meanwhile, the explanation went on, and they missed essential elements of it. These students, who repeatedly disconnected from the explanation to return to the exercise, all missed essential elements of the instruction phase. The negative consequences are reported in the following A.1.

The other significant observation concerns the difficulty the teacher experienced explaining arrays to keep up with the NLD requirement of not anticipating anything in the instruction phase about using the target concept to solve the problem that the class had just unsuccessfully addressed. In other words, the teacher knew that he had to introduce arrays and their essential elements (i.e., initialization, access by index, and linear scan) in a general way and through simple examples (following the agreed-upon approach and materials; see 10.1.3.4) without ever referring to their use in solving the P!S exercise. The teacher had to sidestep all the numerous and frequent questions from many students asking if arrays were indeed the tool to solve the exercise and how to use what he was explaining to solve it. Also, these questions often interrupted the explanation, causing difficulty for the teacher himself and constituting

another (other than the temptation of getting back to the exercise by themselves) source of distraction for the students.

10.2.5 PS phase: second problem solving

Of particular note are two things in the problem-solving phase concluding the necessity sequence.

The first was a vitality level in the experimental class that contrasted (again, according to the teacher) with its "usual" habit. As mentioned, many students were already pawing during the previous instruction phase to get back to work on the exercise. In general, all students seemed willing, and some were even enthusiastic, to get back to the computer to develop a solution.

The second observation is less positive. It concerns the difficulty of solving the exercise even though, at this phase, students had the target concept (i.e., basic knowledge about arrays) at their disposal. Only a few students, about one-third, managed to develop a correct and working program that solved the proposed exercise using arrays. During the students' autonomous work, as the teacher moved around the lab room giving support to anyone who asked, two main reasons for this difficulty emerged. The first is anticipated already in the previous report of the instruction phase. Those students who, while the teacher was explaining, became distracted and tried to solve the exercise already at that phase missed fundamental elements from the arrays' explanation. Indeed, without those pieces of knowledge, they could not produce a correct and working program, although most of them seemed to have understood the general purpose and use of the new tool.

On the other hand, the second reason for this difficulty is due to a design flaw in the P!S exercise. Indeed, developing it, we did not realize how the solution required not only using an array as a structure for keeping track of linked data (the extracted lotto numbers) but also using the same array within a loop other than the one that would perform its linear scan. The solution loop repeats for the number of extractions (thus, it is not the loop that would scan the array linearly shown in the instruction phase). Within this loop, it is necessary to understand that one must use the extracted number not only as the information updating the corresponding counter but also as a mechanism to access the corresponding position in the array.

Many students did not realize this particular conceptual step: using the extracted number (also) as an index. Undoubtedly it would have helped to include among the information and examples of the instruction phase the use of a variable – other than the traditional index (conventionally `i`) of a *for loop* – as an index to access an array element. Some reflections on this are in the following 10.3.4. One final observation. Although the experimental class already knew how to use loops to realize determinate iterations loops (and had been using them for a few months), it seems that up to that point, the loop index was for them only an "invisible" cog participating in its operation.

10.2.6 Correction and alignment, consolidation, and later steps

The teacher's participatory correction of the P!S exercise – answering the class's questions and making continuous reference to the essential elements of the arrays presented in the instruction phase – seemed quite effective in aligning all the students in understanding the optimal solution (despite the difficulty unintentionally introduced by the P!S exercise design illustrated in the previous section). Even in this phase, the class was quite engaged and lively, again in contrast (according to the teacher) to the attitude shown up to that point in the school year.

The subsequent consolidation phase (autonomous yet assisted exercises and collective correction) and the final learning assessment then took place according to the agreed timetable, with no particular events to report.

We have not yet had a chance to analyze the students' programs developed in the consolidation phase, nor to analyze (concretely implementing the protocol described in 10.1.2.2) the grades assigned by the teacher for the assessment test on the arrays' introductory knowledge and use.

However, the teacher gave us his preliminary impression, which he urged us to take with caution as it was rough and not the result of a more precise analysis. The control class, traditionally motivated and well-prepared, scored in line with its history and expectations, generally positive or very positive, with few failures. The experimental class, on the other hand, appeared to be slightly improving from its usual level. There were only two severe failures (whereas there were usually about a quarter on the first test on a new topic), and less than a quarter of the class failed. Many students (about half) were around or just above sufficiency, while about a quarter scored good or very good. Thus, about three-fourths of the students met at least the minimum objectives on the introduction of arrays. This landscape is an improvement over the class history, which, again from the teacher's estimate, usually saw about half or a little more of the students getting a passing grade on the first assessment test after the introduction of a new topic.

As mentioned, in a later phase of our research project devoted entirely to rigorous analysis of the data (the assessments grades, but also the questionnaire fillings and the observations collected in our journal), it will be possible to calculate that *delta* (see 10.1.2.2) on a per-student and class basis and more objectively evaluates the impact of using NLD to introduce the arrays. In addition, being able to rely on the same measures in the control class will allow us to factor out (at least partially) a possible perturbative component inherent to the array topic itself.

10.2.7 Administering questionnaires to the students

There is nothing relevant to report about the two moments of administration of the initial questionnaire on students' self-perceptions about programming and their experience and motivation in learning it (see 10.1.2.3). These were two relatively short moments (about 15 minutes each) that took place, both in the experimental and control classes, before every activity of our experimentation and at the end of everything (after the final assessment). On the other hand, about the quick in-progress questionnaires (see again 10.1.2.3) that we asked

to fill out immediately after the most significant phases, we confirm what was already foreseen in the concrete design phase of the experimentation (see the end of 10.1.3.5). Although structured in an extremely light form to be answered quickly, this questionnaire significantly interrupted the classes' activities. Indeed, aiming to give all the students a chance to answer the two questions personally, some took longer than expected (about 10 minutes total for the two questions), despite the invitation to answer quickly, even with one word. During these moments, most students, having already answered quickly or very quickly, set themselves in "pause mode", making it challenging to resume the activities, resulting in wasted time and drops in attention.

10.2.8 On our role as external observers

Finally, about our role as observers, we report that it was not easy to stick to the purpose of being as external to the activities as possible. In particular, the writer's habit of working with high school students led him to give some guidance of a purely practical nature to make the development of activities easier for some experimental class students who, at various moments, seemed uncertain about what to do. On the other hand, this cooperative attitude led some students to see us as a resource also from an educational point of view. As a result, we received several questions on the programming exercise of the necessity sequence during the P!S phase and the PS phase. Deflecting the questions was relatively easy. On the other hand, however, some students asked if our presence was related to the new way they were dealing with the arrays. Unfortunately, this testifies to the fact that somehow, although not intended, the careless suggestions we gave betrayed, at least in part, our intent to remain as invisible as possible and to interfere as little as possible with the learning context (see 10.1.1).

10.3 Preliminary observations and results

10.3.1 The teacher must keep the secret

The need (already evident in the development of the learning design) for *none* of the students in a class in which NLD is used to introduce a new programming concept to be familiar with that same concept emerged strongly at the beginning of the implementation. Verifying this precondition a priori and checking that it applies to all students in the class makes this the weakest point of NLD. First, it gives even greater force to the recommendation of using necessity sequences only for introductory programming and with substantially novice students. This is to reduce the likelihood that students have already been exposed to the concepts to be introduced with NLD and, at the same time, do not have the tools to immediately discover the missing concept during the P!S phase. Indeed, at that stage, it is essential that they do not easily discover the existence of the concept/construct because it would be like exposing them to it right away with an instruction phase that precedes the problem-solving phase. They need to experience a little frustration at not being able to solve the proposed exercise, so they can mature the necessity of that concept and thus prepare for its learning. Finally, the difficulty experienced during the first school implementation reaffirms that NLD

cannot be used on many occasions or too close together, at the risk of students expecting this modality, thus making it ineffective.

In addition, during the instruction phase, it was not easy for the teacher (so it appeared to us, and the teacher reported the same) not to answer the multiple questions from the students about using arrays for solving the P!S exercise. He struggled to find ways not to answer without making the students feel ignored. We report this difficulty but cannot formulate meaningful reflections about it, except again, the recommendation to use NLD sparingly.

We thus return to one of the recommendations already expressed in the description of our learning design (see 9.4). Educators should limit the use of NLD to introductory programming concepts and do so primarly when their introduction results in a change in the level of abstraction (upward or downward, within the language studied) from the student's perspective.

10.3.2 Students' frustration

We take the opportunity of what has just been stated to reflect on the students' frustration level in the experimental class described in 10.2.3. These observations are confirmed by a preliminary reading of the responses of the in-progress questionnaires conducted after the P!S phase and of the final four open-ended questions designed to investigate students' opinions of NLD as a learning mode (see 10.1.2.3).

From the in-progress questionnaire administered immediately after the P!S phase, we report (our translation of) some significant student answers to the question *'What do you think about the activity you have just experienced?'*: (i) "It is frustrating not to be able to find a solution"; (ii) "An incitement to hate"; (iii) "A punch in the stomach".

From the same questionnaire after the P!S phase, we report (our translation of) some significant student answers to the question *'How do you feel after the activity?'*: (i) "Stressed"; (ii) "Frustrated by humanity"; (iii) "Exhausted"; (iv) "Interested, but nervous and agitated".

From the post-questionnaire administered at the end of the experimentation, we report (our translation of) some significant student answers to the question *'How did you feel during the time when you did not know how to solve the lotto problem (because you did not yet know arrays)?'*: (i) "My head was exploding"; (ii) "Helpless"; (iii) "Confused, with my back to the wall"; (iv) "Fairly blown away".

From the same post-questionnaire at the end of the experimentation, we report (our translation of) some significant student answers to the question *'What aspects did you not like [about the way arrays were introduced (having you try to solve the lotto problem before explaining arrays)]?'*: (i) "It makes no sense to have a student who is a first-year in informatics programming do something that has not even been explained"; (ii) "For me it was crazy stress"; (iii) "I find it was a little frustrating".

All these answers precisely reiterate the recommendation made at the end of the previous paragraph, namely to use NLD sparingly, only when it can be most helpful (when abstraction changes). The students' perspective thus lends additional depth to this recommendation, mainly for two reasons. The first is related to the effectiveness of NLD. Using too much NLD would cause students to tip off and passively wait for the instruction phase without any

benefit to motivation. The second is about respecting their well-being and sense of justice, which is often particularly strong in young people. Suppose a student feels too frustrated or, worse, deceived. In that case, students may challenge this learning method altogether, not only making it ineffective but even counterproductive compared to more traditional methods, possibly even raising relational and trust issues with the teacher.

In addition, during the implementation, we observed an increase in frustration and agitation as the hour was ending, even among those students who had set out at the beginning of the P!S phase to work calmly (see 10.2.3). So the P!S phase cannot last too long. One of the challenges, then, to be reinterpreted from time to time depending on the concept introduced with NLD and the characteristics of the class (educational, human, and also the familiarity of the class with NLD) is to find the right balance. The balance is to be found between a time long enough for the students to mature the necessity of the target concept but not too much to make them too frustrated and thus ill-disposed.

10.3.2.1 Managing student frustration

Student frustration is a natural consequence of that *desirable difficulty*, a condition for the *necessity mechanism* (9.2.1). However, frustration must be kept within limits that make it functional for learning and not a potential risk of falling into the same issues of prior informatics education work, where learners drop out or outright see programming as an unattainable skill mainly because of the frustration.

The recommendation that NLD has to be used sparingly at specific and carefully selected learning moments also serves to limit the negative effects of frustration within functional boundaries, as already recognized in the development of the learning design (see 9.4.1).

The other fundamental counterbalance to student frustration is keeping the P!S phase short, which emerged during this experimentation (10.2.3). Specifically, one hour had been allocated for the P!S phase (10.1.3.7). However, during the experience with the class, after about 45 minutes, the teacher estimated that all students had experienced the abstract necessity of arrays and at least a hint of frustration in not knowing them. The following is not a scientific consideration, but the teacher should have a finger on the pulse of the situation. The teacher's experience and knowledge of those particular students could help her or him decide to anticipate the following instruction phase or to allow a little more time if the students seem positively engaged. Ideally, the I phase should begin when all students have experienced precisely that necessity, i.e., they have understood the problem and at least intuited the tool they would need (e.g., "I would like a way to index variable names").

In addition, as an antibody to the negative effects of frustration, NLD is structured to "transform" frustration immediately. The I phase (minimal and rapid) directly follows the P!S phase. The following PS phase allows the energy of frustration to be put back into circulation, transforming it into success, i.e., the exercise solved. When this does not happen, the subsequent consolidation phase, designed to align all students with the new learning, should put frustration aside.

10.3.3 A boost to motivation

In the second phase of problem solving following the instruction, only about one-third of the experimental class students were to solve the P!S exercise using arrays. This is not necessarily a negative result since solving the P!S exercise is not the true objective of NLD (although it is an indicator of its effectiveness). The objective is to prepare students for learning arrays, both from a motivation and a cognitive standpoint. The observations, confirmed by the teacher and his familiarity with the class, reported increased interest and motivation. The performance during the instruction phase, with students generally more participatory than usual and focused on the explanation, is also a good indicator (albeit a preliminary one) that the P!S phase is a helpful cognitive preparation for learning a new concept (in line with one of the most significant benefits of Productive Failure, see 2.6.3.1).

In addition, the experimental class's lively participation in the second problem-solving phase (even though some students still failed to solve the problem) is very promising. It (preliminary) suggests that the choice not to anticipate the solution of the P!S exercise in the instruction phase (unlike Productive Failure and, in general, PS-I approaches) helps capitalize on a learning potential generated along with the development of the feeling of necessity. It also suggests that this choice effectively stimulates motivation, even in fragile students. We recall that on many occasions, the teacher declared himself amazed at how the experimental class, usually passive and unmotivated, lit up with unusual vitality. In support of this decision, and preliminary (with respect to a more accurate analysis of students' self-assessments) evidence of NLD's positive impact on motivation, the vitality and interest shown by the students during the three phases of the necessity sequence also prolonged in the subsequent correction and alignment phase and also, albeit to a lesser extent, in the consolidation exercises phase.

10.3.4 Difficulties in solving the exercise in the second PS phase

However, let us return for a moment to the fact that it was still not possible for many students to solve the exercise in the second stage of problem solving. There are some reflections to be made in this regard.

The first is the most general. Namely, *strategic knowledge* is the most difficult to acquire of the three knowledge related to knowing how to program (see 2.3.1), and the one that takes the longest to develop since it corresponds to the skill of using something one knows and knows how to use to solve a more general problem. It is obvious that this knowledge/skill cannot be fully developed in only three hours of a necessity sequence, nor is this the goal of NLD (see 9.2.2.2). A similar argument can be made for the higher levels of knowledge of the SOLO taxonomy; on this, we refer to the end of 9.2.2.1.

The second reflection concerns the level of engagement generated by the P!S problem, which is also determined by feelings of frustration. We observed that this engagement could distract students in the instruction phase because they feel compelled to return to the P!S exercise. This distraction prevents them from acquiring all the elements of the target concept necessary to solve the exercise effectively. In addition to suggesting that teachers should exert more control over students, which is only possible up to a point, this observation prompts

considering a change of location between the P!S phase and the subsequent instruction. Indeed, without having the computer at their disposal during the instruction phase, students can focus solely on the explanation.

The third reflection concerns the nature of the P!S exercise. In the problem-solving phase after the instruction, some students did not solve the exercise with arrays because they decided to go ahead with the 90-variables suboptimal solution (to borrow terms from the Productive Failure literature; see 2.6.3.1). This scenario testifies to the unexpected difficulty (yet foreseen during the development of NLD; see 9.3.2) of moving up in abstraction level and abandoning known tools for easier ("on paper") and more suitable ones. In a *soft necessity* scenario (see 9.2.2.3), this convenience is not evident from the perspective of students who, therefore, must be effectively pushed not to use what they already know. Evidently, the 90 variables (which corresponded to the 90 possible lotto numbers) were too much for many but not for all. This experience prompts us to consider reformulating this P!S exercise (and others in *soft necessity* scenarios) in an even more extreme way, for example, by imagining a scenario in which the variables were not just 90 but, say, 200. In short, what from the theoretical point of view of NLD is a *soft necessity* must turn out to be a full-fledged necessity (i.e., *hard necessity*) from the student's perspective.

Again on the P!S exercise, we recorded the difficulty for many students to solve it even after knowing the arrays due to its design flaw (described in 10.2.5). In this regard, it only remains to urge educators who intend to use NLD to take extreme care in developing particularly the P!S exercise, which is the heart of the necessity sequence and, ultimately, of the whole learning design. The P!S exercise must be calibrated precisely to the content of the instruction phase, no more and no less. Added to this recommendation is the invitation, if possible, to have (at least) the P!S exercise tested by students in a similar situation as the students for whom the exercise is intended. The aim is that problems such as this may emerge before the actual use of NLD.

10.3.5 Positive students' feelings and opinions

Having highlighted the positive effects on motivation that emerged from the use of NLD in our experimentation, but also made recommendations to limit the negative ones and collected NLD's main limitations, we like to end this review of preliminary results by reporting the most relevant positive statements of the students. We particularly point out that in the last question reported at the end of this subsection, we record an overwhelming majority of positive opinions about the usefulness of NLD in learning arrays.

From the in-progress questionnaire administered immediately after the P!S phase, we report (our translation of) some significant student answers to the question *'What do you think about the activity you have just experienced?'*: (i) "A nice challenge"; (ii) "Formative"; (iii) "Fascinating"; (iv) "Useful".

From the same questionnaire after the P!S phase, we report (our translation of) some significant student answers to the question *'How do you feel after the activity?'*: (i) "Interested; (ii) "Curious about what the solution is"; (iii) "Excited to see if I can do it"; (iv) "Interested and intrigued".

From the post-questionnaire administered at the end of the experimentation, we report (our translation of) some significant student answers to the question *'What aspects did you like about the way arrays were introduced (having you try to solve the lotto problem before explaining arrays)?'*: (i) "I liked the method of asking to do an exercise before explaining the solution"; (ii) "I appreciated the idea of trying to solve the problem without the arrays"; (iii) "The fact that I had a chance to try it on my own first"; (iv) "I find it was a good idea to show us how useful an array is"; (v) "[I liked] Everything".

From the same post-questionnaire at the end of the experimentation, we report that to the question *'What aspects did you not like [about the way arrays were introduced (having you try to solve the lotto problem before explaining arrays)]?'*, 7 students (out of 23) responded (in various forms) "None".

From the same post-questionnaire at the end of the experimentation, we report (our translation of) some significant student answers to the question *'How did you feel during the time when you did not know how to solve the lotto problem (because you did not yet know arrays)?'*: (i) "Engaged and curious"; (ii) "Definitely interested in finding a solution"; (iii) "Motivated"; (iv) "I knew something like this would exist". The last answer highlights one of the feelings that, from the beginning of the research that led us to develop NLD, we would have liked to stimulate in students: the feeling that something must necessarily exist to solve a particular problem. This intuition is one of the ingredients of the *necessity feeling*.

From the same post-questionnaire at the end of the experimentation, we report (our translation of) some significant student answers to the question *'Did [having you try to solve the lotto problem before explaining arrays] help you to learn arrays, or did it hinder you? Why?'*: (i) "Yes, it helped me because it was a clear example"; (ii) "It rocked, it was very good and helpful"; (iii) "The lotto exercise served to demonstrate right from the start the usefulness of the array in that kind of exercise"; (iv) "It did not hinder me at all... in fact, I think it was useful to me". We like to report that out of 23 responses received to this question, as many as 18 were positive or highly positive.

10.3.6 Future works: the fourth phase

The design and implementation of experimentation are the second and third phases of our research project, whose original ambition (see also chapter 8) is to mitigate the problem of introductory programming, which is described and discussed in 1. The first phase was the research and development of our *Necessity Learning Design*, extensively reported and discussed in the previous chapter (see 9). As anticipated several times throughout the chapter, the fourth phase of this project will be a rigorous analysis of all the quantitative and qualitative data to understand NLD better, provide even more timely recommendations and possibly evolve it.

In particular, the quantitative analysis of the learning assessment results will allow us to understand whether using NLD to introduce arrays had an impact (positive or negative) on early learning related to arrays (as described in 10.1.2.2).

The quantitative analysis of the self-assessment on the *State Motivation Scale*, cross-referenced with the analysis of (our adaptation of) the *Cognitive Learning/Learning Loss*

Measure, will allow us to more analytically measure the impact of using NLD on novice students' motivation and perceptions about learning programming (as described in 10.1.2.3).

Finally, a more rigorous qualitative analysis (than the preliminary one described here) of the students' open-ended answers and our journal will give us an even better understanding of NLD, its applications, limitations and possible evolutions. However, we do not expect to discover anything remarkably different from what emerged from the preliminary analysis of the same data reported in the previous subsections of this section.

Chapter 11

The Online Course Was Great: I Would Attend It Face-to-Face

This chapter reports our teaching and research experience on "The Good, the Bad and the Ugly"[1] of *information technology* in emergency remote teaching of CS1. We describe how we redesigned, because of the 2020 COVID-19 pandemic, the CS1 course for math undergraduates to be held online yet reflecting the face-to-face experience as much as possible. We present the course structure, the IT tools we used, and the strategies we implemented to preserve the benefits of a synchronous experience in learning introductory programming. We discuss the positive and negative aspects that emerged from the students' opinions through qualitative analysis from the perspective of the challenge of providing optimally guided constructivist instruction in a remote-only CS1. We use the COI framework as a lens to explain what worked, what did not, and what can be improved to strengthen the perception of a face-to-face experience and mitigate the "presence paradox" we found. Despite students being enthusiastic about the online format, most would still prefer a face-to-face course.

This chapter is based on the poster "*The Good, The Bad, and The Ugly of a Synchronous Online CS1*" [Sbaraglia et al., 2021b] – published in the proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '21) – and even more on the article "*The Online Course Was Great: I Would Attend It Face-to-Face: The Good, The Bad, and the Ugly of IT in Emergency Remote Teaching of CS1*" [Lodi et al., 2021b] – published in the proceedings of the ACM Conference on Information Technology for Social Good (GoodIT '21).

11.1 Introduction

One of the challenges of the modern university is the ability to match the skills needs expressed by the labor market while maintaining its original trait as *universitas*. That is, teachers and learners collaborate in knowledge production, in that never-ending dynamics where the teacher's experience and students' fresh energy make any course instance both unique and a

[1] The reference is to the famous 1966 film "The Good, the Bad and the Ugly" 'by Sergio Leone.

building block of the democratic society[2]. One of the main ingredients of this *universitas* is the lecture, where students' active participation (with questions, comments, and different opinions) orientates the (usually standard) material towards a non-standard organization of the subject matter. A lecture delivered *and* attended at the same moment (synchronously) is when the interaction between students and instructors also produces new material on the spot[3].

We consistently maintained this view over the years, even for an introductory, technical course such as "CS1 for Math majors". Nonetheless, we were confronted with the COVID-19 emergency, which forced us to move online all the teaching on short notice. Well aware that it is not possible to simply transpose online a course designed to be *face to face* (also said *in person*), we used the one week we had before the beginning of the lessons for a drastic redesign. Even though delivering an online *synchronous* course can be challenging, we were determined to provide a remote-only deployment – including all lab work – that could stand the comparison with a face-to-face course by preserving teachers' presence and students' participation.

Having feedback on remotely connected students' engagement and understanding was much harder. The unforeseen emergency made it infeasible to set up quantitative, experimental research; instead, we choose a qualitative approach that is particularly suited when "*we may not fully understand a phenomenon – or even what the important phenomena are in a situation*" [Tenenberg, 2019, p. 172].

Concretely, in this work, we address the following questions. (RQ1) Could *consumer technologies* be successfully used to move online a face-to-face CS1 course on short notice? (RQ2) What are the effects of our design choices – made to preserve the advantages of a face-to-face experience – on the students' experience? (RQ3) What worked, what did not, and what can be improved to provide a fulfilling online synchronous experience?

See section 2.8 of the literature review (part I) for a review of the literature on remote teaching and learning in CS1 (and, more generally, in introductive programming courses).

Section 11.2 of this chapter presents the specific CS1 course whose experience we report and discuss in the chapter, the technologies we used, and our role as teachers and researchers. Section 11.3 describes the main methodologies adopted for the research, the characteristics of the participating students, and the COI framework (found during data analysis) that helped us interpret the collected data and shape our reflections. Section 11.4 reports the most relevant findings emerging from the analysis of our remote CS1, organized in themes (which emerged from the inductive categorization of the collected data). In particular, we were surprised by the insights that emerged about time management during labs (here) and the presence paradox we found (here). Section 11.5 discusses these findings (and the course experience in general), their usefulness (and limitations) for CS1 educators, and frames this work in the research for *optimally guided constructivist instruction*. Section 11.6 reports a brief conclusion and outlines possible scenarios for future research.

[2]We cannot argue here on this vision, which goes back to Humboldt, and has been elaborated by J.H. Newman, Jaspers, Heidegger, or Habermas, among the many others.

[3]It is worth citing the "conversational framework" from Laurillard [2002], which argues learning is "*a continuing iterative dialogue between teacher and student*".

11.2 Context

At the University of Bologna, CS1 for math is a mandatory course for first-year students in mathematics. It is an introduction to programming in Python with no prerequisites.

As for most CS1 courses, its goal is to teach, in an integrated way, both programming skills and their linguistic expression in the chosen programming language. Students should fully understand local and global scopes, aliasing, and side effects and develop a simple but accurate (albeit not complete) model of a Python abstract machine. Emphasis is placed on a good programming style. CS1 for math is 80 hours, 30 of which are supervised lab sessions (1-2 times per week) delivered in four 2-hours slots per week. On each lab, homework is assigned and is due at the beginning of the following lab. The course is offered once a year by four instructors: one professor and three teaching assistant (TAs). Enrollment is around 180 students.

A significant portion of students finds the course difficult because it deviates from the other courses offered for math majors, being more experimental and without "definitions, lemmas and theorems".

In the previous years, CS1 had a traditional organization – formal lectures at the hand-written blackboard, supported, when needed, by the projection of a programming IDE; BYOD (*Bring Your Own Device*) lab work with pair-programming, with no replications.

CS1 was about to start when the first wave of COVID-19 struck Italy at the end of February 2020. All instructors were asked to move online the first lectures and deploy the remote-only classes on a week's notice. CS1 started on March 2nd, 2020, broadcast from the instructors' homes. While the course contents remained essentially unchanged, the organization was radically rebuilt to support fully remote and synchronous teaching, including all labs.

During online lectures, attending students varied between 180 and 200 (with a decrease to around 160-170 during the last two weeks of lectures).

11.2.1 Technologies and Methodologies

Lectures and labs were broadcast using *MS Teams*, a video conferencing tool used as a standard at our university, plus Moodle-based platforms to disseminate the learning resources. Teams allowed for integrated broadcast[4] of audio, video, the shared instructor's screen, and for a public chat. We used a private *Telegram* chat for real-time synchronization between the instructors, especially during labs.

During lectures and labs, instructors always encouraged students to ask general questions (or to comment) as they liked, either on audio, by writing in the chat, or by publicly sharing their screen. We felt the need to structure the interaction explicitly: rules that are natural face to face – both because of non-verbal communication and students' previous experience with face-to-face lectures – must be precisely defined in online settings. Therefore, we iteratively

[4]Broadcasting from home, instructors used the laptop's built-in camera to capture the instructor's head and maintain a "postage-stamp"-sized video feed, more to enhance the sense of connectedness than as a tool for content transmission [Lidstone and Shield, 2010].

built and shared explicit rules and hints on how to interact with instructors during lectures, labs, and asynchronously.

11.2.1.1 Lectures

The primary instructor shared a screen divided into two halves. The left part was a *MS PowerPoint* canvas, where the instructor would type as on a blackboard. The canvas could be initially empty or present some content (e.g., snippets of code, short titles, or brief enumerations), which was *not enough* for understanding the subject. During the lecture, the canvas evolved into a more self-contained (though not complete) resource, later uploaded to the Moodle platform, together with any Python code shown or constructed during the lecture, for offline use. The right part of the screen was a window of a Python IDE, where code could be presented and run as needed. We used Thonny [Annamaa, 2015] because it is easy to install and use for moderately complex programs, consistent across different OSs, and has extensive logging capabilities. Despite Thonny's debugging facilities, the instructor insisted, instead, on using the online tool *Python Tutor* [Guo, 2013], which allows for a visualization of the internal state evolution, especially helpful with mutable values. A browser window with Python Tutor would replace one of the two halves of the screen when required. This arrangement was used consistently during all the lectures.

The instructor used a graphic tablet and digital ink software to make notes or handwritten drawings on the screen. Such annotations were later integrated into the MS PowerPoint canvas. The primary instructor kept the live chat of the course on a second non-shared screen. Typically, all the instructors used an additional device for their private synchronization chat.

The professor held the lectures. The teaching assistants were always present during the theory classes, giving support mainly in three ways. (i) In the course chat, they answered about materials and organizational issues and also provided answers to trivial questions about class topics. (ii) In the private instructors' chat, they report to the professor any student questions he had missed while conducting the lesson, mitigating the *instructor's blindness* and ensuring that no student felt ignored nor that any relevant issue remained unaddressed. (iii) At the same time, they summarized or rephrased important concepts live in the course chat to ensure that no one would miss any crucial element, emphasizing the importance of a topic and supporting understanding of a challenging concept.

Students asked their questions mainly in chat. After the lesson, instructors remained online for some minutes to answer more questions or discuss with the students. Occasional email exchanges occurred during the course.

11.2.1.2 Labs

In our view, one of the advantages of a *synchronous* approach is to bring instructors' experience and guidance in crucial learning moments like application and exploration. Therefore, we accepted the challenge of keeping all the labs as synchronous activities. Laboratory lessons were given by one of the teaching assistants, in turn, and were also attended by the other instructors. During labs, after a brief theory recap, programming exercises were assigned. A request for help in the public chat was followed by a private chat (or call, always using

Teams) between one of the instructors and the student. The student could share her screen with the instructor helping her. Students were encouraged to use Thonny for solving the in-class problems. Students had to upload homework assignments (simple programs) on a Moodle platform with the *CodeRunner* plugin for the automatic assessment through test cases [Lobb and Harlow, 2016]. Solutions to those exercises were discussed at the beginning of the following lab.

11.2.2 Teachers-researchers

The primary instructor is a senior professor of informatics with several years of experience in CS1 for math and consistently good feedback from students. The teaching assistants are either Ph.D. students or post-docs in informatics, all with a research interest in informatics education. After the lectures, the primary instructor and the TAs had detailed debriefings. The debriefings were very helpful in improving the course delivery, both because of the challenge of emergency online implementation and because the primary instructor does not usually have any colleagues observing the lectures.

11.3 Methods

Qualitative research is standard in social sciences and other disciplinary education research (such as math and physics), far less in informatics education research [Hazzan et al., 2006]. We follow the recent advice on qualitative methods for informatics education research [Tenenberg, 2019].

11.3.1 Data collection

At the course's beginning, we obtained informed consent from all participants, approved by the "Council of Mathematics Degree".

Halfway through the course, we organized a focus group with ten students, randomly selected among the most active – and so, we believed, more inclined to share comments and proposals. The suggestions from the focus group influenced some changes made in the course (especially in the organization of lab lectures: see section 11.4.3).

Moreover, the discussion brought us to design a questionnaire, submit a preliminary version to the focus group students, and then ask the whole class to fill in the final version.

The questionnaire was delivered as an anonymous Google Forms module. Three weeks after the last lecture, a message explaining motivations for helping instructors in the research was sent to all the 274 students who joined the online platform. Reminders were sent in the following weeks, and a link to the questionnaire was published in every online space related to the course.

We received 113 fillings. However, three were duplicated and discarded, leaving 110 fillings. We collected students' insights through the questionnaire via close-ended and open-ended questions and (anonymous) demographics.

Among all the questions (see Sbaraglia et al. [2021c] for an English translation), we focused on 15 open-ended ones related to the contrast between online and face to face, other broad

aspects of the course, and some specific but crucial elements of our 2020 implementation. We use a short tag to identify each question quickly. To those who had already attended previous years, we asked what they found better (WHAT_BETTER) and what was worse (WHAT_WORSE) this year. We asked all students what they found effective and ineffective both during theory lectures (THEORY_OK, THEORY_BAD) and labs (LAB_OK, LAB_BAD). We asked if they believe CS1 fits more than other courses to online education (CS_FIT_ONLINE) and then to motivate why they would choose to attend face-to-face (WHY_PRES) or online (WHY_ONLINE). We asked students why they would (WHY_SHARE) or would not (WHY_NOT_SHARE) share their screen during labs. We then asked about two specific aspects of our course: how to decide the time assigned to each lab exercise (LAB_TIMES) and whether they found having teaching assistants even during lectures helpful (TAs_THEORY). We asked for suggestions on how to encourage students to participate and ask for help (MORE_HELP) and general suggestions for improving the course (SUGGEST).

11.3.2 Participants

By chance, 55 students identified themselves as male and 55 as female. The students in the course were from 18 years old (the youngest, born in 2001) to 45 (the oldest, born in 1974). The median is 19 years old, representing the age of 69% of the students, followed by 20 years old, covering 13% of them. In our sample, 77 students did not have previous programming experience; 26 students studied programming in high school, and the others in different contexts (private courses, self-taught, and so on). Regarding the amount of theory and laboratory lessons they attended, from 0 (did not attend any) to 5 (attended all), 80% attended all lectures and labs (5), 10% almost all (4). In contrast, only a few students attended none (0).

11.3.3 Data analysis: inductive categorization with a grounded approach

Our coding involved the students' responses to each open-ended question on the questionnaire. Each question is identified by a label (see 11.3.1). For each question, we extracted categories that would reduce and systematize the variety of student responses as much as possible without losing important information in the coding process. We performed an inductive and iterative categorization [Tenenberg, 2019, p. 191] of the answers by identifying and assigning (*coding*) each filling of that question to *one or more* categories. In doing so, we decided to take an approach inspired by grounded theory by Strauss and Corbin [1998] to analyze qualitative data to produce new theoretical insights grounded in the data. A grounded theory approach emphasizes a systematic and rigorous data analysis process guided by the data rather than researchers' preconceived notions or hypotheses. Therefore our categories were not chosen *a priori* but constructed from the data in a grounded fashion.

First, note that multiple researchers engaged in collaborative coding. The four authors (see 11.2.1.1) worked together and always co-present in all coding stages, maintaining an open and peer discussion despite their different roles in the course (and university) and different experiences. Every decision was made together, discussing the various options aloud and always reaching a shared agreement. The co-presence at all coding stages and the

agreement on each decision were time-consuming. However, these realize a more constructivist approach to coding, closer to the spirit of grounded theory and less to the classical (more quantitative) approach to coding. This constructivist approach (substantiated by co-presence and agreement-building through peer discussion) also means that requirements of classical coding, such as measuring *inter-rater agreement*, did not have to be considered.

Based on the original formulation of Grounded Theory by Glaser and Strauss [1967], Strauss and Corbin [1998] propose three stages of coding for a grounded theory approach: *initial*, *focused*, and *theoretical coding*.

Initial coding

Initial coding is the first stage in a grounded theory approach to data analysis. In this stage, researchers examine the data and identify concepts, categories, and patterns that emerge from the data. Researchers start to break down the data into smaller units that can be analyzed more easily and begin to develop initial codes that capture the essence of the data. The goal of initial coding is to identify and label the various elements of the data while remaining open to the possibility of discovering new concepts or categories that were not initially anticipated. This approach is often referred to as "open coding" because researchers are open to any potential patterns or themes that may emerge from the data. During initial coding, researchers may use a variety of techniques to identify concepts and categories, such as highlighting key phrases, making notes, or creating concept diagrams. In our initial coding, we divided each student response into autonomous semantic units, when more than one, and assigned each unit a preliminary label summarizing its meaning. Overall, initial coding is a critical stage in a grounded theory approach to data analysis, as it lays the foundation for all subsequent coding and analysis. It allows researchers to identify the most important concepts and categories in the data and to begin to develop an understanding of the underlying patterns and themes. Once the initial coding is complete, researchers will have a set of codes describing the data's various elements. These codes can then be used to develop more focused categories in the following coding stage, referred to as focused coding.

Focused coding

Focused coding is the second stage of coding in grounded theory, following initial coding, and it involves a more detailed examination of the data related to a select few core categories identified during the first coding stage. Focused coding is used to refine the categories and subcategories generated in the open coding stage and to develop a more detailed and nuanced understanding of the relationships between categories. The goal of focused coding is to identify the most important categories to focus on in the theoretical coding stage. By selecting a few core categories to focus on during focused coding, researchers can identify the most important themes and patterns in the data and develop a more coherent and comprehensive theory. Specifically, we returned to the students' open-ended responses and the preliminary labels we had assigned. We tried to abstract more, defining new, more general, and broad labels that could contain several of them to uniform the landscape of responses without losing too much detail. This process of abstraction was also based on recognizing some recurring

themes in the responses to each question, recognizing that labels from the first stage could be subsumed into more general labels that summarized those specific themes. In addition to identifying relationships between categories, focused coding may involve identifying the properties and dimensions of categories and comparing and contrasting different categories to identify similarities and differences. In particular, for each question, we considered the dimension of each label (counting how many answers fell under that label) to recognize the most recurring themes in students' responses. Overall, focused coding is an essential part of the grounded theory process, as it allows researchers to identify and refine the most important categories in the data and to develop a more detailed and comprehensive understanding of the importance of those categories.

Theoretical coding and COI as a unifying framework

In the final stage of coding, *theoretical coding*, researchers identify the core category or categories that tie all of the other categories together. Researchers start to develop a theoretical framework explaining the relationships between the categories and the core category. Researchers also begin to look for ways to extend the theory and to make connections to other theories and research in the field. In particular, we found that the Community of Inquiry (COI) framework [Garrison et al., 1999] could help make sense of our focused categories and preliminary insights. COI describes the essential elements (called presences) of a successful online higher education: cognitive presence (i.e., construction of knowledge through discourse and reflection), teaching presence (i.e., design, facilitation, and direction of learning processes), and social presence (i.e., learners' ability to feel affectively connected with peers). The results of our theoretical coding, through the unifying lens of COI, are given in the next section. Often, we report excerpts from students' answers to "*provide a "prototypical" semantic unit that illustrates, concretizes, and in this way represents the entire category*" and support the trustworthiness of our categorization [Tenenberg, 2019, p. 186,199].

11.4 Findings

We analyze and discuss the most relevant aspects that emerged from the day-to-day work and analysis of students' opinions.

11.4.1 Individual assistance and live tutoring

To provide the individual support that students usually get during in-person laboratory sessions, we designed a simple interaction protocol to ask for assistance. Beyond the help in overcoming programming difficulties, we wanted to make students feel less isolated and more connected with instructors. A student had just to ask for help in the course chat, and the first instructor available would "Like" that message to let the other instructors know that the request had been taken care of. Then, the instructor would send a private message to the student, initiating individual assistance.

In the question on what worked during the labs (LAB_OK), 43% of respondents (32 out of 74) praised the assistance given by the TAs (being always present, competent, supportive,

and different in their style). At the same time, there is no mention of the teaching assistants' assistance in the symmetrical question on what did not work (LAB_BAD).

The teaching assistants' presence during the theory classes (see 11.2.1.1) was highly appreciated: in the specific question (TAS_THEORY), 92% of respondents (101 out of 110) found the TAs' presence useful or very useful.

In summary, the teaching assistants' availability to provide support[5], their number, their summaries, and re-elaborations of crucial concepts, as well as their helpfulness and closeness, were much appreciated.

We recognized that students perceived teaching assistants as both "deskmates" (filling the lack of face-to-face classmates) and an integral and competent part of the teacher's presence and support. This idea emerges from answers like *"The teaching assistants help the professor with the many questions since there is no deskmate to ask"*, *"The opportunity to ask for help for a specific doubt without having to interrupt the lesson and putting at ease the shyest people"*[6], *"The teaching assistants have always been friendly, helpful and competent"* and also *"They offer human contact with almost peers"*. This ambivalent perception about TAs is an example of how *teaching* and *social presence* could affect each other positively [Garrison et al., 2010].

11.4.2 Live-built materials, LMS and auto-grading

As described (see 11.2.1.1), slides were built or completed during lectures, based on interactions (both chat messages and voice interventions) with students, to promote active learning, foster interest, and highlight the importance of participation. Programs were written and executed live alongside the slides.

When asked about what they found effective in theory classes (THEORY_OK), students expressed positive feelings about the live construction of teaching materials. On a total of 82 answers, 52% of respondents liked one or more of these aspects: (i) live programming examples, (ii) live-built slides during the lecture, (iii) instructor handwriting on the shared screen. As a possible drawback, live-built slides cannot be available before the lesson, as few students requested in SUGGEST.

The Learning Management System (LMS) Moodle was useful for organizing materials across lessons, and uploading slides, programs' code, and homework after every class. We used the CodeRunner plugin to enrich the assignments with automatic tests and grading. The results of the multiple tests of each programming exercise provided students with progressive, specific information about their code. This strategy allowed us to give students constant feedback about their homework, which is otherwise impossible for just four instructors with more than 200 students. It is worth noting that providing an adequate number of auto-graded

[5] This finds evidence in literature: from Bowers' review [Bower et al., 2014, p. 16], it is crucial to hire teaching assistants to respond to text chat, managing issues not related to core aspects of the lesson. Moreover, increasing the ratio of TAs to participants helps minimize disruption and "can also lead to a richer learning experience for students".

[6] Other works confirm this. For Bower et al. [2014, p. 15], "*students who have the choice of attending face-to-face or remotely, often choose to participate remotely [...] because they can unobtrusively contribute to the lecture discussion via text chat*".

exercises each week took much time, effort, and precision.

Results show 44 positive answers across four questions. The most relevant categories are: 'Materials available online' (17 in THEORY_OK), 'Home assignments with automatic tests' (12 in LAB_OK), and 'Solutions available online' (4 in LAB_OK).

11.4.3 Time management in labs

In redesigning the laboratory routine, we initially decided not to allot prefixed times for autonomous activities. First, we believed that prefixed times, established by instructors and equal for all, were not inclusive. Second – coherently with our premise – we also wanted the laboratory classes to evolve from participants' contributions. Therefore we devised an *ad hoc* interaction protocol. For every autonomous activity, the first student completing it should write "Done" on the course chat, and the coursemates that followed should just "Like" that message. For each activity, depending on its difficulty, the instructors would evaluate how many "Done" were sufficient to end the autonomous work and start the discussion. This "quorum" mechanism gave us the (false) impression that we had the pulse of the situation, relying on quantitative information to assess better when to move forward.

Students in the focus group expressed tepidly about the quorum mechanism, and many of them said they would prefer a fixed time for every activity. At the time, we naively attributed this preference to the downtime experienced by skilled students. Nonetheless, we decided to test prefixed times for autonomous activities in the remaining laboratory lessons. In the questionnaire, we asked students about their preference and their motivation.

The preference for prefixed times (across questions FIXED_TIMES and LAB_BAD) is based on two opposite perspectives. The quorum mechanism displeased the more skilled students. 36 respondents perceived it as a waste of time (*non-inclusive vision*, contrasting ours). More surprisingly, it displeased the fragile students, too. Seventeen respondents perceived it as an anxious run-up to the execution speed of the best colleagues. Remarkably, according to 17 students, the instructor would know the ideal resolution time of each exercise, assuming an objective one exists. Communicating this univocal time would be the most "democratic" way to allow students to measure themselves against their limits and without looking at others (i.e., at the increasing number of "Done" in the course chat).

The strong preference for prefixed times and this latter misconception show that it is crucial to systematically share the didactic choices with students, especially in an online learning context[7]. Moreover, being able to count "Likes" inspired us with excessive and unfounded confidence and resulted in an abuse of the quorum mechanism that displeased most students. However, it remains an open problem to figure out and balance the different competence levels of such a large class.

11.4.4 Sharing the screen

During lab lessons, it was hard for the instructors to note if a student needed help. Contrary to what happens in the lecture hall, where instructors can look at students' screens, the only

[7]According to Peachey [2017, p. 150], online education "*need[s] to provide exceptional levels of student support*", by explicitly explaining the didactic relevance of the tasks.

way to understand if someone needed help was when they explicitly asked. We asked the students if, during labs, they would share their screen with *instructors only* (SHARE_SCR). The preference is clear: 85 (out of 110) would share the screen to get assistance, and 25 would not.

The main motivations of those in favor were (i) the opportunity of receiving more effective and even unsolicited assistance – e.g., when the student would not know what to ask for or is too shy to ask for help; (ii) the idea that sharing the screen is *"just the same as in the classroom"*.

Among the students opposed to screen sharing, the largest group fears for their privacy (*"I don't want to be observed while I could also mind my own business"*). As a possible solution, a screen-sharing system could warn students in advance that an instructor will look at their monitor, just as students in the classroom realize that instructors are approaching their station. Moreover, a system that allows sharing just the IDE should be used.

11.4.5 Presence paradox

One of the most interesting aspects that emerged from the analysis of students' opinions is the coexistence of two antithetical judgments on the course. First, the end-of-course questionnaire revealed a high level of satisfaction for almost all the students[8]. Moreover, by looking (in all the questions presented in section 11.3) for *explicit and strong* statements in favor of the course, we found that 64 out of 110 respondents highly valued the online course. For instance, when asked for suggestions to improve the course (SUGGEST), one student answered *"No, the course was perfect like that!"*.

However, when asked if they preferred distance or face-to-face learning if they had a choice, 68 chose face to face and 42 distance. In particular, half of those strongly in favor of the online mode replied that they would choose the face-to-face course. The reasons for preferring face to face are primarily related to the lack of interaction with instructors and peers. The reasons are either didactic (*"Being able to talk face to face with the teacher allows me to explain myself better"*) or socio-relational. On the other hand, those who choose the online mode reported logistical reasons[9].

We believe this "presence paradox"[10] is the effect of our effort to provide an online synchronous experience as rich as the face-to-face one.

[8] An external, university-level evaluation reported that 94% of students expressed high satisfaction with the course (N=165).

[9] Other works confirm it: reasons for choosing online courses are mainly practical – flexible schedules, costs, time, no need to commute (see, e.g., [Rudestam and Schoenholtz-Read, 2010; Bower et al., 2014; Bower, 2009; Joyner et al., 2020]).

[10] Paraphrasing the "Synchronicity Paradox" of Joyner et al. [2020]: students seem to desire synchronicity, despite being attracted to online courses mainly because of asynchronicity.

11.5 Discussion

Using a movie metaphor, we will discuss what we believe to be "The Good, the Bad and the Ugly"[11] of our adaptation of a face-to-face CS1 to be held entirely online.

The Good is that students highly appreciated the course. They praised those aspects that favored synchronous interactions (e.g., live-built slides, live programming examples, individual support from TAs during labs, and course chat interactions) and leveraged technology to mitigate online learning drawbacks (e.g., Tas' presence and support during theory lessons, a learning management system for sharing materials and automatic homework testing).

The Bad is mainly related to instructors' misconceptions (over-reliance on quantitative tools to track live the completion of exercises and manage lab times accordingly) and students' misconceptions (overconfidence in instructors' ability to help in any situation).

The Ugly concerns human aspects of the face-to-face experience not to be lost, like the instructors being able to see students' screens during labs and proactively help them (but with attention to privacy). Moreover, instructors need to be more explicit about didactic choices, which are more difficult for students to understand online. Finally, most of our students would choose a face-to-face course, especially for the unmediated social interactions (didactic and socio-relational) with instructors and peers.

Conclusively, the strong and appreciated *teaching presence* is one of the key factors – the most recurrent one in students' answers – related to high course satisfaction. At the same time, the lamented lack of social interactions is a direct symptom of a poor *social presence*. We hypothesize that this deficiency is the primary cause of students preferring face to face and also the cause of most misconceptions we found. First, the "isolation" of the students may have generated the wrong perception, hence the anxiety, that most of their coursemates had already finished the exercise (see 11.4.3). Also, the strong *teaching presence* not counterbalanced by a robust *social presence* may have generated the misconception of omniscient and omnipotent instructors, partly deresponsibilizing the students.

About *cognitive presence*, it is worth pointing out that an introductory and most technical course – whose primary goal is literacy in the basic programming and informatics concepts – does not focus on a critical analysis of knowledge. That said, our students positively received the strategies that could foster *cognitive presence* by favoring reflection (i.e., CodeRunner testing, live construction of materials, program reading and comprehension, time for questions and discussion of alternative solutions).

11.5.1 Online CS1 and the search for optimal guidance

Being the course remote only, especially not being able to see students' screens as they program, leads instructors – if no correctives are made – to the "minimal guidance" of the most extreme forms of constructivism (see 2.5.3.5). Despite being online, we wanted to find ways to tend as much as possible to the "*optimally guided constructivist instruction*" by Taber [2012]. In this sense, all the interventions documented and discussed in this chapter can be framed in this overarching effort. Adapting to our face-to-face CS1, these correctives intend to

[11] We refer to the 1966 movie "The Good, the Bad and the Ugly" by Sergio Leone.

prevent a solely online context from being minimally guided by reducing *instructor's blindness* (11.2.1.1) and (we discovered in a grounded fashion) fostering the three COI presences.

11.5.2 Validity and Limitations

The validity of this qualitative research is inevitably tied to the *trustworthiness* of the presented analysis. We, four authors, all informaticians with experience in education, actively coded and discussed all the open-ended answers together (see 11.3.3). Provided examples are representative of the kind of answers we received and coded. The large sample forms a solid base for the analysis.

In qualitative research, the primary goal is not generalizability but gaining a deep understanding of specific phenomena in context. This vision is widely accepted, as evidenced by the leading reference publications in the research field, which articulately motivate how qualitative research provides a unique and necessary way of knowing and experiencing the world [see, e.g., Lewis, 2015; Denzin and Lincoln, 2017]. Qualitative research is beneficial for understanding complex social phenomena challenging to measure or quantify, such as attitudes, beliefs, and social interactions. Consequently, it is an essential way to investigate educational contexts. Specifically, we aimed at "*making explicit the anomalies, problems, and contradictions*" [Tenenberg, 2019, p. 179] in a specific situation, like the COVID-19 pandemic.

Finally, an obvious limitation is the *social desirability* of responding positively to a questionnaire provided by the course instructors themselves. We tried to mitigate this by opting for anonymity.

11.6 Conclusions and Future Works

We presented how *consumer* information technology and tools could be assembled to successfully move online a face-to-face CS1 course, thus positively answering our (RQ1). In particular, we evaluated the impact of the technologies against students' perception using the COI framework – (RQ2) and (RQ3). We maintained that the online, synchronous experience must reflect the face-to-face experience – although not necessarily with the same tools, methodologies, and behaviors. While we managed to preserve *teaching presence*, the *presence paradox* indicates that improvements are still necessary, mainly to help students experience *social presence* too, even online. For example, we could foster casual interactions in the course meeting room while waiting for the lesson, facilitate social connections with more structured activities (like remote pair programming), or introduce homework peer correction. We plan to introduce and evaluate such activities in future implementations of CS1 for math majors.

Chapter 12

Castle and Stairs to Learn Iteration: UMC Co-design

This chapter presents a participatory process that involved primary school teachers and informatics education researchers. The objective of the process was to co-design a learning module based on the *Use-Modify-Create* methodology to teach iteration to second graders using a visual programming environment. The co-designed learning module was piloted with three second-grade classes. We experienced that sharing and reconciling the different perspectives of researchers and teachers was doubly effective. On the one hand, it improved the quality of the resulting learning module; on the other hand, it constituted a very significant professional development opportunity for both teachers and researchers. We describe the co-designed learning module, discuss the most significant hinges in the process that led to such a product, and reflect on the lessons learned.

This chapter is based mainly on the article (yet to be published) "*Castle and Stairs to Learn Iteration: Co-designing a UMC Learning Module with Teachers*" by Capecchi et al. [2023].

12.1 Introduction

Despite the critical role it has recently gained in public debate and policy, informatics education in primary school is still not compulsory in our country. Moreover, primary school teachers' background does not usually include any specific education in informatics or informatics education. However, several initiatives – often promoted by universities, associations, or teacher networks – have been launched that offer teaching materials, learning platforms, suggestions to schools and teachers on curricula, and professional development opportunities. Among those initiatives, two are particularly relevant to the experience we report in this paper.

- Nardelli et al. [2017] developed a proposal for a national informatics curriculum (grades 1 to 10) in Italy, developed by the national association of informatics departments and informatics/informatics engineering professors.

- "Programma Il Futuro", the national localization of Code.org [Code.org, nd], provides a support website and training initiatives for teachers, with over 3 million students and 41,000 teachers involved since 2014 [Corradini et al., 2017a].

Following up on these initiatives, a national project was recently funded (by a research grant to Italy's CINI National Lab "Informatica e Scuola") to disseminate programming education in the early grades of Italian primary school. This project aims to investigate the effectiveness and feasibility of different pedagogical approaches to teaching programming in the school context of our country, focusing in particular on iteration – a fundamental concept in introductory programming. The topic, target students, methodology, and programming environment for the learning module were established in the context of the general national project (see 12.2). In particular, the Code.org platform was chosen because it is already well known to teachers in our country through the popular "Programma Il Futuro" initiative [Corradini et al., 2017a].

As a preliminary step of the project, a learning module for teaching iteration to second graders was developed based on the Use-Modify-Create (UMC) methodology [Lee et al., 2011]. UMC engages young learners through a three-stage progression: first, they use a computing artifact created by others, then they modify it, and finally, iteratively, they approach the creation of their own artifact.

A *participatory process* involving four informatics education researchers (and authors of this report) from three universities, four primary school teachers, and about 60 students was conducted (by us researchers) to develop the learning module. This design approach[1] actively involves stakeholders in the design process to ensure that its outcomes are usable and meet their needs [Bødker et al., 2000].

This chapter focuses on the *participatory process* we carried out with the goal of co-designing with teachers (parts of) the learning material for the national project (i.e., the materials for teaching iteration using the UMC approach to grade 2 students). We describe the participatory process with the teachers and analyze how such a process positively influenced the outcome (i.e., the co-designed learning module). We highlight the positive effects of the participatory process (e.g., the materials that teachers were able – and willing – to use autonomously, valuable training moments for teachers and researchers) and discuss those that can be improved (e.g., more structuring and facilitation, better-defined roles in the participatory process). As was the case for this experience, we believe that the participatory design methods can be fruitfully applied to the design of learning materials. We hope the insights we offer promote wider adoption of such methods by the informatics education community.

See section 2.9 of the literature review (part I) for a review of the relevant literature on participatory design and section 2.6.4 (part I) for UMC-related teaching methodologies.

The chapter is organized as follows. The following section 12.2 describes briefly the national project, which is the general context for the research work we present in this chapter. Section 12.3 describes the learning module developed during the participatory process and the related material. Section 12.4 discusses the most significant hinges in the co-design process. Finally, Section 12.5 reflects on the lesson learned.

[1] Also called *co-design*, it was initially developed in urban planning but also applied in other fields such as software development and product design Bødker et al. [2000].

12.2 General context

We investigated the efficacy of two alternative instructional methods to scaffold the learning of iterations for young students in grades 2-3 in a project partially financed by research grant PANN20 00690 to Italy's CINI National Lab "Informatica e Scuola". The project is jointly conducted by eight university groups from throughout the nation in two phases that took place in 2022. Teachers' feedback collected across the two rounds helped fine-tune the deployment of the interventions. The trial results demonstrate measurable short-term outcomes variations between the two interventions.

This section is based on the poster *Learning Iteration for Grades 2-3: Puzzles vs. UMC in Code.org* [Nardelli et al., 2023], in press in the proceedings of the 54th ACM Technical Symposium on Computer Science Education (SIGCSE '23).

12.2.1 Project research goals

In order to carry out the project, we recruited 125 primary school teachers in two rounds, aligning them to the design of the two learning modules, both focused on the idea of iteration using block-based programming and aimed at students in grades 2-3. The project's goal was to compare how well those two learning variants performed in terms of the students' measured effectiveness and every participant's perceived satisfaction. Variant V1 required children to use and modify projects created for them within the Code.org Artist (Pre-Reader) lab [Code.org, 2022] before they could create their own projects in the same environment. Variant V1 used the Use-Modify-Create (UMC) approach [Lee et al., 2011]. Variant V2 used a standard set of puzzle-style coding exercises from the Code.org platform and had a more rigid structure. With regard to duration, expected outcomes, and evaluation criteria, the two learning variants were isomorphic.

12.2.2 Overall approach

The project was completed in two phases, each lasting three weeks. The first round occurred in the spring of 2022, and the second in the fall. The only difference between the two rounds was that the second round's deployment was improved thanks to the first round's lessons learned. Several hundred primary school teachers from all over the country were invited to the project. We recruited 22 of them for the first round and another 93 for the second round. The Fall semester worked better for the teachers' schedules. We divided the teachers into two groups, one per variant, nearly equal in size, and we paid attention to balance provenance and professional profile. The V1 learning variant (UMC) was given to one group, and the V2 variant (standard Code.org) to the other. All teachers in both groups: a) used two 1-hour lessons from Code.org to align their students on sequences; b) administered an identical pre-test to assess the children's comprehension of sequences; c) taught the idea of iteration following the group-specific learning variant in two to four 1-hour lessons; d) administered an identical questionnaire to collect the children's satisfaction with the activities and an identical post-test to evaluate their understanding of iteration (concept and use); e) filled out an evaluation survey on their own experience.

12.2.3 Project preliminary findings

In the first round of the experimentation, a total of 184 students (87, V1 – 97, V2) participated in all the activities carried out by 13 (of the 22 recruited) teachers. In the second round, the entire program was completed by all 93 recruited teachers for 1250 students (624, V1 – 626, V2). In total, 1434 students participated in the experimentation (711, V1 – 723, V2).

The children-side responses from the first round's post-project analyses reveal measurable V1-to-V2 differences in a few hotspots. The V1 group (UMC) felt slightly more fatigued by the learning effort, had more trouble understanding the code shown in two pre-test questions, performed worse in two post-test questions (a counted iteration of a single instruction and a counted iteration of two instructions), and better in one (a sequence of two counted iterations).

The second round of responses is currently being analyzed. The program was very engaging for all the students (between the ages of 8 and 12) and the teachers.

12.3 The learning module

This section presents the learning module resulting from the co-design process. All material, translated in English, is available at `https://codesignumc.web.app/` [Capecchi et al., 2022]. Section 12.4 analyzes how the process influenced the outcome described here.

The learning module aims at introducing second graders to the programming iteration construct in a graphical programming environment – namely in Code.org Studio – using the Use-Modify-Create methodology. The module's learning goals are consistent with promoting the development of skills described in the "Proposal for a national Informatics curriculum in the Italian school" that suggests that pupils should be able "*to use loops to concisely express that a certain action has to be repeatedly executed a prefixed number of times*" [Nardelli et al., 2017, p. 3].

As prior knowledge, we assume that pupils can build programs formed by a sequence of instructions, knowing that such instructions will be executed in the order they occur in the program. We also assume that they are familiar with Code.org Studio and the blocks of *Artist (Pre-Reader) lab* [Code.org, 2022]. The prerequisites and learning outcomes for the learning module, with reference to the CSTA K–12 CS Standards [CSTA, 2017] and ANON-PROP, are better articulated in Capecchi et al. [2022]. Together with the topic (i.e., iteration), and the context (second grade), several design constraints had already been determined by the overall design of the general project (see 12.2). First, the learning module would be structured according to the Use-Modify-Create (UMC) methodology (see section 2.6.4). Second, the learning module would be based on the Code.org Studio platform [Code.org, nd]; this was chosen as it is also the platform used by "Programma il Futuro", hence it is already known by a relevant portion of Italian school teachers [Corradini et al., 2017a].

We took inspiration from other publicly available materials, such as UChicago STEM Education [nd] and ECforALL [nd]. Even if they align with our goals, we could not simply translate them into our spoken language for two main reasons. First, they are very "Scratch-centric" (whereas our constraints required us to use the Code.org, see section 12.3), and

secondly, the storytelling needed adaptation to our country's cultural context.

12.3.1 Prerequisites and learning objectives

The "Proposal for a national Informatics curriculum in the Italian school" suggests that pupils should be able "*to use loops to concisely express that a certain action has to be repeatedly executed a prefixed number of times*" [Nardelli et al., 2017, p. 3]. Other proposals' objectives align with our activity, for example, pupils being able "*to understand that difficult problems can be solved by breaking them down in smaller parts*", "*to order the sequence of instructions correctly*", and "*to recognize that a sequence of instructions can be considered as a single action, which can be repeated or selected*" [Nardelli et al., 2017, pp. 2-4].

In addition, the objectives of the Italian proposal for an informatics curriculum overlap strongly with CSTA K–12 CS Standards [CSTA, 2017] that we consider in the following. The general objective of our learning module aligns with objective 1A-AP-10 of the CSTA Standards (i.e., developing programs with sequences and simple loops to express ideas or address a problem). Other related objectives are 1A-AP-11 (breaking down the steps required to solve a problem into a precise sequence of instructions) and 1A-AP-14 (identifying and fixing errors in an algorithm or program that includes sequences and simple loops).

The module's expected learning outcomes can be articulated as follows. (i) Pupils know that the block `repeat` in a program determines the repeated execution of actions and that the role of the number in the block is to determine the number of repetitions. (ii) Pupils can build a program with the `repeat` block. (iii) Pupils can create a desired (simple) pattern (e.g., *stairs*, including a sequence of instructions in one `repeat` block). (iv) Pupils can predict the behavior of a program containing a `repeat` block. (v) Pupils understand the role of `repeat` blocks into programs . (vi) Pupils can distinguish the effects of different `repeat` blocks in the same program (built by themselves or others).

In addition, we assume the following prerequisites. (i) Sequence construct. Pupils can build programs formed by a sequence of instructions and know that such instructions will be executed in the order they occur in the program. (ii) Familiarity with Code.org Studio. Pupils can distinguish between the *play area* (where the program will run), the *toolbox area* containing the blocks available to build the program, and the *workspace* (where blocks are dragged to build the program); pupils can add/remove/combine blocks in the program area to create programs; pupils can run their programs. (iii) Familiarity with the blocks of *Artist (Pre-Reader) lab*[2] [Code.org, 2022]. Pupils know the semantics of `move` and `jump` blocks in programs that draw turtle-graphics pictures.

12.3.2 Classrooms activities

The learning module is designed according to the *Use-Modify-Create* principles and structure and consists of two 2-hour lessons. Three purposely prepared Code.org programs [Capecchi et al., 2022] are available for pupils to run, look into, and modify. The first of such programs

[2]Code.org also offers puzzles (usually grouped into *lessons* of around 10 puzzles, see, e.g., https://studio.code.org/s/course1/lessons/10) where the subset of available blocks is limited for the specific puzzle.

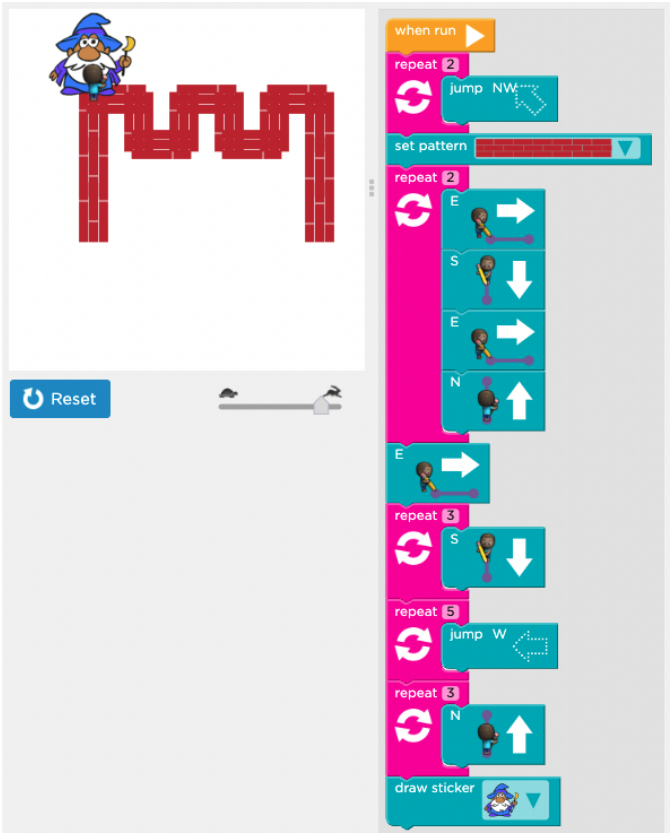


Figure 12.1: The program that draws the tower with the wizard.

is shown in Figure 12.1. In each lesson, tasks are proposed, and questions are asked that guide the pupils to become familiar with the programs, understand their behavior, modify them, and finally learn how to use the iteration construct.

The first lesson uses the first program and focuses mainly on the *Use* and *Modify* stages. In the *Use* stage, pupils do not have access to the program's source code. They are asked to run the program, observe its behavior and the effect of its execution (e.g., "what did the artist draw?"), and reflect on what happened (e.g., "which part of the drawing was drawn first?"). To answer the questions, they are invited to run the program as often as they like, possibly changing the execution speed.

In the *Modify* stage, pupils see the source code; this stage aims at making them read and explore the blocks of the program. We propose three types of *Modify* tasks.

(i) *Syntactical change*. Under direct instruction, pupils are asked to make a specified change in the program (e.g., move or replace a block, modify the parameter in the repeat block), then run it, observe, and verbalize the effect of the change on the execution and/or resulting drawing.

(ii) *Prediction*. The teacher describes a specific program change (as in the previous task),

and pupils are asked to predict how this change will affect the execution and/or resulting drawing before actually modifying the program.

(iii) *Intentional modification*. Pupils are given a simple, specific objective, such as modifying the program behavior or getting a drawing variation (e.g., "make the castle tower taller"), and are encouraged to take a trial-and-error approach. They can edit and run the program as often as they want; if necessary, they can discard their current program and start over from the original one.

In the first lesson, all the modifications concern the simplest loops, that is, those defined as a `repeat` block with only one instruction. At the end of the first lesson, pupils are invited to modify the program, for example, by adding characters or simple elements.

The second lesson uses two other programs, each producing a variant of the first drawing. For both programs, the *Use* stage is limited and aims only at getting pupils familiar with the new program. In contrast, the *Modify* stage focuses on loops (drawing the stairs) that contain two-instruction sequences. The second lesson then includes a substantial *Create* stage. Pupils are invited to make their own drawings by building a program that uses the `repeat` block. Some drawing ideas are made available to possibly inspire pupils who do not know what to draw or are stuck trying to realize drawings that are too complex and unfeasible.

12.3.3 Developed materials

12.3.3.1 Programs

Three Code.org programs were developed purposely for the learning module; see Capecchi et al. [2022]. They produce drawings of a castle tower with additional characters and stairs; the drawings are similar but obtained differently (e.g., starting from a different corner). Figure 12.1 shows the program proposed to pupils first.

The programs are built using the *Artist (Pre-Reader) lab* [Code.org, 2022]. Each contains multiple (five or six) `repeat` blocks; some loops are simple in that they include only one other block, while others are more complex. In particular, all programs include a `repeat` block containing a sequence of four movement blocks used to draw the tower battlements.

These programs are designed to be short enough to be managed by the pupils, rich enough to provide interesting, attractive behavior, and complex enough to challenge the pupils' understanding and cognitive processes.

12.3.3.2 Support slides

We prepared a series of slides Capecchi et al. [2022] to support teachers in conducting the lessons. The slides collect all the questions and tasks proposed to pupils and are to be displayed at the appropriate moment during the lessons. Slides mainly use visual language and include very little textual content; see, ss an example, Figure 12.2. Pupils work in pairs following the indications given orally by the teacher, with the visual support of the slides.

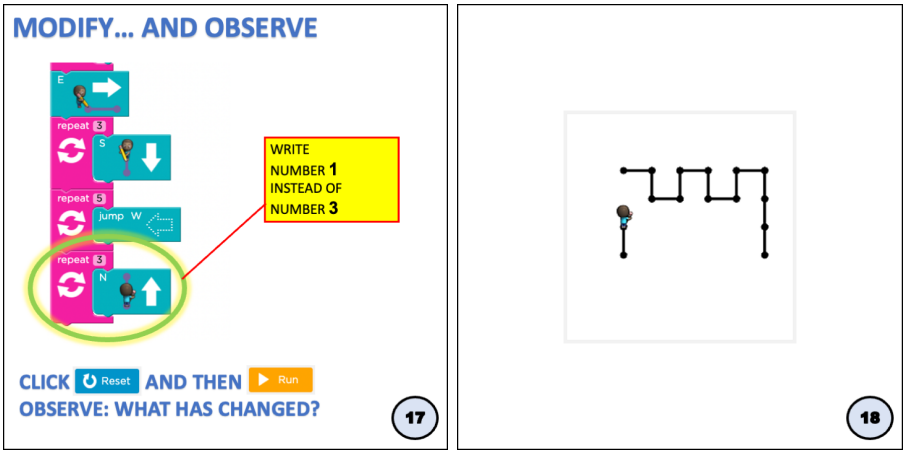


Figure 12.2: Examples of support slides. The slide on the right is used as checkpoint w.r.t. to the task posed in the slide on the left.

Besides the slides with questions and instructions, there are also some "checkpoint slides" whose purpose is to align the class group at critical moments (see the right-hand slide in Figure 12.2). For instance, some checkpoint slides show the effect of a specific change in the program. Generally, they allow pupils to verify that they got the same effect or help them catch up if they had difficulties understanding or executing the questions/instructions. The checkpoint slides also allow the teacher to discuss or clarify crucial points, starting with pupils' answers and comments. The use of checkpoint slides is intended to allow pupils to proceed at their own pace while maintaining an overall pace for the whole class. Hence it helps the class group and the teacher keep the focus on essential points, avoid dispersion and confusion, and guarantee a certain level of autonomy for the pupils.

Motivation for the slides. Pupils' learning pace can vary significantly, and for the above-described activities to be fruitful, they should be allowed to proceed at their own pace. However, asking them to read and understand a long sequence of written assignments and questions on their own while carrying out the related activities is not reasonable. Indeed, given their age, reading and understanding individual assignments or questions may be alone too complex a task. Therefore, we decided to make pupils work in pairs and follow the teacher's proposals, also being visually supported by the slides. On the one hand, this allows for some meaningful autonomy of pupils' pairs when addressing the proposed tasks. For example, if they have fallen a little behind and missed the spoken instruction, they can look at the slide and understand the next step to be carried out. On the other hand, it allows the teacher to manage the whole class group and compensate for the different pupils' paces.

12.3.3.3 Teacher's guide

The learning module's classroom activities are presented and commented on in a "Teacher's Guide" [Capecchi et al., 2022]. Although teachers could also use the guide as a reference

during the lessons, its purpose is to support teachers in preparing for the lessons.

The guide has a threefold function. First, it presents the module's pedagogical approach, illustrating the three stages of the *Use-Modify-Create* methodology. Second, it provides a detailed description (consistent with the slides' content) of the series of tasks/questions to be proposed to the pupils in the two lessons. Third, it states the intended purpose of each proposed tasks/questions, anticipates possible reactions, comments, doubts, and mistakes that pupils might manifest, pinpoints the critical role of some specific tasks/questions in the learning process, and frames the tasks/questions in a general learning context.

Hence, the guide aims to provide a solid background to support teachers in conducting the learning module.

12.4 Co-designing with teachers

This section describes the participatory process we conducted and how it affected its outcome, i.e., the learning module described above.

12.4.1 Phases of the participatory process

Three main phases can be identified in the process.

1. First, the group of researchers drafted some ideas for programs and tasks.
2. Then, a few selected primary school teachers were involved in discussing and revising such initial ideas.
3. Finally, these teachers in their classes piloted the resulting learning module and related material.

During the first phase, the group of researchers – the authors of this paper from three different universities – met online to draft some preliminary ideas of programs and tasks for the learning module, structured according to the UMC methodology. The programs were designed to be short enough for pupils to handle, rich enough to provide interesting and attractive behavior, and complex enough to challenge pupils' understanding and cognitive processes. Those preliminary materials were then informally tested with some children from the family environment of the researchers. The *Use* and *Modify* stages worked well, as tasks positively challenged children. However, the first program turned out to be too complex, and the wording of the questions was difficult to understand. We revised the materials accordingly, but we felt that our work would have benefited greatly from the contribution of primary school teachers that are experts in the specific target learning context and population.

Rather than simply asking for separate feedback on the material, for example, through an online questionnaire, we decided to promote a participatory process and invite a diverse group of primary school teachers to act as *informants* (see section 2.9). Our design ideas, still in preliminary form, would be submitted for open discussion to a diverse group of primary school teachers. We opened up for a comprehensive review and (possibly major) revision of

the material, where teachers would have the chance to interact with both the researchers and each other.

More precisely, we reached out to primary school teachers we had already worked with for other STEM projects. Four teachers from four different cities accepted our short-advice invitation to collaborate. They did not know each other; all had a strong motivation and long teaching experience; their background with respect to programming ranged from novice to expert, as well as their experience in teaching informatics. This collaboration lasted about two months and consisted of four online meetings (attended by both researchers and teachers), some intermediate meetings between researchers only, and continuous asynchronous work (by individual participants or small subgroups) on the shared documents. One of the researchers acted as a facilitator of the process.

This co-design phase ended with three of the four teachers piloting the learning module in their classes with the related material. Overall, about 60 pupils were involved in the pilots. We did not collect data on pupil performance or behavior, nor did the teachers use the learning experience to assess their pupils. The teachers were interviewed briefly after the completion of their pilots. Since the pilot tests were successful and did not reveal any particular problems, the learning module was finalized by implementing minor fixes.

All meetings with the teachers and the final interviews were conducted online and recorded using a videoconferencing tool. In addition, several documents were shared, collaboratively edited, and commented on during the process. In order to refine the activity and its materials (and write this report), we researchers watched the recordings and transcribed the most relevant passages, then reviewed all the transcripts, together with the comments and history of the shared documents. All the involved people were informed and agreed to be recorded for these purposes.

12.4.2 How the process affected the outcome

As mentioned earlier, the programs and tasks that compose the learning module presented in Section 12.3 are significantly different from those initially designed by the researchers and submitted to the teachers at the beginning of the participatory process. The comments and objections raised by the teachers led to fruitful discussions. They led either to confirmation and clarification of core ideas (e.g., the UMC structure, the cognitive progression of activities) or revision of some aspects related to the learning module's content or the presentation of the material.

We report some concrete episodes that we deem both meaningful for their impact on the process outcome and representative of the kind of interactions that occurred. We organize them under five main themes for the sake of readability, but it is worth mentioning that, as they are strongly interconnected, most of them occurred interleaved and in more than one meeting. Teachers and researchers are named respectively using the initials 'T' and 'R'.

Storytelling and visual aspects. Many comments focused on the setting for the programs to be proposed in the classroom, as the initial programs produced simple drawings (e.g., stairs, letters of the alphabet) unrelated to each other. For example, T3 noted the importance of

the storytelling and visual aspects, emphasizing that these also increase the effectiveness of the activities from a learning perspective; T2 pointed out the difficulty of devising programs with these visual and narrative enjoyable features and whose complexity is still manageable by young children. After several interactions, a good balance was found and the initial programs were dropped. They were replaced by three new programs (described in Section 12.3) with similar complexity that, however, refer to one single story with two characters (a wizard and a dragon) moving around a castle tower (see Figure 12.1).

Informatics content and relationship with other disciplines. When preparing the first draft of the learning material, the researchers mainly focused on its informatics content. Since the first interactions with teachers, it was clear that this focus would likely conflict with other points of view that do not concern informatics but are very relevant, especially in early education.

For instance, when learning to write the parts that make up a letter, children are encouraged to move the pencil in a prescribed sequence and direction. However, in one of our initial programs, the artist drew a letter following a purposely scrambled sequence and directions so that the relation between the program blocks and their effect would be less predictable. Although this trick seemed justifiable and even useful from the researchers' informatics perspective, it was challenged by T3 and finally deemed unacceptable by the teachers: they know that many second graders still have difficulty writing letters in the correct direction, so they should not be presented with contradictory examples. Similar observations were made by T1 and T4 regarding difficulties with laterality, as some tasks relied on the ability to distinguish right from left or similar topological relationships that may still be under acquisition for some second graders. These discussions also strongly affected the revision mentioned above of the overall setting of the programs.

Presentation of questions and tasks – wording and format. A large part of the discussion concerned how to present the tasks (including the questions to be answered) so that pupils would easily understand them. The researchers designed a sequence of tasks to propose to pupils and drafted a written document with preliminary notes describing those tasks. That document was intended as a tool for discussion; thus, we deliberately omitted to work on refining its wording. Instead, we explicitly left this matter to be explored in the co-design process and asked the teachers to help us shape the format and linguistic aspects of the tasks and questions.

Indeed, all teachers found in the text several hurdles for pupils, such as complex terms or long sentences. They also found imprecise or misleading wording and, while raising their doubts, offered the researchers the chance to clarify the intended purpose of the tasks from a cognitive point of view. This discussion was also very formative for the teachers, who had the opportunity to understand better some informatics aspects involved in both the programs and the questions.

As for the format, the initial version relied very much on verbal communication, the main question from the researchers being whether the tasks should be presented to pupils in written form (e.g., using a worksheet, as in Salac et al. [2020]) or explained orally by their teachers.

The teachers brought up a different approach that was more based on visual communication, as they observed that relying only on words is inappropriate for second graders. In particular, T3 suggested using colors and arrows to identify parts of the screen or portions of a program. To overcome R3 and R4's initial hesitation, T4 went further and prepared some slides to show how to present the tasks in a visual format, which convinced the group of the effectiveness of this approach. Some concepts were then formulated using a graphical notation instead of text. For instance, to answer the question "Where did the artist start to draw the castle?", colored quarters can be used instead of text options like *up*, *down*, *right* and *left*. As an example of the outcomes of these interactions, see the slide in Figure 12.2, where one can see that the role of text is marginal.

From then on, instead of acting only as an informant, T4 played a much more active role in preparing the slides and the teacher's guide, working closely with R1 and R2 as a *partner* in the process (see section 2.9). Figure 12.2 shows an example of the resulting slides.

Foster reflection in pupils. The teachers learned about the *Use-Modify-Create* methodology from the researchers and appreciated it. R2 emphasized that the approach allows for more complex programs, balancing the greater difficulty with increased involvement of pupils. T3 noticed that the *Create* stage motivates learners and linked the *Modify* stage to experiential learning, as it fosters reflection on doing (remember 2.4). T4 remarked that the approach accommodates different expertise levels, effectively engaging also students with previous extracurricular experience in coding. Several comments focused on the effectiveness of the UMC methodology in fostering reflection in pupils, whose *trial-and-error* style is often too impulsive and not very organized and principled. For instance, the tasks in the *Use* stage make pupils observe important details and reflect on what happened; the prediction questions in the *Modify* stage force them to pause to think and reason about the expected outcomes.

However, T4 noted that such reflective and open-ended questions (e.g., "What do you notice?") are difficult for young children because they require them to give explanations, which is a challenging task at their age. When R2 and R4 replied that the answer is not the essential part, as what is more relevant is thinking about the question, T4 insisted that keeping things practical and concrete is crucial from a learning perspective. A question that is not "answerable" is bad for both the pupil – who does not know how to deal with it – and the teacher – who cannot understand whether the pupil has grasped the point. This concern was addressed by rewording several questions, paying more attention to the cognitive progression of tasks, and adding checkpoint slides to help pupils check their reasoning without having to produce verbose explanations.

Class management and the teachers' role. Given the context of the national project, which aims to compare two ways of teaching iteration(see 12.2), we tried to provide teachers with maximum clarity. In particular, clear guidance on how the learning module should be conducted in the classroom or, in other words, on their role as teachers in managing the class group during the activities. In the first phase, however, the researchers deliberately omitted to define this role more precisely than just providing the tasks structured and organized according to the UMC methodology. On the contrary, this issue was explicitly left to be

explored in the co-design process, hoping to find a balance that would let each pupil explore and reflect on the tasks and questions while avoiding dispersion and confusion and keeping the group class focused on the crucial points.

The discussion was initiated by asking teachers how they would conduct the activities. Should pupils work individually at their own pace (as, for example, in Code.org lessons [Code.org, nd]), or should the activities be carried out together with the whole class? If explained verbally, would the questions be understood? Does it make sense to give second graders written questions to be read on their own, or should the questions be somehow mediated by the teachers? T3 noted that the answers to that questions did not depend only on the activities as such but on many other factors as well, including the characteristics of the pupils' group (e.g., listening skills, the habit of working on open-ended problems) and the teacher's personal style in conducting any activity. T4 and T3 agreed that all pupils should be offered the opportunity to explore, thus suggesting that work in pairs or small groups should be encouraged. R2 and R4 were concerned that the teachers might have difficulties managing the class because of different group paces.

At this point in the co-design process, we sketched out the following scheme of work. Pupils are set in pairs. Then, the teacher explains the following task to the whole class group using the related support slide. Pupils work, reflect, and discuss in pairs. After a while, the teacher asks the whole class what answers they gave and helps them discuss, trying to include all the pupils.

T2 objected that, following this approach, the discussions might take too much time and be annoying. Moreover, T1 pointed out that the resulting pace may not be respectful to the slower pupils. At this point, the idea of using checkpoint slides (see 12.3.3) as a tool to balance the pupils' freedom to follow their own pace while preserving the overall pace of the class arose. Checkpoint slides emerged as a way to align the class at the most critical moments of learning, to help pupils catch up if they had difficulties understanding or executing the questions and tasks, and to focus on and clarify the essential points (starting from the answers and comments from the pupils).

Moreover, T4 emphasized the importance for teachers to accurately understand the intended purpose of the proposed tasks and questions and to anticipate possible reactions, comments, doubts, and mistakes that pupils might manifest. The resulting discussion led to another significant decision. We agreed on the importance of providing a solid background to support teachers in conducting the learning module. Therefore, we decided that the teacher's guide would contain not only a detailed description of the series of tasks and questions but also some crucial reflections on their purpose and meaning from the informatics perspective. At the same time, the guide would also suggest to teachers possible adaptations of the learning module (e.g., different timings, additional details when proposing tasks, methods for promoting and leading discussion during activities) to take into account the characteristics of their class groups.

12.4.3 Possible improvements

Even though external constraints or lack of resources often hinder this, the best practices in participatory processes suggest that the process be planned thoroughly in advance, use

structured participatory methods, and be facilitated by external experts not involved in the design itself [Bødker et al., 2017].

In our case, we acknowledge that we turned to co-design only after a preliminary design phase; hence we did not plan the participatory process in detail, and we conducted it in a short time, with close meetings. Moreover, we did not have the time to organize preliminary forms of infrastructure and "backstage work" to create rapport and trust between participants [Bødker et al., 2017]. Therefore, we could only include in the process teachers with whom we were already in contact for collaborating on other informatics education-related projects. That previous knowledge acted as a "backstage work", though incomplete. Also, the process was facilitated informally, with one of the researchers acting as *facilitator*, based on her prior expertise in participatory design.

Despite this, participatory design principles were followed. In particular, we cared to create a context that genuinely allowed an open discussion and establish equitable power relations within the co-design group, as testified by the fact that T4 was unexpectedly promoted as a *partner* in the design process[3].

12.5 Conclusion

In this chapter, we reported the process of co-designing a learning module aimed at teaching iteration to second graders with the UMC methodology in a block-based programming environment. The effectiveness of the developed learning module for teaching iteration is compared with that of a more traditional approach (a standard set of puzzle-style coding exercises with a more rigid structure) as part of a national project involving 125 teachers and 1434 students. Consistently with what was reported by other UMC research [e.g., Lytle et al., 2019], the process successfully produced teaching material that the teachers felt comfortable experimenting with in their classes without the researchers' support.

The heterogeneous co-design context (including primary school teachers and informatics researchers) helped hold different educational needs together. The presence of specific diverse expertise ensured that the fundamental aspects and objectives of the learning experience and informatics content were correctly preserved, which would have been impossible without a homogeneous project team. For example, this allowed us to find a satisfying mediation regarding different – and sometimes contrasting – needs. For example, balancing the freedom in interpreting the teacher's guide to accommodate different class group characteristics while assuring a correct implementation of the activity; or balancing the activity's ease, economy, and length while making students focus on specific informatics aspects. The result is an educational product that maintains its informatics scientific soundness while being usable by teachers and appropriate for second-grade students.

The ample space granted to the teachers' peer discussion, the researchers' openness to listen and – when needed – heavily revise the work (while always being firm on crucial informatics concepts), and the teachers' generous participation in the process were crucial for the design's success.

[3]Even if, according to Coenraad et al. [2022], to put teachers in the role of *design partner*, specific approaches are needed to establish equitable power relations with the researchers' group.

As illustrated in the analysis, the co-design process has been very formative for both the teachers and researchers. Teachers had developed a better understanding of critical cognitive aspects of teaching iteration by diving into the material to understand it and looking for the best way to engage students. Researchers learned to better take into account pedagogical aspects not strictly related to teaching informatics. Most importantly, while keeping the original constructivist approach, we moved from a minimal guidance proposal to a more scaffolded one, creating materials that can help teachers provide pupils with the optimal guidance they need Taber [2012] (see also our 2.5.3.6 in part I).

Overall, we advocate for such participatory design methods as a helpful approach to both increase the quality of the resulting learning material and contribute to the professional development of the people involved. We also hope this report can contribute to illustrating how these participatory methods can be fruitfully applied in the development of learning material, thus promoting a wider adoption by the informatics education community.

Part IV

Original Contributions – Informatics for All

Introduction to part IV

In this second original part of the thesis, we present the contributions of our research on teaching informatics principles through unconventional pathways (such as exposing young students to the basic principles of cryptography) to convey its value as a necessary lens for understanding our society and as a set of principles, ways, and tools for acting in it. In an ideal scan (ideal since the reader will also be able to grasp intertwined references), this production is situated within the literature reviewed in part I.

Inspired by the big ideas of science and particularly the big ideas of informatics education, we report how we tried to take the same approach to teach cryptography in high school after recounting the research project from which this initiative originated. We tried to expose and teach the basic principles of cryptography, which have revolutionized our society and that we use daily. However, the purely cultural perspective did not neglect the minimum and necessary scientific and technical elements functional to understand a principle.

Then we present an activity for teaching public-key cryptography using graphs to help pre-service STEM teachers explore fundamental concepts and methods in informatics and mathematics. The activity was designed using the Didactical Engineering research methodology and the Theory of Didactical Situations. Participants were required to explore and understand concepts and methods from mathematics and informatics and move between different disciplines and semiotic registers.

Again, the teaching approach was a "balanced constructivism", adopting active and reflective activities as much as possible but always pursuing optimal guidance for adaptive scaffolding. The common goal of these research initiatives is to enable a broader audience (than those who choose to learn specifically informatics) to understand the importance of informatics in our lives and its interdisciplinary implications while increasing motivation to learn more about it and fostering the building of skills in the realm of computational thinking.

Chapter 13

SIGCSE Special Project on Cryptography

In 2020, we submitted a research proposal on cryptography education to SIGCSE for a 5,000$ *Special Projects Grant*.

> These grants help SIGCSE members investigate and introduce new ideas in the learning and teaching of computing. Projects must provide some clear benefit to the wider disciplinary community in the form of new knowledge, developing or sharing of a resource, or good practice in learning, teaching, or assessment. [SIGCSE Special Projects, [n.d.]]

At that time, we wanted to seize two opportunities: the SIGCSE special projects grant and the proposal of a local high school (a scientific lyceum) for a short course within the national project of *Liceo Matematico*[1].

Our research group was working on cryptography education, specifically on leveraging cryptography to train STEM disciplines teachers to foster interdisciplinary teaching of mathematics and informatics. Chapter 15 extensively discusses this research work, part of a massive European Erasmus K2+ project involving five European universities.

Working on cryptography education, we had been fascinated for some time by the idea of being able to distill its big ideas (see section 4). The reason was twofold. First, from a review of cybersecurity education literature, we found that cryptography was underrepresented. Second, in our teaching experience, we encountered widespread ignorance of the essential cryptography principles as a cultural lens for grasping many aspects of today's digital society.

That is the scenario in which we developed our proposal for cryptography education and submitted it to SIGCSE as a special project. At the end of 2020, our proposal was awarded one of three *Special Projects Grants* for that year. The following is a summary and a report of our project, which frames and motivates most of our research on cryptography education.

[1] *Liceo Matematico* offers volunteer lyceum students extracurricular activities to let them experience interdisciplinarity between mathematics and other disciplines and broaden their cultural horizons; more in 14.2.1

13.1 Background

In today's digital society, cryptography is at the core of many activities and tools (e.g., instant messaging, e-commerce, stock exchange, cryptocurrency). Various frameworks (e.g., DigComp [European Commission, 2017]) and curricula (e.g., CSTA K–12 CS Standards [K-12 CS Framework, 2016] and the UK computing curriculum [UK Department of Education, 2013]) include cybersecurity competencies. Some of them are more oriented on using security for personal purposes, others on understanding how digital security works, but they all recognize that cybersecurity skills are essential for students to be active citizens of digital society. Cryptography is one of the foundations of cybersecurity. In addition, novices identified cryptography "*as an interesting context for computer science lessons*" [Lindmeier and Mühling, 2020, p. 3]. pretertiary (or K-12) education does not aim to train professionals but to help students understand our world and act in it, therefore it is important to help them understand the principles of cryptography and their importance in our society. Much more on this in the review section, particularly in 6.1, 6.2.1 and 6.2.2.

13.2 Research activities

We designed a short course with no prerequisites, built around different types of activities. Since educational research has shown the effectiveness of active and cooperative learning methodologies [Loui and Borrego, 2019, p. 304], we designed non-traditional hands-on activities to be interactive and meaningful for students. We developed cryptography playgrounds for students to use, understand, and attack emblematic cryptosystems (e.g., Caesar cipher, One-time pad) and a "remote-unplugged" activity to enact the Diffie-Hellman key agreement in pairs. We realized both types of activities with Snap!, a block-based graphical programming language. Due to the ongoing COVID-19 pandemic, we had to design the first iteration of our intervention as remote-only.

Our pathway includes a few emblematic cryptographic systems and schemes, carefully selected as representatives of cryptography core ideas. With the aim to create a motivating progression, the introduction of a new scheme is always triggered by the *necessity* (which we stimulate in students; see [Sbaraglia et al., 2021a; Sbaraglia, 2021] and also chapters 9 and 8) to overcome the limitations of the previous one(s).

13.3 Project outputs

The project has two primary outputs.

1. A learning progression to teach fundamental cryptography ideas by making students encounter some representative cryptosystems (from classical to modern) and experience their limitations, and consequently, the *necessity* to overcome those limitations towards more secure systems. The cryptography learning progression is presented at `https://bigideascryptok12.bitbucket.io/#progression`, discussed in Lodi et al. [2022b] and in more detail in chapter 14.

2. Teaching materials for 4/5 lessons (suitable for high school students) following the path of point 1, consisting of

    - ad-hoc Snap! environments to experience firsthand how relevant cryptosystems work, their weaknesses, and possible attacks
    - unplugged activities (which can also be used in remote teaching): for example, a Diffie-Hellman key agreement simulation through color mixing (with the mixing based on the actual math of the protocol)
    - animated slides showing the high-level functioning of asymmetric encryption scenarios step by step and a narrative evaluation summary

All materials [Lodi et al., 2021a] are freely available and presented at https://bigideascryptok12.bitbucket.io/#playgrounds; they are also discussed in Lodi et al. [2022b] and in more detail in chapter 14.

13.4 Outcomes, ongoing and future research

The outcomes of cryptography learning progression, materials and their school implementations are presented in 14.3 and lengthy discussed in 14.4.

As mentioned, the grant-winning project became the framework of a broader line of research on cryptography education.

13.4.1 Cryptography big ideas

We interviewed experts in cryptography and informatics education with the aim of distilling the "big ideas of cryptography", sharing the core motivations, and following the example of the big ideas in science education and informatics education (see our review in 4). The developments of big ideas – for example, for pretertiary education in science [Harlen et al., 2015], informatics [Bell et al., 2018], and in artificial intelligence [Touretzky et al., 2019] – are ongoing processes; current proposals are sometimes very mature (e.g., for science and informatics education) but are not to be considered definitive.

It is useful to distill the fundamental concepts of a topic in a way that is understandable to students and teachers who are not necessarily experts in that particular field so that these fundamental ideas can serve as "beacons" to guide teaching and learning (more on this in 4).

While we are still analyzing the interview transcripts and planning more structured questionnaires to circulate among experts, many of the emerged ideas have been integrated into our learning progression (see 13.3 and more in detail in 14.2.4 and 14.2.5) built around representative cryptosystems to teach the fundamental ideas of cryptography.

13.4.2 Cryptography course implementations

We tested our course two more times. See chapter 14, particularly 14.2.2, 14.3 and 14.4.

13.5 Publications and Dissemination

An experience report paper [Lodi et al., 2022b] on the first iteration of the course has been accepted and presented at ITiCSE 2022 in Dublin.

In addition, our work was presented at the *2021 Cryptography and Coding Theory Conference*[2]. A short paper (in Italian) is published in the conference proceedings [Lodi et al., 2022a].

Finally, for *Rendiconti*[3], the international mathematics journal of the University of Turin, an extended article (in Italian) describing and comparing different aspects of the two iterations is in press [Lodi et al., 2023].

A website[4] has been set up with all the material in English and Italian [Lodi et al., 2021a]. All material is available under an open license to promote its dissemination and reuse.

The project has been presented several times to hundreds of teachers and several educational researchers. For example,

- at the "Bologna Linux Day 2021"[5] (Bologna, 23rd October 2021);
- at a meeting of the Cryptography and Coding Theory subgroup of the Italian Mathematical Union working on cryptography teaching and outreach (Online, 23rd June 2021);
- at a meeting between informatics and mathematics education researchers (University of Milan, 1st February 2022)

The course will be repeated a fourth time as part of the "Science Degree Project" (PLS), an Italian project to attract high school students to enroll in science degrees.

[2] https://sites.google.com/view/crittografiaecodici/convegno-annuale
[3] http://www.seminariomatematico.polito.it/rendiconti/
[4] https://bigideascryptok12.bitbucket.io/
[5] https://docs.google.com/document/d/1kpOo3tAvhhI6dCzrZVkYBWgg8j7lQkF_zICf4rSW35k/

Chapter 14

Cryptography in Grade 10: Big Ideas with Snap! and Unplugged

We describe an introductory course on the "big ideas" of cryptography, designed for the second year of the *Liceo Matematico*, an Italian experimental strand of the scientific lyceum (high school).

The key feature of our course is the approach "by discovery" (see 5.1.1). It involves a succession of cryptosystems (from classical to modern ones); of each one, students can experience the characteristics, possible attacks, and limitations to feel the *necessity* to discover the following. After experimenting with each system, students are involved in a Socratic discussion on how to overcome the discovered limitations. Precisely the necessity of overcoming such limitations motivates introducing the following system in the learning path. We used *Snap!*[1] – a block-based graphical programming language for learning – to build *playgrounds* (i.e., *task-specific* programming languages (see) with a minimal selection of targeted instructions) where students can experiment with various cryptosystems and schemes. We also used Snap! to conduct an *unplugged* activity on the Diffie-Hellman protocol.

Our first goal is to describe how we taught some big ideas of cryptography by making students encounter a progression of representative cryptosystems and discover their characteristics and limitations. Our second goal is to evaluate the students' perceptions and learning of cryptography core ideas. They appreciated the course and felt that, despite being remote, it was fun and engaging. According to the students, the course helped them understand the role of cryptography, mathematics, and informatics in society and sparked their interest, particularly in cryptography and informatics. The final assessment showed that the students well understood the cryptography ideas covered in the course. Our third goal is to discuss what worked and areas of improvement. The "remote-unplugged" Diffie-Hellman, where the meeting chat was a metaphor for the public channel, engaged the students in understanding this groundbreaking protocol. Overall, they praised the activities as engaging, even when challenging. However, a strong "instructor blindness" induced by remote teaching often prevented us from giving the students the right amount of guidance during the exploration

[1] https://snap.berkeley.edu/about

activities.

In the following, we present in detail the course, activities, and materials, as well as a preliminary evaluation of the teaching intervention performed after two classroom implementations (the first online and the second in person).

14.1 Introduction

Section 6.1 illustrates how cryptography is an essential ingredient of many of the activities and tools of our contemporary digital and connected society. In addition, section 6.2.1 shows how European frameworks and various international curricula include skills related to cryptography as a pillar of informatics security. Pre-college education does not aim to train professionals but to help students understand the world we live in so that they can take an active part in it, pursuing their own goals. Therefore, we believe it is important for students to know and understand the principles of cryptography and their relevance to the activities and tools of the digital society.

With these premises in mind, we realized a short course on cryptography. It is a course without specific prerequisites, built, on the one hand, around the fundamental concepts of cryptography (its *big ideas*; see 4), and, on the other hand, on various kinds of hands-on activities that allow for a direct and active experience of those concepts. This approach has its roots in educational research, which has extensively documented the effectiveness of active and cooperative methodologies in fostering learning [Loui and Borrego, 2019, p. 304]. About active learning for informatics education, see also.

Therefore, we designed and developed innovative, hands-on, interactive activities that were as immediately meaningful as possible for young, novice students. In particular, we built cryptography *playgrounds*, that is, digital environments for experimentation in which students, through visual block programming, could use, understand, and attack some of the most significant classical ciphers (e.g., Caesar and one-time pad ciphers). We also developed an *unplugged*[2] yet remote activity, as the course's first iteration, in February 2021, was held remotely due to the COVID-19 pandemic. Through such "remote-unplugged" activity, students could run the Diffie-Hellman protocol in pairs to generate (securely, despite communicating only over an insecure channel) a secret communication key. We implemented both kinds of activities using Snap!, a block-based graphical programming language designed to be easily used even by those completely new to programming.

The course addresses some of the most significant cryptography schemes and systems (classical and modern), selected so that they are emblematic representatives of some key ideas of cryptography. In order to realize a motivating progression, the introduction of a new scheme or system is always triggered by the *necessity* to overcome the limitations of the previous system(s). Indeed, the progression is designed to stimulate in students the *necessity mechanism*, a general learning mechanism discussed in 9.2.1 (which is the core of our learning design for introductory programming described in chapter 9 and the journal article by Sbaraglia [2021]).

[2]See for a general review of the unplugged approach in informatics education and 6.2.5 for its use to teach cryptography.

As mentioned, the first iteration of the course (February 2021 for the school year 2020/21) was held exclusively remotely due to the pandemic, while the second one was held in person (November-December 2021 for the school year 2021/22).

In the following 14.2, we present the learning path we developed to introduce the fundamental ideas and principles of cryptography. We discuss in detail the concepts and schemes that drive the learning progression (14.2.3) and describe the development and testing of the activities implemented with Snap!, both the cryptography playgrounds and the Diffie-Hellman *unplugged* activity (14.2.6.2). We present the results of data collection on two iterations of our course (14.3 and 14.4) both from the perspective of comprehension of the cryptography concepts addressed (14.3.1) and in terms of the students' own perceived satisfaction and usefulness of the course at the end of each iteration (14.3.2). Finally, we discuss what worked and what can be improved (14.4.1) and how to help teachers of informatics (but also of mathematics and STEM disciplines in general) to adopt (and adapt) our learning progression and its hands-on activities in different contexts (14.4.3).

This report is an extended version – expanding on the course design and, in addition, analyzing the second iteration – of the experience report we published in ITiCSE '22 titled "*Cryptography in Grade 10: Core Ideas with Snap! and Unplugged*" [Lodi et al., 2022b], which was shortlisted for best paper.

14.2 The course

14.2.1 Context: Mathematical Lyceum

In Italy, upper secondary school lasts five years (usually, students start when they are 14 years old and finish when they are 18 years old), but only the first two are compulsory. The upper secondary level has three main strands, lyceum, technical school, and professional school, each with various tracks. Our intervention took place in a lyceum[3] that gives a theoretical basis in classical, scientific, or artistic areas and naturally leads to university studies. When choosing a lyceum, students select among specific tracks (e.g., "classical", "scientific", "linguistic") established at a national level. In Italy, lyceums "*provide students with the cultural and methodological tools for an in-depth understanding of reality so that they can place themselves, with a rational, creative, planning, and critical attitude, before situations, phenomena, and problems and acquire knowledge, skills, and competencies coherent with personal abilities and choices and appropriate for pursuing higher-order studies, entering social life and the world of labor.*" [MIUR, 2010, p. 4, our translation]. Our intervention took place in the context of a national experimental project called *Mathematical Lyceum* ("Liceo Matematico"), which involves more than 140 schools. It consists of extracurricular activities in which students from all lyceum tracks can voluntarily participate. Students are involved in laboratory activities so that they can have concrete interdisciplinary experiences involving mathematics and other disciplines. The purpose of Mathematical Lyceum is not to impart new notions to students but to engage them in interdisciplinary thinking on the fundamentals of knowledge so that they can expand their tools for interpreting reality and ultimately

[3] See Bellettini et al. [2014] for a summary of the Italian secondary school system.

broaden their cultural horizons [Liceo Matematico, [n.d.]]. We delivered a cryptography course as one of these extracurricular activities in a local scientific lyceum. Our short course on cryptography was conceived and developed in the context of Mathematical Lyceum and, in line with the general objectives of this national project, does not aim to provide new (technical and professionalizing) notions but to help students understand cryptography importance today by letting them experience its fundamental principles and transformative ideas concretely through an interdisciplinary teaching approach.

14.2.2 Two iterations: online and in person

We conducted two iterations of the course with students who chose the extracurricular path of *Mathematics Lyceum* in a scientific lyceum in Casalecchio di Reno (Bologna). The first iteration was held in the school year 2020/21 with students in their second year of lyceum (15-16 years old), and the second iteration in 2021/22 also with students in their second year.

The course was taught by two of the authors (myself included) of the ITiCSE paper [Lodi et al., 2022b], who are informatics education researchers and have extensive experience teaching informatics in secondary school.

First iteration (online) The first iteration took place in February 2021. We held four lessons (of 2 hours each) on Tuesday afternoons. Due to the COVID-19 pandemic, the lessons were held online through the Google Meet platform adopted by the school.

Fifteen students (5 girls and 10 boys) participated. None of them had previous programming experience.

Second iteration (in person) The second iteration occurred between November and December 2021, in 5 lessons (2 hours each) on Tuesday afternoons. The classes were held, all in person, in one of the school's informatics laboratories, where a computer was available for each student.

Thirteen students (5 girls and 8 boys) participated. Three students had already done a few simple visual block programming activities in primary or secondary school. All others had no previous programming experience.

14.2.3 A progression driven by the limitations of the previous cryptosystems

The course aims to teach fundamental cryptography ideas (crypto *big ideas*) by having students experience firsthand significant cryptographic schemes and protocols, from classical to more modern ones. Students, using block programming, experiment with encryption, decryption, and attack mechanisms of some cryptosystems and, with an interactive simulation in pairs, generate a shared secret key. These hands-on activities are intended to help students understand the cryptographic principles of the schemes and protocols they encounter. Understanding is often stimulated by students' direct experience of the limitations of such schemes and protocols. In particular, students encounter schemes and protocols in a progression designed so that the introduction of a new system is always motivated by the *necessity* to overcome the limitations of the previous one(s), always advancing toward

more secure systems. The course also addresses some mathematical concepts fundamental to cryptography (e.g., modular arithmetic), seeking to show, with an interdisciplinary approach, their role in the conception and implementation of the systems encountered.

14.2.4 Cryptography principles and ideas through meaningful cryptosystems

As mentioned, the choice of cryptosystems was motivated by the cryptography ideas and principles that can be taught through them.

Caesar cipher, proposed in the version where the key is a positive integer indicating the shift in the alphabet, was chosen because it has a simple function. Therefore, it is convenient to introduce fundamental concepts such as key, plaintext message, ciphertext message, encryption algorithm, and decryption in a concrete context. Moreover, it is easy to perform brute-force and frequency-based attacks and to reflect on the reasons for its vulnerability to these two kinds of attacks (respectively: few keys and the fact that the same plaintext character is always mapped into the same cipher character).

In contrast to traditional pathways, which consider many historical ciphers, we decided to mention only the Vigenère cipher and Enigma briefly. In order to promote a more straightforward understanding, polyalphabetic ciphers are presented as ciphers in which the key is composed of a finite sequence (of fixed length) of "Caesar's keys". The first letter of the message will be ciphered with Caesar using the first number of the key sequence, the second still with Caesar but with the second number of the key sequence, and so on. Even following this simple approach, it is still possible to easily show the vulnerabilities of polyalphabetic ciphers. For example, common words that repeat at distances multiple of the key length allow discovering this distance (i.e., the length of the key) and thus attacking with frequencies.

It is precisely this vulnerability that suggests moving to a system in which the key is a sequence of random numbers that is as long as the message. We continue to informally call the numbers of the key "Caesar's keys" again to support students' understanding. In this new system, each letter in the plaintext message is encrypted with a different key number, the one whose position corresponds to the letter's position in the plaintext. This way, the one-time pad cipher is introduced. Through the related hands-on activity (implemented through one of the Snap! playgroungs), students soon realize that, for long messages, each letter has the same probability of being encrypted with all possible "Caesar's keys". This phenomenon corresponds to a "flattening" of the frequencies of each character in the encrypted message, thus making frequency-based attacks ineffective. Brute-force attacking one-time pad cipher is even more surprising. Since the keys are all possible sequences of numbers (from 1 to the length of the alphabet), the encrypted message has the same probability of originating from every plaintext message of the same length. A brute-force attack generates all possible message-long permutations (with repetitions) of the alphabet letters (i.e., all possible sequences of alphabet characters of a given length), thus without revealing any information about which of those might be the original plaintext message. Then the idea that one-time pad reveals no information about either the original message or the key (as long as it is changed each time and is truly random) is made explicit to students, thus realizing *perfect security*. One-time pad security is perfect but not sustainable in practice because of the difficulty of generating,

using, and exchanging truly random keys, as long as the plaintext message, and different each time.

Then, some mention is made of transposition ciphers: the transposition mechanism and the substitution mechanism (extensively tested with Caesar and one-time pad ciphers) are fundamental elements of modern ciphers.

Modern block ciphers are then introduced. This is an opportune time to show how computers (without which modern cryptography could not concretely come to fruition) represent information. In particular, it is shown how all characters are represented by 0 and 1 through specific and conventional encoding schemes. It is then made explicit to students that computers represent, through some form of encoding, any kind of information with sequences of only 0 and 1, *binary digits* called *bits*. Simple operations (which computers perform very quickly) on bits are also shown. In particular, we illustrate XOR, the exclusive disjunction bit operator, which results in 1 only if exactly one of its two operands is 1. XOR is much used in cryptography because it hides all information about the value of the operands: it only reveals whether they are the same or different from each other without giving any other information. Because of this property, bit-to-bit XOR between a plaintext and a key (whose bit sequence is as long as the plaintext's) reveals nothing about the plaintext and key other than whether their corresponding bits are equal.

The transition to using bits is necessary to show some fundamental properties of block ciphers. For example, the fact that appropriate and repeated substitutions and permutations of bits generate the so-called "avalanche effect": a small change in the plaintext and/or key causes a large change in the ciphertext. The avalanche effect ensures the properties of confusion and diffusion. This topic provides an opportunity to discuss the security model. Security is no longer perfect, like that of one-time pad, but computational. *Computational security* is based on the fact that such systems are not attackable with the computing power of today (and the immediate years to come) in a "reasonable time"; performing a successful attack would require a time frame much longer than the human lifespan.

It is also made clear that modern block ciphers still base their security on the existence of a secret key shared between the parties; in other words, they do not solve the problem of key distribution.

Ceasar cipher

- *Representative for:* monoalphabetic substitution ciphers
- *Motivations:* basic example of symmetric-key cryptography; easy to show the typical cryptosystem elements (plaintext, ciphertext, encrypt and decrypt functions, key) and simple attacks; easy to understand and experiment with it

↓ *Problems to overcome:* attackable with both brute-force and frequency analysis

One-time pad chiper

- *Representative for:* polyalphabetic ciphers (taken to the extreme); perfect secrecy; resistant to both brute-force and frequency attacks (no clues about the key or the plaintext from the ciphertext)
- *Motivations:* easy to explain as "a different Caesar for each letter"

↓ *Problems to overcome:* key-distribution problem; feasibility issues (e.g., one-time and random keys, key as long as the message)

Simple substitution–permutation network

- *Representative for:* modern symmetric block cryptosystems (e.g., AES); confusion and diffusion (avalanche effect); efficient hardware implementation; "only" computationally secure
- *Motivations:* introducing operations on bits; grasping how modern cryptosystems are implemented (efficiently) with computers

↓ *Problems to overcome:* key-distribution problem

Diffie-Hellman key-agreement protocol

- *Representative for:* shared-key generation protocols; groundbreaking solution to the key-distribution problem
- *Motivations:* understanding how the discrete logarithm (easy to calculate, hard to invert) allows generating a shared secret over a public (insecure) channel

↓ *Problems to overcome:* person-in-the-middle attack

Idea of public-key secrecy and authentication

- *Representative for:* asymmetric cryptosystems
- *Motivations:* grasping that the properties of certain math functions (e.g., prime factorization) can be used to achieve both secrecy and authentication

↓ *Problems to overcome:* computationally expensive

Idea of hybrid cryptosystems

- *Representative for:* today's complex cryptosystems
- *Motivations:* understanding how the best of symmetric and asymmetric cryptosystems combine in today's practice; grasping how relevant modern services (e.g., e2e instant messaging) work

○ *Problems to overcome: not raised in the course*

14.2.5 Contents

We implemented the learning path in 4 lessons in the first iteration and 5 in the second. For both iterations, we schematically report the contents below.

First iteration *Lesson 1*

- The social debate about encryption in digital communication
- Caesar cipher: encryption, decryption, brute-force attack
- *Homework: transposition vs. substitution, Kerckhoffs's principle*

Lesson 2

- Caesar cipher: frequency-based attack
- One-time pad cipher: encryption, decryption, frequency-based attack
- *Homework: encoding characters as bits, example of toy cryptosystem using XOR and bit permutation, hints at modern block ciphers (DES, AES)*

Lesson 3

- One-time pad cipher: brute-force attack, perfect secrecy, limitations, key-distribution problem
- Diffie-Hellman protocol: simulation with colors
- *Homework: quiz on one-time pad cipher and Diffie-Hellman protocol*

Lesson 4

- Math of Diffie-Hellman protocol: modular arithmetic, exponential and its inverse (discrete logarithm), primes and coprimes (and hints at generators)
- Diffie-Hellman protocol: an example with small numbers, computational security, person-in-the-middle attack
- Asymmetric-key cryptography: terminology, properties of public/private key pairs, non-technical, high-level schemes for authentication and secrecy, intuitive idea of *one-way function* (prime multiplication vs. factorization)
- Putting all together: how *intuitively* combine asymmetric and symmetric schemes for authentication and secrecy
- *Homework: satisfaction survey, fill-in-the-blanks assessment (and consolidation)*

Second iteration In the second iteration, we decided to add an introductory lesson (for this reason, called "Lesson 0") to

- foster the creation of a class group (students are from different classes and often do not know each other) by having them discuss in pairs and bigger groups;
- stimulate students better about the importance of cryptography by bringing out their ideas and prior knowledge through an initial open discussion;
- introduce students to Snap! simply and playfully (by showing the creation of a toy video game) so that they become familiar with the environment without having to pay attention to cryptography concepts (thus reducing cognitive load).

The structure of the lessons is presented below. Differences from the first iteration are highlighted in bold.
Lesson 0

- **Peer discussion: where and for what purposes cryptography is used in our lives; why it is important in society**
- The social debate about encryption in digital communication
- **Introduction to Snap!: creating a simple video game**
- Caesar cipher **(on the blackboard)**: encryption, decryption, **today's uses: ROT13**
- **Homework: invent a safe way to communicate**

Lesson 1

- Caesar cipher: encryption, decryption, brute-force attack
- *Homework: transposition vs. substitution, Kerckhoffs's principle*
- Caesar cipher: frequency-based attack
- **Homework: invent a cryptosystem that resists both brute-force and frequency-based attacks**

Lesson 2

- **Hints at polyalphabetic ciphers and Enigma**
- One-time pad cipher: encryption, decryption, frequency-based attack, **brute-force attack**
- One-time pad cipher: **reasons for its perfect security**, limitations, key-distribution problem
- *Homework: quiz on one-time pad cipher,* **reflection on the key-distribution problem**

Lesson 3

- **Example of a transposition cipher**
- **Encoding characters as bits, XOR operator, UNICODE**
- **Interactive educational cipher: simple substitution and permutation network; reflection on confusion and diffusion properties**
- **AES: hints on key length and computational security**
- **Homework: further reflection on how to realize a secure exchange of a key via an insecure channel**

Lesson 4

- Diffie-Hellman protocol: simulation with colors
- Math of Diffie-Hellman protocol: modular arithmetic, exponential and its inverse (discrete logarithm), primes and coprimes (and hints at generators)
- Diffie-Hellman protocol: an example with small numbers, computational security, person-in-the-middle attack
- Asymmetric-key cryptography: terminology, properties of public/private key pairs, non-technical, high-level schemes for authentication and secrecy, intuitive idea of *one-way function* (prime multiplication vs. factorization)
- Putting all together: how *intuitively* combine asymmetric and symmetric schemes for authentication and secrecy
- *Homework: satisfaction survey, fill-in-the-blanks assessment (and consolidation)*

14.2.6 Tools, activities and methodologies

14.2.6.1 Tools used

The host school uses Google Workspace for Education, a suite of software tools to support education. Therefore, in the first iteration, we used Google Meet for the video conferencing lessons.

In both iterations, we used Google Classroom to share links to the activities, instructions and announcements, materials and homework assignments. In addition, we used Google Slides to present the content and animations of the cryptographic schemes and as ongoing support for the various moments of the lessons. We used Google Docs to share longer texts and collect student reflections and works. Furthermore, we used Google Forms as a working tool to structure hands-on activities with guiding questions and homework assignments. At the end of the course, we relied on Google Forms also to collect students' perceptions and opinions about the course and students' answers to the final consolidation and evaluation activity.

Snap! is a block-based graphical programming language for learning. It is a re-implementation of Scratch, a well-known block language for children ages 8 and up. Compared to Scratch, Snap! has many additional features that allow it to introduce programming in a simple and playful yet rigorous way to novice students in primary and lower secondary school and even upper secondary. In our specific case, we chose Snap! because it allows new customized blocks to be created (in addition to the standard blocks provided). Unlike in Scratch, such blocks can return values, thus realizing functions (in the informatics sense of the term), and their implementation (the function's body) can be kept hidden from students.

14.2.6.2 Tools created

Using Snap!, we created a progression of crypto playground, i.e., digital environments in which students can experiment with some relevant cryptosystems using programming. However, the programming is limited to simple combinations of a few available blocks. Such environments allow students to use and try to attack some significant cryptosystems (e.g., Caesar and one-time pad ciphers) and also to gain concrete experience of their limitations (for example, the ease or difficulty of attacking them or the processing time required to carry the attacks). Specifically, our playgrounds are Snap! projects in which the visible and available instruction set (the blocks) has been limited.

We take advantage of the possibility given by Snap! to hide some predefined blocks of the language and especially to create new custom blocks. For each cryptosystem, we provided students with only the blocks they needed to encrypt and decrypt messages and possibly, depending on the activity, to run some of the potential attacks on the system. We identified specific cryptographic mechanisms we wanted to draw students' attention to and implemented them via Snap! as *ad-hoc* blocks of the language. We tried to maximize the expressiveness of these new blocks for the cryptographic context while minimizing the technical aspects related to programming to facilitate their understanding and use. Under these choices, our playgrounds can be considered as *teaspoon languages* [Yadav and Berthelsen, 2021], i.e., "*task-specific languages [... that]: support learning tasks that teachers (typically non-CS teachers) want students to achieve; are programming languages, in that they specify computational processes for a computational agent to execute; and are learnable in less than 10 minutes, so that they can be learned and used in a one hour lesson*" [Guzdial, 2021] (about task-specific languages, see 5.1.3).

Since our high-school students had no prior programming experience, we did not consider it feasible – or even beneficial, given the mathematics context and the high-level goals of the course – for them to program the (algorithms of) cryptosystems at a lower level of abstraction than the combination of cryptographic blocks. In such a context, an excessive focus on programming might pull students away from the cryptographic scenario and prevent them from focusing on its fundamental ideas. However, we still wanted to allow students to build their knowledge through concrete manipulation of computational objects [Papert, 1980] related to cryptography. Specifically, we avoided putting students into micro-worlds *à la Papert*, which usually rely on general-purpose programming languages such as LOGO – which, even when designed for educational purposes, require significant learning time [Guzdial and Naimipour, 2019]. Instead, we have developed and implemented much simpler (and

thus easier to use) mini-languages, circumscribed (thus not general-purpose) to our specific learning goals, according to Guzdial [2021] and also our vision on computational thinking (see 3.1)[4]. For example, in the playground where students can try to attack Caesar cipher using frequencies, the blocks available are: calculating the letters' frequencies in a text, sorting a table (i.e., a matrix of letters and their frequencies) by frequency, representing table data with a histogram, and the table of letters' average frequencies in Latin (fig. 14.1).

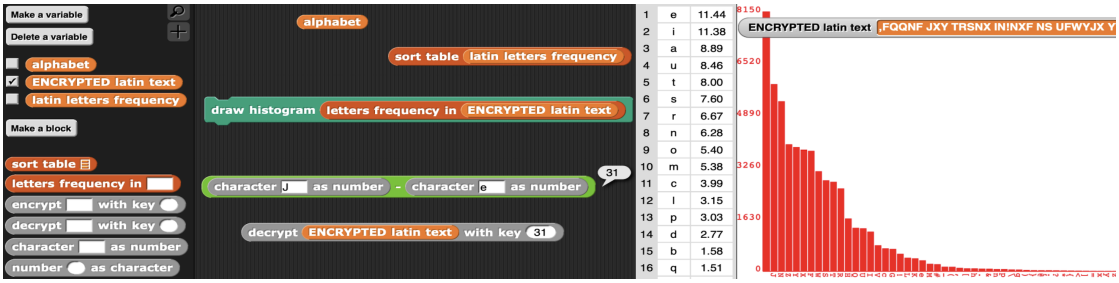


Figure 14.1: Snap! playground for frequency-based attacks on Caesar cipher

Our playgrounds are available for exploration and use [Lodi et al., 2021a]. Following the instructions we shared with them each time, the students engaged in small challenges in which they had to encrypt, decrypt or attack a particular system by combining some specific Snap! blocks provided in the playground. These challenges' broader goal was for students to experience cryptosystems so as to foster an understanding rooted in the experience of how they work and their limitations. In addition, we aimed to contextualize these hands-on activities in scenarios meaningful for students. For example, the activities on Caesar cipher involved deciphering and then discovering a Latin text (the same text they were to translate in the upcoming test) that their teacher, quarantined for COVID, had to communicate to the substitute teacher without students being able to intercept it[5].

Since the students in the two courses were entirely new to programming, we felt that it was too challenging for them (and not appropriate in a short course with other objectives) to program a cryptographic protocol such as the Diffie-Hellman key agreement. As a result, we decided to adopt an *unplugged* approach so as to develop an activity that was concrete – leading to the actual generation of a shared secret key – while maintaining a solid connection to the formal aspects of this revolutionary protocol. Indeed, the unplugged approach can help understand at a high level how meaningful informatics algorithms work by having students perform them "firsthand" through kinaesthetic and fun activities [Bell et al., 2009].

In the literature and instructional materials analyzed, we found at least two ways traditionally used to explain the workings of the Diffie-Hellman protocol for generating a shared secret key (e.g., Wikipedia contributors [2022]). Color mixing, which is more evocative but

[4]We believe that leveraging this kind of hands-on activity (e.g., crypto playgrounds in which students use programming as a "task-specific" tool) in a learning path aimed at big ideas embraces both the constructivist soul of computational thinking [Papert, 1980; Papert and Harel, 1991] and its broader, soft-skills-oriented perspective [Wing, 2006; 2008].

[5]All students in our courses had Latin among their curricular subjects.

simplified, and the step-by-step execution of the algorithm with small numbers, which is more precise and faithful but also more complex to understand at first. We tried to combine these two by developing a series of learning steps to gradually move from the color metaphor to the algorithm's mathematical operation.

An unplugged activity in pairs is the core of our strategy to get students to understand the high-level operation of the Diffie-Hellman protocol. The activity is based on an executable-only Snap! project (the code cannot be modified or explored) which functions as an interactive app. This app guides students in the step-by-step execution of the Diffie-Hellman protocol through color mixing [Lodi et al., 2021a]. The color mixing is not done with classical additive or subtractive algorithms; instead, it is based on the actual protocol calculations (albeit initially hidden from students) on small numbers, from 0 to 99, representing the colors available in Snap!. Using the Snap! project as an interactive app, the student pairs were able to generate a shared secret color. Communications within each pair – necessary to agree on an initial color and then to exchange their respective calculated intermediate colors – took place in the meeting public chat of the videoconferencing service (i.e., Google Meet) by which the lessons were broadcast. The public chat of the meeting was used to represent, and did so effectively, an insecure channel: all participants in the lesson could "listen" on that channel (i.e., read the chat) without any restriction (see fig. 14.2).

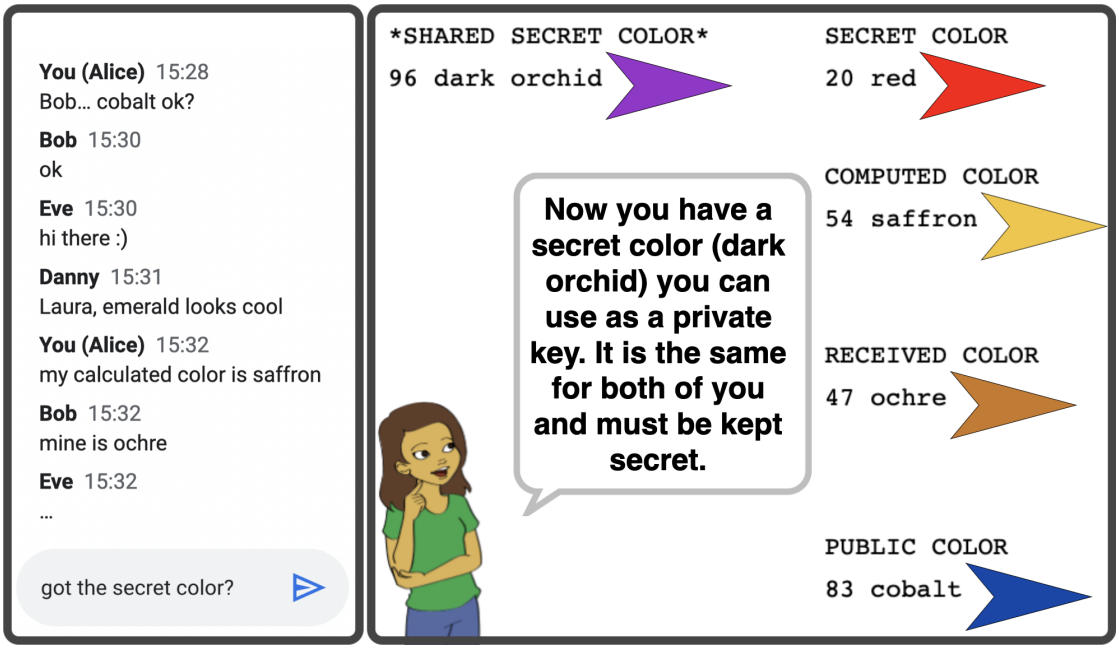


Figure 14.2: Meeting chat and support app for the Diffie-Hellman activity

After students experienced firsthand the high-level functioning of the Diffie-Hellman protocol, we illustrated the essential mathematical tools for its operation (see 14.2.5). Then, we showed the step-by-step execution of the protocol between two communicating parties

using an animation [Lodi et al., 2021a]. This animation shows the correspondence between colors and the small numbers (from 0 to 99) that represent such colors in Snap!, revealing the actual calculations behind the color mixing experienced in the unplugged activity.

Because of the short duration of the course, we decided to use a still interactive and dialogic but more frontal method to teach more advanced public key cryptography schemes. We think that hands-on activities – effective in supporting learning but necessarily requiring a longer time – are strategic in the first part of the course to facilitate everyone's understanding of the basic concepts and mechanisms. Hands-on activities on easier cryptosystems and schemes are also instrumental in sharing specific terminology, language, and reasoning in context from the beginning. Based on the solid foundation built through the initial (meaningful yet easy-to-understand) hands-on activities, we just illustrated the high-level mechanisms of asymmetric cryptography and their usage in concrete scenarios and discussed their importance today. On the other hand, given the overall objectives of the course (i.e., a general understanding of the cryptography core ideas and the awareness of how the interplay of the various cryptographic schemes enables today's safe, fast, affordable security and authentication), we decided to increase the pace and abstraction level to complete the overview of cryptography. Indeed, we believe that a comprehensive and "operational" overview, even if simplified and abstract, is necessary to engage students on a real ground that connects to their everyday experiences with cryptography (e.g., confidential communications with friends, participation in social media, paid digital services).

We illustrated to the students that it is possible to create two keys, one public and one private, linked by the property that what has been locked (encrypted) with one key can only be opened (decrypted) with the other. We merely suggested the mathematical mechanism on which this asymmetric encryption works. In a key pair, the public key is related to the multiplication of two very large prime numbers (easy), while a private key is related to the factorization of their very large product (difficult). We asked the students to imagine how to use the key pair to realize a secret communication. Then, we guided them to the concept of authentication – which emerges here for the first time in the course – and illustrated its implementation with the asymmetric scheme. Following, we discussed at a high level the specular use of the two keys to realize secrecy and authentication and how to combine their use to achieve both properties. In order to support the understanding of the schemes of asymmetric cryptography, we developed animations [Lodi et al., 2021a] of simple communication scenarios. In these animations, we used two characters from the well-known TV series *Power Rangers*[6] so that the actions and messages of the different parties were evident through the characters' colors. The various scenes in these animations paced and supported the introduction of cryptographic concepts, encouraging reflections through their concrete visual elements. After finishing the step-by-step (yet high-level) analysis of the two proposed communication schemes, specific closing animations allowed students to visually summarize the overall functioning of the asymmetric authentication and secrecy schemes.

Modern block ciphers using symmetric keys (such as AES) had been covered in the first iteration only in the homework, through a reading on the origin of DES and with a simple exercise of bit manipulation. Since we found this approach to be unfriendly and not

[6] https://en.wikipedia.org/wiki/Power_Rangers

particularly engaging, we decided to use some of the extra time to give students a better understanding of the mechanisms underlying modern substitution and permutation networks (SP-networks). Therefore we realized (using a Google Sheet spreadsheet) a simplified version of an SP-network, working on four blocks of four bits each, with three substitution rounds and two permutation rounds [Lodi et al., 2021a]. The key and message can be changed in the spreadsheet to observe live changes on the bits across the network. Despite the small size of the network, the spreadsheet allows students to easily and interactively observe the avalanche effect on the encrypted message with minimal changes in the bits of the key or plaintext message.

The other relevant change from the first iteration concerns the other homework more generally. The first three (out of four) homework assignments in the second iteration encouraged students to invent their own cryptosystems. The first was completely free, and the second asked for resistance to frequency-based and brute-force attacks. Then, for the third homework, the students were asked for a way to exchange a secret key securely between two parties. These tasks always preceded the explanation of possible solutions found by cryptography throughout history. In this way, students experience even more the limitations and crucial difficulties of cryptosystem design and can thus feel the *necessity* of new cryptographic schemes that overcome them. The students in the second iteration often came up with creative solutions, although – as was to be expected – rarely as secure as required. The lessons that followed a homework assignment always began with highly participatory discussions on students' solutions. These discussions were a perfect springboard for introducing an emblematic cryptosystem or scheme (e.g., *One-time pad*, Diffie-Hellman) that solved the problems of students' proposals that we teachers helped bring out during the discussion.

14.3 Data collection and analysis

At the end of each course iteration, we asked the students to fill out two Google Forms to collect their feedback and evaluate their learning. No grades were provided for these conclusive activities.

In the first iteration, the two questionnaires were filled out by the same 14 students (out of 15). In the second iteration, the two questionnaires were filled out by the same 11 students (out of 13).

14.3.1 Learning assessment

We prepared a summary of about 2000 words [Lodi et al., 2021a] of the important ideas and concepts covered in the course, identifying several key passages in the text. Students had to choose between the right fill and the wrong alternative for each one. We also wanted the activity to be a review opportunity for students, so we structured it as a "story" summarizing the most important content of the course to consolidate their learning.

The original text (used at the end of the first iteration) includes 43 key passages in which to choose between two options. In the second iteration, the text remained largely unchanged,

except for minor stylistic changes and the addition of a few sentences related to additional content. Thus, in the second version (the one available in the course materials [Lodi et al., 2021a]), the key passages became 46.

First iteration From a learning perspective, the results were positive. Out of 43 choices to be made, the mean of correct answers was 32.5, and the median was 34, with a range of correct answers between 17 and 41.

Second iteration The results of the second iteration were even more positive. Out of 46 choices to be made, the mean of correct answers was 40.7, and the median was 40, with a significantly smaller range of correct answers between 35 and 45.

14.3.2 Student satisfaction and perceptions

In both iterations, the students' participation was high.

First iteration Out of 14 students who responded to the questionnaire, 13 attended Lesson 1, 12 attended Lesson 2, 11 attended Lesson 3, and 12 attended Lesson 4.

Second iteration Out of 11 students who responded to the questionnaire, all attended Lesson 0, 10 attended Lesson 1, 7 attended Lesson 2, 10 attended Lesson 3, and 9 attended Lesson 4.

14.3.2.1 Feedback on the experience

In the satisfaction questionnaire, we asked students for feedback on their learning experience and the course's impact on their perception of informatics, mathematics, and cryptography. All questions were mandatory.

The course duration was adequate for the vast majority of students in both iterations (see fig. 14.3).

As for the activities, most students found them easy, engaging, and useful for their personal development and understanding of the world. Some students reported that their prior school background was not fully adequate for the course activities (see fig. 14.4). In the second iteration, the students were asked about the course's usefulness for their own school career, the only dimension most students were not positive about.

Concerning the course difficulties, in both iterations, only one and two students, respectively, reported that they 'often' experienced difficulties during the course, while almost all of them only 'sometimes' (see fig. 14.5).

In both iterations, most of the difficulties reported were related to the complexity of classroom activities. On the other hand, regarding timing and organization, the second iteration showed a significant improvement, probably due to the additional lesson and the in-person context (see fig. 14.6).

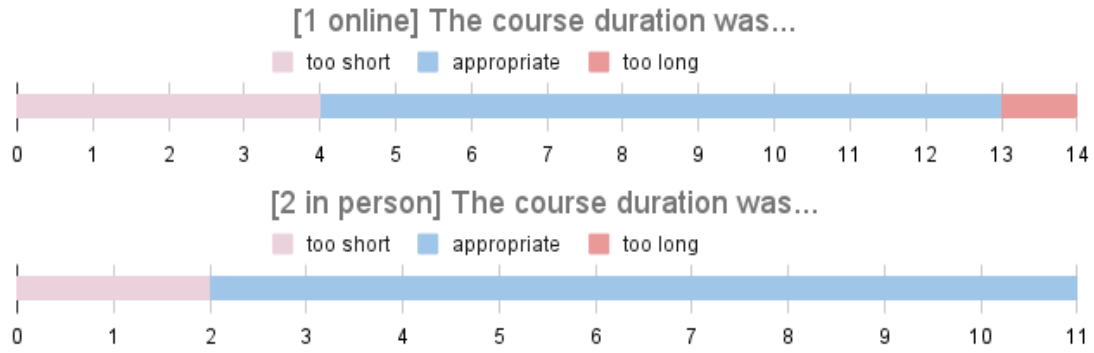


Figure 14.3: Students' opinion on the course duration

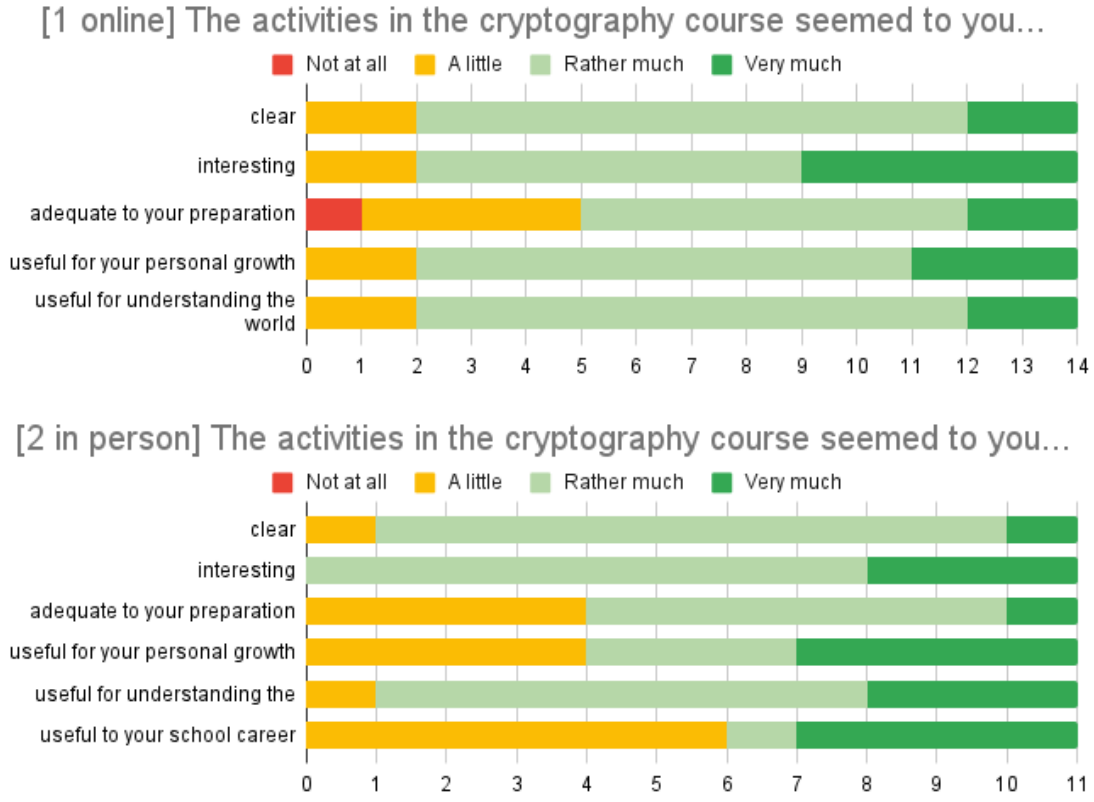


Figure 14.4: Students' overall evaluation of the activities

The students were extremely satisfied with their interaction with course teachers in both iterations (see fig. 14.7).

Overall satisfaction with the course was also very high in both iterations; only one student

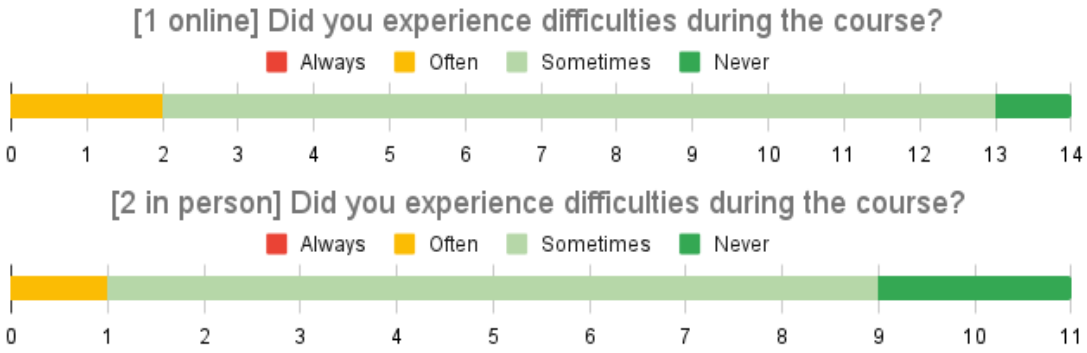


Figure 14.5: Students' self-assessment of difficulties experienced

in the first iteration said he was 'A little' satisfied (see fig. 14.8). As an indication of high satisfaction, 13 out of 14 students in the first iteration, and all 11 in the second, would recommend the course to a friend.

Getting more specific, we asked students for their opinions on the specific tools and methodologies to better evaluate their perceived effectiveness and satisfaction. We refer to the figures (from 14.9 to 14.12) for a detailed overview; however, we report here the emerging insights that are most relevant.

The students reported that Cryptography block-programming activities with Snap! playgrounds were 'Rather much' or 'Very much' useful and engaging; less than half of the students did not find such activities easy (see fig. 14.9).

In the second iteration, being in person allowed us to have very participatory discussions that we felt were stimulating, an impression confirmed by almost all of the students who found them useful and engaging (see fig. 14.10).

Since students in the first iteration considered the homework not engaging, the homework in the second iteration was reconsidered entirely and redesigned. The students' opinions show a clear shift in perception in favor of the new proposals (see fig. 14.11).

In the first iteration, the students appreciated the unplugged activity on the Diffie-Hellman protocol 'Very much' (see fig. 14.12). A school network malfunction prevented us from replicating it in the second iteration, so we were forced to show it in action by involving only two students. Consequently, the related question was not included in the questionnaire of the second iteration.

14.3.2.2 Perceived usefulness

Regarding the perceived usefulness of the course, most students in both iterations found it useful in better understanding cryptography, what it is about, and its role in society. There was also an improved understanding of the role of informatics and mathematics in society. These aspects were particularly positive in the second iteration. More generally, perceived usefulness was higher in the second iteration.

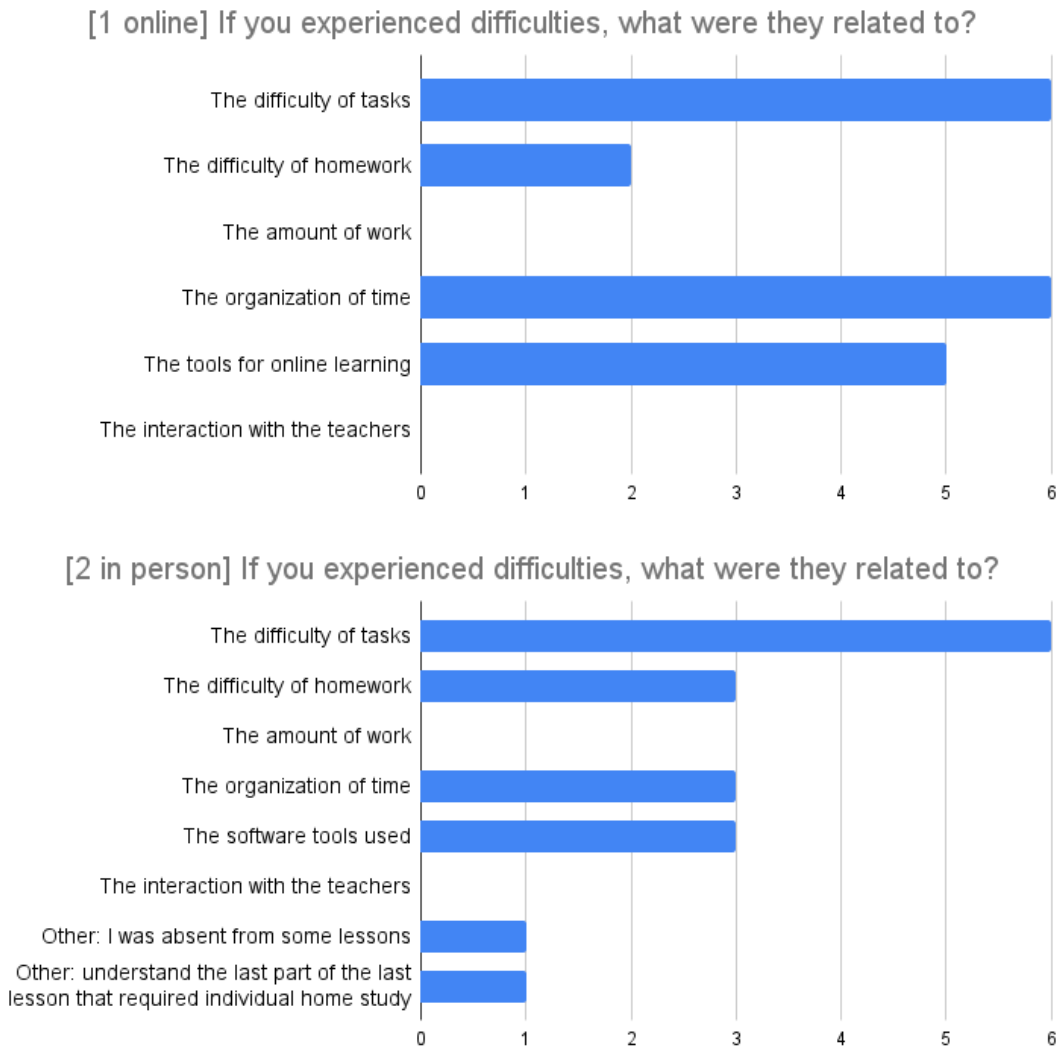


Figure 14.6: Students' self-assessment of the kind(s) of the difficulties experienced

Finally, in both iterations, while the course prompted interest in informatics and cryptography in about $2/3$ of the students, only less than half perceived an increased interest in mathematics (see 14.13).

14.3.2.3 Students' free comments

At the end of the satisfaction questionnaire, we gave the students a chance to freely express comments, remarks, and suggestions (i.e., "A free space where you can write down what you feel like").

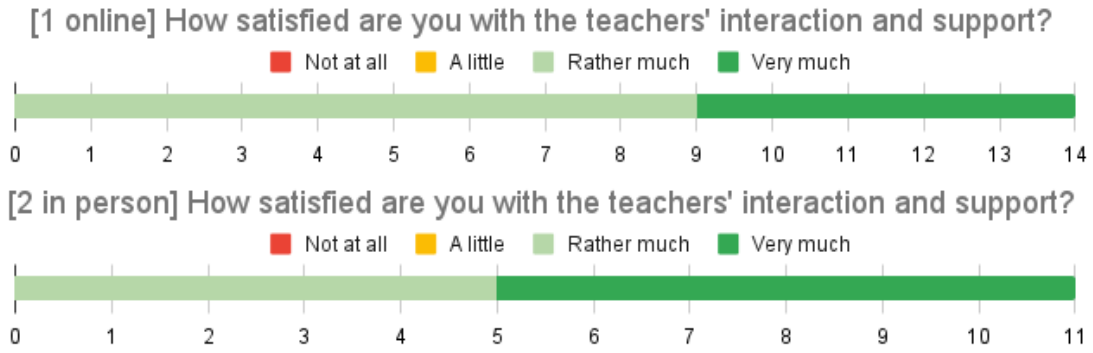


Figure 14.7: Students' evaluation of the interaction with the teachers

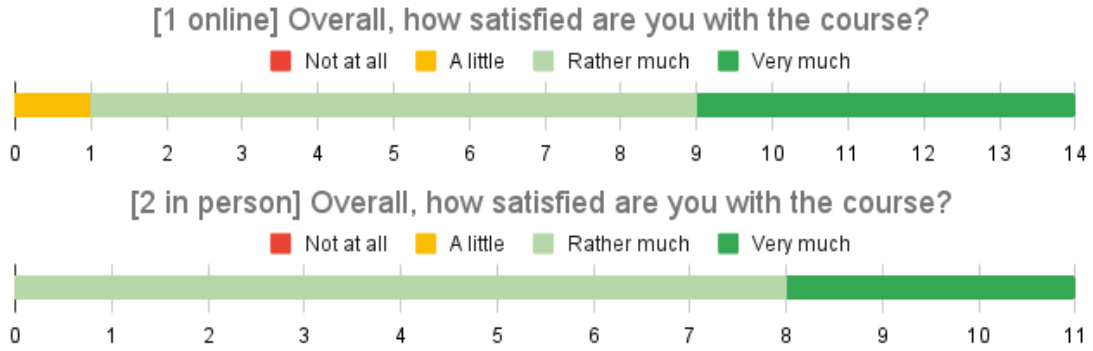


Figure 14.8: Students' overall satisfaction with the course

First iteration All but one of the comments were positive. Most students said the activities were interesting and fun (e.g., "I liked the fact that through Snap! we could play and experiment with cryptography"). According to them, the lessons were engaging and well organized, even at a distance. A student wrote: "I enjoyed myself and really appreciated that there was room for debate. I learned a lot and would have liked it to last longer." Other students also wished the course had been longer. A student would have liked to explore programming with Snap! beyond the boundaries of our playground activities.

The only non-positive comment concerns comprehension difficulties; the student would have liked more "schemes and animations before moving on to more practical work".

We are pleased to report that a student used (one of) the Ceaser cipher playground to encrypt (with a key unknown to us) his positive comment.

Second iteration The open comments from the second iteration were all very positive as well and confirmed the appreciation for the course.

The course was considered stimulating, so much so that some students continued to study

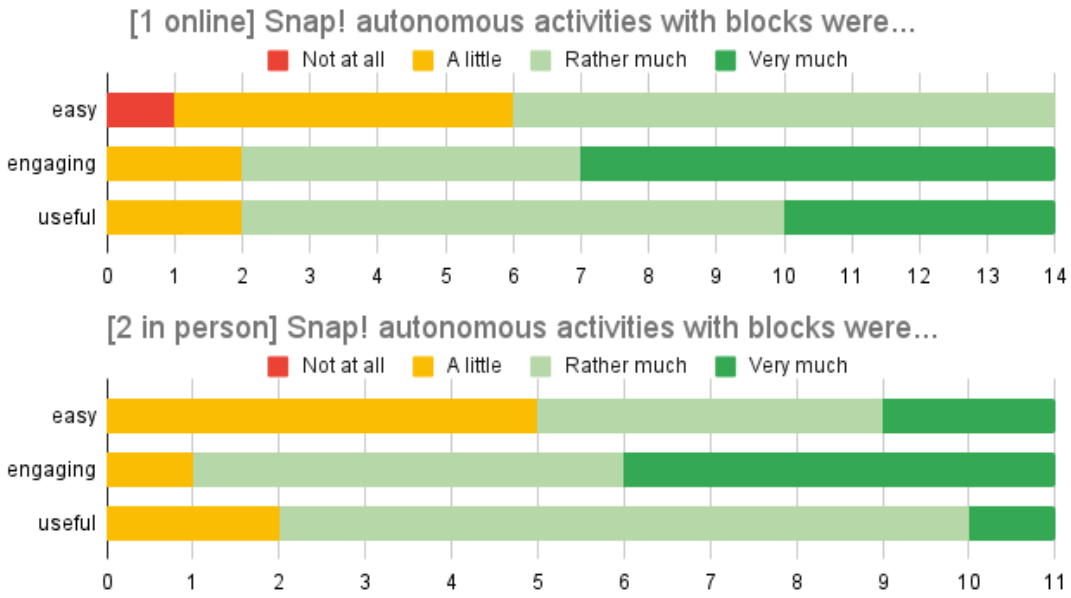


Figure 14.9: Students' evaluation of the activities with Snap! playgrounds

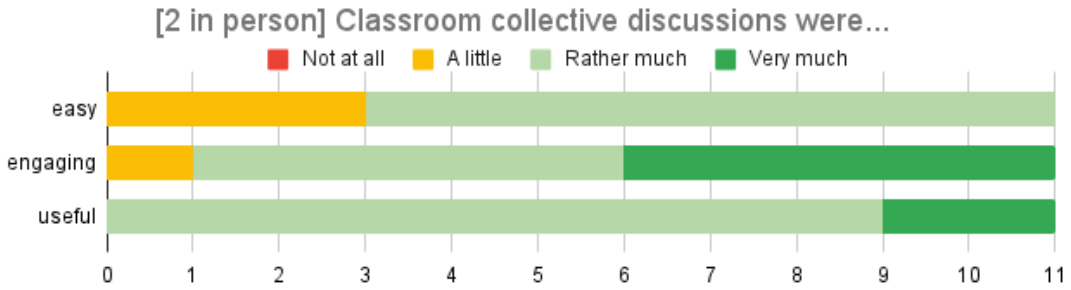


Figure 14.10: Students' evaluation of classroom discussions

cryptography independently. In the words of two students, the course was "more interesting than I could have thought", "because it teaches things they don't teach you in school".

14.4 Results and observations

14.4.1 Methodologies used and two iterations' results

Activities on the Diffie-Hellman protocol were highly appreciated. The simulation of the insecure channel through the meeting public chat was perceived, as we intended, as a helpful metaphor for understanding the protocol. The students found the Diffie-Hellman unplugged

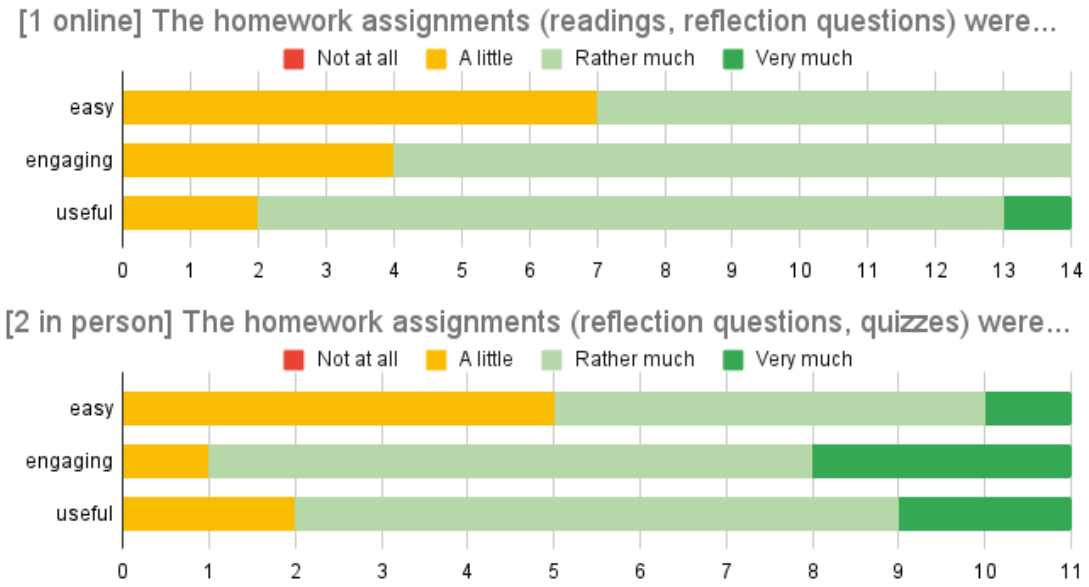


Figure 14.11: Students' evaluation of homework

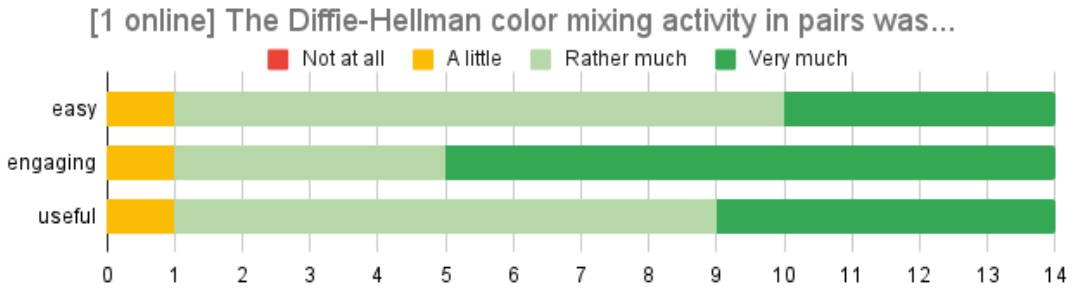


Figure 14.12: Students' evaluation of the activity in pairs on the Diffie-Hellman protocol

activity engaging and fun (see fig. 14.12), confirming our positive impressions.

We consider it a "remote-unplugged" activity because it has almost all the characteristics of a *CS Unplugged activity*, except that it was delivered via technological devices because of the COVID-19 situation. Specifically, such characteristics are: to be *real computer science* – as it presents fundamental concepts and algorithms of informatics; to be based on *learning by doing*; and to be *fun*, *co-operative*, *stand-alone*, and *resilient* to student error (see CS Unplugged [[n.d.]]). However, in our scenario, the technological devices were merely a means of communication rather than a necessary informatics tool for the activity itself. The interactive app (i.e., the "execute-only" Snap! project) made this activity on the Diffie-Hellman protocol concrete and easy to follow, proving to be an excellent solution for the remote context of the course's first iteration. As mentioned, we could not replicate this activity in person at

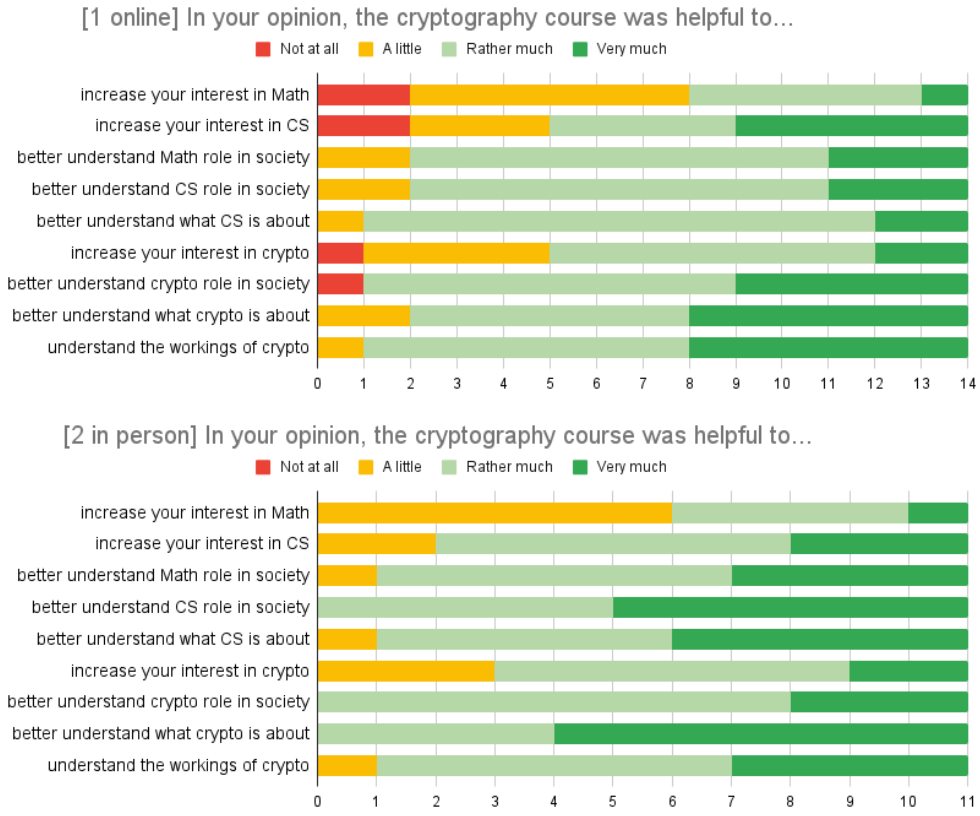


Figure 14.13: Students' self-evaluation of course effects

the second iteration of the course due to a malfunction of the host school's network. In an exclusively in-person scenario, the activity on the Diffie-Hellman protocol could be adapted so that it becomes unplugged for all intents and purposes, as long as an appropriate medium is found to represent the insecure channel (e.g., a blackboard on which everyone can read and write).

The entire course was structured around collective discussions guided by the students' doubts and insights. The space given to these interactions was greatly appreciated (see 14.3.2). If we compare the two iterations, the in-person one registered broader participation in the discussions, which we believe can be attributed to several factors.

- The additional *Lesson 0* – which began with pair and group discussions before even introducing any cryptography concept – early on set up a working mode based on interaction.
- Being in person favored more natural interaction with the teachers and among the students (who did not know each other at the beginning of the course, coming from different classes).

- More creative and open-ended homework fostered debates about the students' different proposals.

The Snap! playgrounds worked well – for the simpler cryptosystems such as Caesar cipher – to gain experience and understanding of the elements of a cryptosystem, some of the possible attacks, and its main limitations (e.g., the computation time required). However, some students found the more advanced playgrounds difficult, especially in the first iteration. In this case, the online context and its limitations weighed in. Not being able to walk around the students' desks led to significant "instructor blindness" (see Lodi et al. [2021b]), making it difficult for us to act as facilitators and provide students with *optimal guidance* (see Taber [2012] and also our 2.5.3.6) while exploring the playgrounds. Indeed, the Snap! hands-on activities ended up being minimally guided (see Tobias and Duffy [2009] and also our review in 2.5.3.5) and, therefore, too difficult, especially for the weaker students. In the second iteration, this negative effect was mitigated. While some students still found the activities with Snap! difficult, all the students vastly enjoyed them, suggesting that they posed the right level of challenge: not trivial but also not overly difficult, and therefore engaging without being discouraging.

For the simpler cryptosystems, the Snap! playgrounds were used to have students experiment before discussing and analyzing those systems. The guiding questions (accompanying all the activities) were essential to get students to focus on precisely those aspects and limitations we needed to highlight. It was only after these hands-on experiences that collective discussions were used to explicit and formalize the most relevant content. This approach was also adopted with the activity on the Diffie-Hellman protocol.

On the other hand, public-key systems, being conceptually more complex, are less suitable for a hands-on, more *constructivist* approach (see 2.5.3). However, even for this more frontal part of the course, the formalization of the schemes and principles occurred only after discussing the ideas and schemes that the students came up with intuitively after the essential ingredients (e.g., the public/private key pair) had been presented.

Coherently with the course objectives, the final assessment [Lodi et al., 2021a] focused on the fundamental ideas of cryptography. Both iterations' results were very positive (see 14.3.1): the students clearly grasped the main contents. We are satisfied with both the citizenship goal (i.e., every citizen should have the tools to understand today's digital society; see particularly 6.1) and the opportunity for university and professional orientation toward informatics and mathematics. The satisfaction questionnaire confirmed the achievement of these two goals, indicating that the students better understood the role of cryptography in society, grasped more precisely what this discipline is about, and saw their interest in cryptography increase.

14.4.2 Learning programming

Our playgrounds can be seen as task-specific languages (see 14.2.6.2) with narrow scopes on specific cryptosystems. They do not aim to (nor could they) teach students how to program (more in 14.4.2). However, they can convey some general principles about programming. For example, the fact that "*programs are assembled out of basic elements, and different orderings*

of elements can sometimes have the same result, and even that the program determines the computer's behavior (there's no magic)" [Yadav and Berthelsen, 2021, p. 186].

Compared to other task-specific programming languages [Yadav and Berthelsen, 2021], our activities more explicitly expose some classic programming concepts (e.g., sequence, function composition, variables, lists). In perspective, more custom blocks could be developed, together with activities requiring more extensive programming, such as using other fundamental elements of structured programming (like conditionals and loops) or "looking inside" the custom blocks provided to understand and adapt them (maybe within a UMC approach; see 2.6.4).

At present, inspecting the code of the blocks in our playgrounds does not have the educational value it could have. In addition to many of Snap!'s predefined blocks, curious students would find some JavaScript code and a few uninteresting workarounds, which we have used to overcome Snap!'s current limitations. For this reason, we plan to design a hierarchy of *notional machines* at different abstraction levels (see [Sbaraglia, 2021] and also chapter 8) so that students can see progressively more detail by inspecting the blocks (thus moving downward the abstraction hierarchy) without being overwhelmed by all the complexity at once.

14.4.3 Suggestions for adoption and adaption

All the content, the learning path, the tools created for our course (e.g., Snap! playgrounds), and the materials (e.g., the narrative text for the final assessment) are available under a free license [Lodi et al., 2021a].

The level of guidance in the hands-on activities can be adjusted. How much to guide the activities or leave them to students' free experimentation can be determined on a case-by-case basis. If the course is in person, instructors can get a clearer picture of students' difficulties and address them immediately while still leaving a high degree of freedom. If the course is held remotely, we suggest more frequent checks and realignments in order to provide students with adequate guidance and support.

Although the course has the big ideas of cryptography as its learning objectives, it is only possible to understand the fundamental ideas of a discipline by dealing with them in (some of) the concrete, albeit simple, scenarios or systems in which they arise (see 4 and also 3.1.1, particularly Voogt et al. [2015]).

Finally, suppose it is possible to allocate more time to the course than we had available. In that case, we suggest that even more time should be devoted to hands-on explorations and discussions rather than covering new cryptosystems (unless such systems are strategic to other fundamental cryptography *principles* instructors intend to teach).

Chapter 15

A Didactical Situation on Interdisciplinary Cryptography

In this chapter, we present an activity to teach the idea of public-key cryptography and make pre-service STEM teachers explore fundamental informatics and mathematical concepts and methods. We follow the *Didactical Engineering* research methodology and rely on the *Theory of Didactical Situations* to design a situation (based on an unplugged activity) about public-key cryptography using graphs. After the preliminary analysis of the content and the constraints and conditions of the teaching context, we conceived and analyzed the situation *a priori*. We specified a *milieu* and the different *didactical variables*. We discussed their impact on the problem-solving strategies participants must develop to decrypt a secret message in the chosen cryptosystem. We implemented our situation and collected qualitative data during the experimentation. We then analyzed the different strategies that participants actually used in the *a posteriori* analysis, which showed the learning potential of the activity. To conceive and develop different problem-solving strategies, the participants needed to explore and understand (at least intuitively) several concepts and methods from mathematics and informatics. They also needed to move between the boundaries of the two disciplines (such as backtracking, stacks, the adjacency matrix of a graph, and matrices for modeling a linear system of equations), also moving between different semiotic and disciplinary registers.

15.1 Introduction

In the last decade, the importance of the introduction of informatics education in pretertiary (or K-12) education has been strongly advocated. Informatics should be recognized as a fundamental, independent scientific discipline to be taught to students (see 3), so they can understand the digital world we are immersed in and become active and informed citizens and, potentially, workers in the ever-increasing digital job market (see 4.1.1 and 4.2.2).

However, in our increasingly complex and rapidly changing world, many criticize the traditional siloed teaching of disciplines in school and advocate a much more integrated, interdisciplinary teaching, particularly for the STEM (science, technology, engineering, and mathematics) fields (see 7.2).

In the context of the IDENTITIES Erasmus+ European Project, a more extensive project about interdisciplinarity in STEM education and pre-service teacher training (see 7.1), we developed a teaching activity on public-key cryptography. The activity was designed for teacher training and was tested during pre-service teacher training events. Furthermore, its content and organization have the potential to be used by teachers for classroom activities or as projects with high school students (although the activity has yet to be tested in school contexts).

The activity we developed aims to teach the big ideas and challenges of public-key cryptography and make participants interact with the interdisciplinary objects (of informatics and mathematics) the activity includes.

We used *Didactical Engineering*, a methodology widely used for decades in mathematics education research. Its main objective is "*the controlled design and experimentation of teaching sequences [. . .] adopting an internal mode of validation based on the comparison between the a priori and a posteriori analyses of these*" [Artigue, 2020, p. 203]. Didactical engineering is an approach to designing learning environments based on the *Theory of Didactical Situations* principles. Theory of Didactical Situations provides a theoretical foundation for understanding the learning and teaching process in the classroom, while Didactical Engineering applies these principles to design learning sequences and environments. In other words, Didactical Engineering is a practical application of Theory of Didactical Situations (see 7.5).

For the content of our activity, we relied on a cryptosystem first described by Fellows and Koblitz [1994], based on the problem of finding a *perfect dominating set* in a random graph. Bell et al. [2003, pp. 209-211] developed a *CS Unplugged activity* for high-school students using that cryptosystem. As we will discuss, while using the same cryptosystem and realizing an unplugged activity, our design differs significantly from the original one.

We choose to design a public-key cryptography activity (based on a computationally hard problem on graphs) for epistemological and educational reasons. Epistemologically, informatics and mathematics are deeply interconnected in the cryptography research field and discipline, and the activity, as we will see, can bring up many topics like algorithms, computational complexity, graphs, matrices, and linear systems. Educationally, informatics and cryptography are well suited to provide *adidacticity*, which is the potential to enable learning independently of teacher interventions (see 7.5.1 and 15.2.2). For example, adidacticity may consist of students being able to test whether an informatics program is correct simply by running it or whether a decryption strategy works by finding a meaningful plaintext from a ciphertext.

Chapter 7 of the review (part II) provides the general context for this work. First, it reports a literature review on interdisciplinarity – within the context of the IDENTITIES project – to better define it (also in terms of *boundaries* between disciplines) and discuss its role in STEM education. Second and more specifically, the main theoretical and methodological underpinnings of our research are presented: Didactical Engineering (7.5.2) within the Theory of Didactical Situations (7.5.1).

Back to this chapter, its sections are organized according to the phases of the Didactical Engineering research methodology. Section 15.2 presents the preliminary analysis of the current epistemological, institutional and didactical context of interdisciplinary and cryptography teaching. Section 15.3 details the chosen cryptosystem's computational, mathematical, and

educational aspects, which are necessary to understand our activity. Section 15.4 describes the design of our didactic situation. Section 15.5 details the *a priori* analysis of the didactical variables and their impact on the different problem-solving strategies and their relative interdisciplinary potential. Section 15.6 describes the implementation of our situation and the data collected. Section 15.7 presents our *a posteriori* analysis in light of the *a priori* one. Finally, section 15.8 discusses our results and offers concluding remarks.

15.2 Preliminary analysis

15.2.1 Institutional analysis

The didactical situation reported in this chapter is part of a module on cryptography for prospective science teachers. The module is one of the outputs of the IDENTITIES European project involving five universities that aims to design novel teaching approaches to interdisciplinarity in science to innovate pre-service teacher education. The project develops and tests innovative teaching modules on interdisciplinary curricular topics (such as cryptography) to explore inter-multi-trans-disciplinary knowledge organizations and to develop interdisciplinary classroom activities and new models of co-teaching. The teaching modules aim to highlight and question the identities of STEM disciplines through reflections on their epistemological and linguistic structures, focusing on the interaction between physics, mathematics, and informatics. Each module must last about 6 hours. Its interdisciplinary content must be socially relevant and potentially suitable for high school students as well. Indeed, the content must be understandable to high school teachers of STEM disciplines without discipline-specific prerequisites. Module activities must be easy to understand for all the participants yet engaging and approachable in the given time.

These interdisciplinary modules were implemented and tested twice in week-long training schools for student teachers. The first training school took place in 2021 and was held online because of the COVID-19 pandemic restrictions. This first remote implementation of our cryptography module informed our design and helped us refine the teaching activity and prepare an observation grid for researchers to use during the implementation. This chapter focuses on analyzing the teaching activity and its implementation during the second training school, which took place in 2022 in physical presence. Twenty-eight student teachers participated, five or six from each partner's institution. They had a disciplinary background (bachelor's or master's degree) in informatics, mathematics, physics, or natural sciences, and experience or motivation in science education[1]. In addition, twenty-one researchers from all the project institutions were involved as both modules' instructors and researchers of the project. The training school was held in English (which was not the native language of any of the participants). Each module was attended by about 14 prospective teachers (out of 28, as it was possible to choose between two alternative modules on different interdisciplinary topics).

Therefore, the institutional constraints for the design of our didactical situation were

[1]The participants had to be enrolled in a master's program (or equivalent course, depending on the national regulations) to become high-school teachers in STEM disciplines.

the following. i) Three hours (out of six) available for the didactical situation during the module. ii) Fourteen participating student teachers with different disciplinary and linguistic backgrounds. iii) No specific disciplinary prerequisites of the participants could be assumed. iv) Participants could use PCs and tablets, and Internet access was provided.

15.2.2 Epistemological and Didactical analysis

15.2.2.1 Interdisciplinarity in STEM and between Informatics and Mathematics

The incredibly rapid development of our digital society has widened the gap between what is taught in schools and what students experience daily. One of the most significant causes may be the rigid discipline-based organization of the school curriculum [Miani, 2021]. STEM movement tries to answer this challenge by proposing the integration of science, technology, engineering, and mathematics in an interdisciplinary and applied approach that deals with real-world problems and problem-based learning (see also 2.6.2). According to the movement, these subjects do not exist in isolation in the real world, and therefore they should not be taught separately [STEM Task Force, 2014, p. 11]. However, the "traditional siloed subject teaching of STEM" is far from being overcome. Many challenges are still to be tackled, like "inadequate teacher knowledge incorporating all STEM fields, and the lack of materials and instructional and assessment support and guidance". Moreover, teachers "*struggle to make connections across the STEM disciplines [. . . and] expressed difficulty in using frameworks from other disciplines [. . .] and felt [. . . not] able to impart meaningful learning*" [Miani, 2021]. Also, it is still fundamental to keep and value the specific characteristics, methods, and ways of thinking of the different disciplines for fruitful interdisciplinary interaction [Barelli et al., 2022]. That is why we designed an activity to make these different disciplinary characteristics emerge in an interdisciplinary context.

In this chapter, we focus on the interdisciplinarity between informatics and mathematics. The disciplines have "strong links and a common history", sharing common foundations, "fields developing at their interface" and "a very similar relation to other sciences through modelling and simulation" [Modeste, 2016, pp. 243-244]. Cryptography is one of the fields that is developing at the interface of informatics and mathematics [Modeste, 2016], and therefore it is a good candidate for our activity.

15.2.2.2 Teaching cryptography in K-12 education

The following is just a summary of what is presented in 6 and particularly in 6.2; it is helpful here to recall what is the more general context of our research on cryptography education. We point out that this context is also shared by both the project that won the SIGCSE grant (see chapter 13) and the cryptography course we developed for high school mathematics (chapter 14).

The Cybersecurity Curricula 2017 [Joint Task Force on Cybersecurity Education, 2018] and K-12 CS Education Standards from CSTA [CSTA, 2017] recommend that graduate programs in cybersecurity include basic cryptography concepts, including symmetric and asymmetric-key ciphers, and suggest hands-on, inquiry-based, unplugged activities for learning.

Nevertheless, a review of ACM education conferences from 2010-2019 [Švábenský et al., 2020] found that most publications on informatics education focus on cybersecurity and often only consider cryptography from a mere technical perspective [e.g., Sommers, 2010; Turner et al., 2011; Brown et al., 2012; Deshpande et al., 2019]. However, some significant indications emerge from a review of the works that deal specifically with cryptography in school settings. Hands-on, inquiry-based activities can improve students' self-efficacy and problem-solving skills [Konak, 2018]. Educational tools that visualize and simulate how cryptosystems work and their vulnerabilities are also frequently used [e.g., Simms and Chi, 2011; Schweitzer and Brown, 2009; Ma et al., 2016; Anane and Alshammari, 2020], but these can be too technical for nonspecialist students and need more interactivity. Unplugged activities that allow students to experience encryption and decryption algorithms, protocols and attacks at a high level without computers have also been proposed and implemented [e.g., Bell et al., 2003; Konak, 2014; Fees et al., 2018], using simple objects and actions to simulate concepts.

Inspired by Bell et al. [2003], our activity involves constructing a graph with a *perfect dominating set* to simulate a one-way function. Such a graph is the base for a cryptosystem that uses elementary arithmetic computations to encrypt a number. Since this is the core of our didactical situation, it is explained in detail later (see 15.3).

15.2.2.3 Didactical aspects of cryptography

Communicating in secret and trying to decrypt messages without knowing the key is engaging and motivating for students [Lindmeier and Mühling, 2020].

Moreover, cryptanalysis has an inherent potential for *adidacticity*, that is, the potential for learning with substantial autonomy left to students' interactions with the problem. Indeed, suppose one is trying to find the secret key of a cryptosystem (where encryption and decryption algorithms are public). In that case, the supposed key can be tested by decrypting the messages that have been encrypted with that key, thus verifying whether the result is the original plaintext message. Similarly, in informatics programming, students can test their program themselves and see if it works by comparing the desired and actual results of the computation without waiting for teacher validation.

Considering these two aspects of cryptography, we have organized a didactical situation based on a public-key cryptosystem.

Based on bibliographic research and the analysis of several proposals, we chose a cryptographic system based on a graph theory problem: the existence of a *perfect dominating set* on a graph. The choice was guided by the need for the activity to be understandable by students with no informatics or cryptography background and to involve interdisciplinary objects like graphs.

In the following, we provide some definitions and formalize the cryptosystem.

15.3 A public-key cryptosystem using perfect dominating sets on graphs

In an encryption scheme, we assume two individuals communicate on a public channel. In order to ensure the confidentiality of their communication, the parties use an *encryption algorithm* to transform a *plaintext* message into an *encrypted* message (a ciphertext). The security of this process is based on (one or several) *keys*, allowing both parties to encrypt and decrypt messages. There are two types of cryptosystems: symmetric (or secret-key) and asymmetric (or public-key). In a symmetric cryptosystem, the encryption and decryption key is the same. The encryption key (public key) and the decryption key (private key) are different in an asymmetric cryptosystem.

In our work, we focus on public-key cryptosystems. The main elements of a public-key encryption scheme[2] are: a key generation algorithm Gen that generates a pair of keys (pk, sk), i.e., a public key and a private key for each user; an encryption algorithm Enc that, given the pk of the receiver and a plaintext message m, outputs a ciphertext message $c = \mathsf{Enc}_{pk}(m)$; a decryption algorithm Dec that, given the sk of the receiver and a ciphertext c, outputs a plaintext $m = \mathsf{Dec}_{sk}(c)$. Both functions Enc and Dec should be easy (that is, efficient) to compute if the keys pk and sk, respectively, are available. The security of the scheme depends on the difficulty (that is, the computational complexity) of computing the function Dec *without* also knowing the secret key sk.

Fellows and Koblitz [1994] proposed an asymmetric cryptosystem based on a difficult problem: the Perfect Dominating Set (PDS) problem.

Let a graph $G = (V, E)$ with V the set of vertices and E the set of edges. A (closed) neighborhood of a vertex $u \in V$ is the set $N[u] = \{v \in V | uv \in E\} \cup \{u\}$, of vertices of V adjacent to u as well as u (in other words, all vertices of distance $\leqslant 1$ from u). A *dominating set* of G is a subset of vertices $S \subseteq V$ such that every vertex of V is included in the neighborhood of a vertex of S. If S is a dominating set of $G = (V, E)$, then every vertex of V is a neighbor to at least one vertex of S, or it belongs to S. If each vertex of V is included in exactly one neighborhood of a vertex of S, then S is said to be a *perfect code*, often referred to also as *perfect dominating set* (noted PDS in the following). Figure 15.1 gives an example of a graph with a PDS.

A practical, useful result is that if a graph has more than one PDS, they all have the same size [Klostermeyer, 2015, p 106].

Thus, the PDS problem is the following [Fellows and Hoover, 1991; Haynes et al., 2013].

PDS Problem

Input: A graph $G = (V, E)$
Output: A PDS of G (if one exists)

In general, deciding whether there exists a PDS in a given graph is an $\mathbb{NP}$-complete decision problem [Klostermeyer, 2015, p. 107], and therefore finding a PDS in a given graph

[2] For a formal definition, see for example Katz and Lindell [2007, p. 366]

(our PDS PROBLEM) is a $\mathbb{NP}$-hard problem[3].

This means that we only know algorithms that take exponential time with respect to the number of nodes, and we do not know if we will ever be able to do better than that. We can use this feature to design a cryptosystem, as we will explain.

For our didactical situation, we used an instance of the PDS PROBLEM, i.e., we have constructed a graph with a PDS. The choice of this graph is crucial, as will be explained later.

Using the PDS PROBLEM, we can design a cryptosystem based on the following two facts:

- given a set of vertices, we can easily construct a graph whose PDS will be this set of vertices;
- given a graph containing a PDS, it is difficult to find the PDS if we only know the graph.

The PDS cryptosystem is the following: Alice and Bob want to communicate confidentially. Bob wants to send a message m (in this case, m is an integer) to Alice. They use the following encryption protocol:

1. Alice builds a graph $G = (V, E)$ with a PDS S. The graph G is Alice's public key, and the PDS S is Alice's private key. Let $V = \{v_1, v_2, ...v_k\}$.
2. Bob chooses integers $m_1, m_2, ..., m_k$ such that $m_1 + m_2 + ... + m_k = m$.
3. Bob assigns to each vertex v_i of V an m_i. We call m_i the *secret value* of the vertex v_i.
4. For each vertex v_i, Bob sums its secret value with the secret values of its neighbors. This new value p_i is called the public value of the vertex v_i.
5. Bob writes on each vertex its public value and deletes the secret values. The encrypted message is the graph G with the public values.

[3]We summarize here, in an informal way, the most relevant ideas. $\mathbb{NP}$-completeness is a vast topic in the study of the computational complexity of problems: for a formal introduction, we suggest Cormen et al. [2022, ch. 34].

We say a problem is in the set $\mathbb{P}$ if we can solve it in polynomial time with respect to the input size (i.e., it can be solved in $O(n^k)$ time for some constant k, with n the input size).

We say a problem is in $\mathbb{NP}$ if we can verify a solution (or, more formally, a solution 'certificate') in polynomial time with respect to the input size. Intuitively, $\mathbb{P} \subseteq \mathbb{NP}$, but if $\mathbb{P} = \mathbb{NP}$ or $\mathbb{P} \neq \mathbb{NP}$ is one of the most famous, relevant and open questions of Informatics.

We focus on a particular set of $\mathbb{NP}$ problems: the $\mathbb{NP}$-complete problems. These problems are considered 'the most difficult $\mathbb{NP}$ problems' because if we find a polynomial-time solution for one of them, then we can solve all the $\mathbb{NP}$ problems in polynomial time. $\mathbb{NP}$-complete problems include relevant problems for today's world. Still, unfortunately, no one has ever found a polynomial solution for any of them, nor has it proven that such a polynomial solution cannot exist.

Formally, the $\mathbb{NP}$-complete set includes only *decision* problems, i.e., those problems whose output is either a 'yes' or a 'no'. For example, as said, determining if a graph has a PDS is an $\mathbb{NP}$-complete problem. Since we are interested here in the complexity of finding an actual instance of that PDS, we are not dealing with a decision problem. However, it should be easy to convince ourselves that our problem is *at least as difficult as the decision problem*. Therefore, we say that finding a PDS on a given graph (our PDS PROBLEM) is $\mathbb{NP}$-hard.

Figure 15.2 gives an example of a graph with its public and secret values.

To decrypt the message, Alice computes the sum of the values on the vertices of the PDS (Alice knows the PDS because it is her private key). Note that the graph G (public key) and the encrypted message (graph G with public values) can circulate without an eavesdropper being able to read the plaintext message (*a priori*). Note also that if the graph has several PDS, then any PDS can be used for decryption.

The system's security is based on the fact that it is ($\mathbb{NP}$-)hard to find the PDS given the graph. Of course, this system is only '*didactically* secure' because simple algebraic attacks are possible [Fellows and Koblitz, 1994], as we will see.

As said, the PDS cryptosystem was first presented in [Fellows and Koblitz, 1994], and an unplugged activity based on it has been included in the Classic CS Unplugged [Bell et al., 2003; 2015].

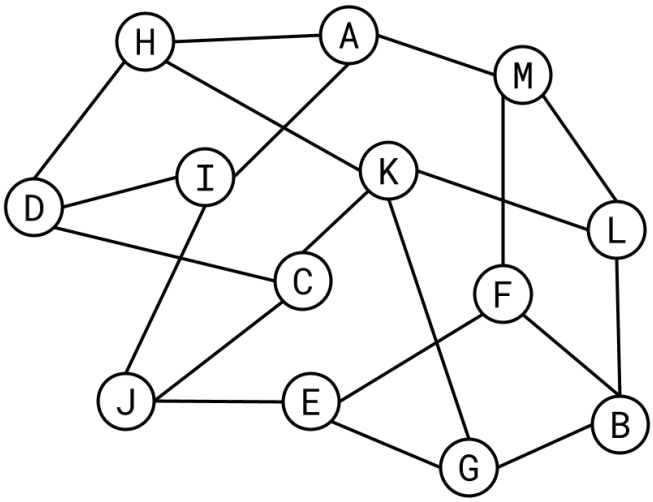


Figure 15.1: $\{I, K, F\}$ is a PDS of this graph.

15.4 Conception

15.4.1 Research purposes

As explained above, this research was developed within a European project whose broader goal is to create innovative teaching modules for pre-service teacher training on interdisciplinarity in STEM fields (with a particular focus on links and interactions between physics, mathematics, and informatics).

In this context, and in order to address the interactions between mathematics and informatics, we have designed, implemented, and analyzed a didactical situation for student teachers about public-key cryptography with the following research purposes.

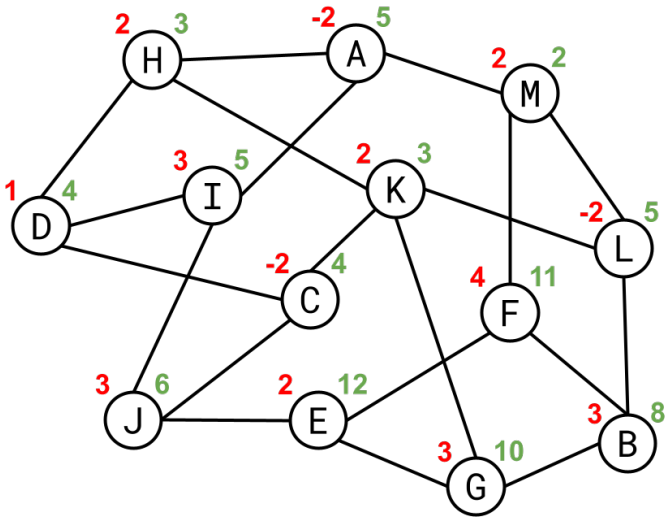


Figure 15.2: Example of an encrypted message using a graph G. Secret values in red and public values in green. The plaintext message m is the sum of the secret values ($m = 19$).

RP1 Examine the different strategies (analyzed both *a priori* and *a posteriori*, after an actual implementation) that student teachers will adopt to decrypt a message starting from different information at their disposal (*access to information* is the main didactical variable of the situation, see 15.5.2).

RP2 Examine how student teachers will interact with different disciplinary and interdisciplinary objects like matrices and graphs, using methods and practices from mathematics and informatics, moving between different semiotic representations[4].

As said, the cryptosystem is already known [Fellows and Koblitz, 1994] and used for didactic purposes [Bell et al., 2003; 2015]. However, we did not just use the original PDS activity. Instead, we have built the didactical situation and the teaching activity around it. Our contribution is the precise organization of its *milieu* (see 7.5.2) and the analysis of the didactical variables that come into play, together with specific choices for their values. Moreover, we propose a classroom implementation that considers the PDS problem's strength as a constructive way to use it for pre-service teacher training.

During the didactical situation, the participants are given a problem (i.e., deciphering a message) that is not broken down into simpler tasks. Because of this (and the careful choices of the didactical variables), the participants need to elaborate specific strategies to address the problem. These strategies (explained in the next section) require using and understanding several concepts and methods from mathematics and informatics (for example,

[4]Intuitively, the *theory of registers of semiotic representation* [Duval, 1995; 2017] is based on the fact that "*there are as many different semiotic representations of the same mathematical object as semiotic registers utilised*" [Pino-Fan et al., 2015].

the representation of a graph by its adjacency matrix) and sometimes the change of semiotic registers.

In what follows, we describe the choices of the didactical variables and the resulting strategies, underlining the concepts and the methods involved.

The more general research purpose of examining this activity's learning potential and impact on related disciplinary and interdisciplinary concepts is left for future work.

15.4.2 The didactical situation

The objectives of the didactical situation are the following.

- Introduce some general concepts and terminology of cryptography (e.g., plaintext and encrypted message, encryption and decryption algorithms, key, attack models, private and public keys, difficult-to-reverse problem, one-way function) and make students understand and explore the principles and issues of public-key cryptography.
- Make students explore and interact with mathematical and informatics concepts and objects on the boundary of the two disciplines (such as graphs, algorithms, and matrices).

The didactical situation is organized as follows.

Step 1: Encryption We explain to the participants the encryption algorithm using a graph G (G is the public key). We do not introduce or explain the notion of PDS (it is not needed to encrypt a message). We do not say that G has a PDS either.

Step 2: Cryptanalysis The participants are divided into three groups. All the groups are given the same encrypted message (i.e., the graph G with public values on it) and asked to decrypt it. Each group is given different information to solve the problem.

- Group A is given the definition of PDS and the (unique) PDS for the given graph G. We do not explain the decryption algorithm. Group A is in the position of a cryptographic engineer who has all the mathematical elements available and needs to combine them to design a public-key cryptosystem.
- Group B is given the definition of PDS and the decryption algorithm (which uses the PDS). They do not know the PDS for the graph G. Group B is in the position of a cryptanalyst carrying out a *person-in-the-middle attack*; that is, the attacker has knowledge of the public-key cryptosystem but does not know the private key.
- Group C has no information other than the encrypted message itself. Group C does not know the decryption algorithm. There is also no reference to the existence of a PDS. Group C is in the position of a cryptanalyst trying to find the plaintext message without necessarily finding the private key.

All groups can independently check whether they have decrypted the message correctly.

15.5 A priori analysis

15.5.1 A priori analysis elements

In the following, we schematically describe the strategies groups may use to decrypt the message considering their available information.

15.5.1.1 Group A

Available information: the definition of PDS and the (unique) PDS for the given graph G. Note that group A is not given any elements on how to use the PDS to decrypt: their goal is to find by themselves the decryption algorithm using the PDS.

Strategy: identifying the neighborhoods of all vertices that belong to the given PDS. Then observe that the intersection of these neighborhoods is empty and that the union of these neighborhoods covers graph G. The neighborhoods can be represented as lists of vertices or graphically as 'stars' on the graph (see Figure 15.3). By the cryptosystem construction, the public value of each vertex is the sum of the secret values of its neighborhood. Thus, the sum of the public values of the PDS vertices is equal to the sum of the secret values of all the nodes, which is the plaintext message.

The definition of PDS is expressed using terminology from set theory. In order to elaborate this strategy, group A needs to interpret this definition on the graphical representation of the graph and make the connection with the encryption procedure. More precisely, they need to deduct what the perfect domination property means for the public values of the nodes. This procedure is not trivial and requires an intuitive understanding of the proof of correctness of the cryptosystem. This way, group A has the potential to explore the central idea of such proof, i.e., the decryption of an encrypted message returns the plaintext message $\mathsf{Dec}_{sk}(\mathsf{Enc}_{pk}(m)) = m$. This can be the object of a formulation phase consisting of making the decryption algorithm explicit and of a validation step of proving the encryption's correctness.

15.5.1.2 Group B

Available information: the definition of PDS and the decryption algorithm (which uses the PDS). Group B knows that there is a PDS in the graph G, but they do not know which nodes are the PDS on that specific graph. This scenario incites the group to try to find the private key (the PDS) using the encrypted message and the public key (G with the public values noted on its vertices). Group B is thus confronted with an instance of the difficult problem of finding a PDS in a graph.

Strategies. We describe three possible strategies for this group. These strategies are interesting because they use different semiotic registers [Duval, 1995; 2017]: the graph representation, the lists of vertices, and the graph's adjacency matrix. These three strategies

amount to a structured, exhaustive search of the subsets of vertices to find a PDS that is known to exist. This search can be done in an organized and structured way (algorithm) or in a more heuristic way based on the same principles.

Strategy 1: Finding stars in the graph. Let a graph $G = (V, E)$ and let S be a PDS of G. This strategy is based on the following ideas:

- Let v be a vertex of V. By the definition of the PDS, in the neighborhood $N[v]$ exists exactly one vertex that belongs to S. Thus, if the v vertex is not in S, then exactly one of its neighbors is in S.
- If a vertex u belongs to S, then (a) the neighbouring vertices of u do not belong to S, and (b) for any neighbour u' of u, the neighbouring vertices of u' do not belong to S either (otherwise u' would be linked to two vertices that belong to S). Thus, if we find a vertex of S, we can deduce that its neighbors and the neighbors of its neighbors are not in S.

Informally, we add step-by-step vertices in a set S in order to find a PDS. When we do not succeed, we *backtrack* to the choices of vertices made to continue the exploration of potential PDS. In informatics, backtracking is a "systematic way to run through all the possible configurations of a search space". It is relevant especially when we "we must generate each possible configuration *exactly* once" [Skiena, 2020, p. 281]. Intuitively, we build a solution incrementally; when we reach a partial solution that can no longer become a correct solution, we abandon the path and backtrack to explore other paths.

Elaborating this strategy first requires understanding the definition of PDS (which is expressed symbolically in set theory language) and then interpreting such definition on the graphical representation of the graph (by drawing subgraphs as stars, see Figure 15.3). Systematizing the steps of the algorithm requires an intuitive understanding of both the properties of domination and perfect domination and the idea of backtracking.

In Algorithm 1, we provide a more formal description of this strategy. Note that, although more rigorous, it is still informal in some operations.

In a heuristic approach to the above strategy, we start with a vertex v of a small degree to deal with a small number of starting cases.

Strategy 2: Lists. Let G be a graph and S a PDS of G. For each vertex of G, we write its neighborhood as a list. We then study these lists to find a set of lists whose intersection is empty and whose union covers the graph G. The basic idea of this strategy is that each vertex of the graph G belongs to precisely one neighborhood of a vertex of S.

Informally, the idea is to incrementally build a collection L of lists ℓ_i, such that the intersection of the $\ell_i \in L$ is empty, while their union contains all the vertices of G.

More rigorously, we formalized this strategy in Algorithm 2, where L is a LIFO stack since the first element removed is always the last one added. In informatics, a *stack* is a collection of elements that implements the LIFO (last-in, first-out) policy: like in a pile of plates, you can only *push* a new plate on top of the stack, or *pop* the plate on the top of the pile.

Algorithm 1 Finding stars in the graph

```
S ← {}
Choose a vertex v, with neighbourhood N[v]
                          ▷ We are sure that v or one of its neighbours is in the PDS
while True do
    Choose t ∈ N[v]
    Add t to S
    repeat
        Mark red all nodes in N[t]
                          ▷ A previous blue may be overridden with red, if necessary
        Mark blue any non-red neighbor of neighbors of t
                          ▷ i.e. the non red nodes in N[x] for all x ∈ N[t]
        if there is an uncoloured vertex w connected to a blue vertex then
                                                          ▷ Backtracking point
            Choose such w and add it to S
            t ← w
            Done ← False
        else
            Done ← True
        end if
    until Done or S is a PDS
    if S is a PDS then
        return S
    else if backtracking is possible then
                          ▷ i.e., if we may choose some other vertex at one backtracking point
        Backtrack (undoing the coloring and the additions to S) to the last possible
backtracking point
        Choose a different w and start again from there
    else
        Remove all the colouring          ▷ We want to iterate again with a new t
        Remove t from N[v]
        S ← {}
    end if
end while
```

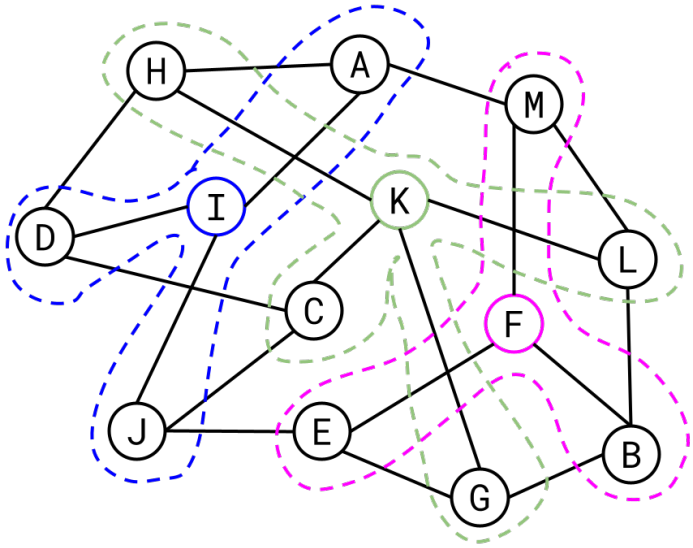


Figure 15.3: We can visualise the 'stars' with the PDS nodes as centres, showing that each node is directly connected to exactly one node of the PDS.

Therefore, the "*order in which plates are popped from the stack is the reverse of the order in which they were pushed onto the stack, since only the top plate is accessible*" [Cormen et al., 2022, p. 254]: the last plate you pushed in is the first you pop out.

Developing this strategy requires understanding the domination and perfect domination properties, expressing these properties using lists, and also an intuitive understanding of a LIFO stack (even if it is not necessarily recognized as such).

Strategy 3: Adjacency matrix of the graph. This strategy consists in writing the adjacency matrix of the graph G and in selecting a set of rows whose sum is a row of 1. Indeed, in the adjacency matrix, for a vertex i, in the corresponding row $l_i = [a_{i1}, a_{i2}..., a_{in}]$ the coefficients $a_{ij} = 1$ if the vertices j and i are connected and 0 otherwise. Note that here $a_{ii} = 1$ for all vertex i (because, in the PDS definition, we are considering closed neighborhoods). Thus, if we find a set of rows whose sum is $[1, 1, ..., 1]$, the vertices corresponding to these lines constitute a PDS (because each vertex of G is adjacent to exactly one of the chosen vertices).

The idea of strategy 3 is very close to that of strategy 2. Still, the register of representation is different: on the same scheme as algorithm 2, we go through the set of rows of the matrix, including or excluding rows, to find a subset of rows whose sum is [1, 1, ..., 1].

Developing this strategy requires, once again, understanding the properties of the PDS definition and expressing these properties using the adjacency matrix.

Algorithm 2 Merging lists

$V \leftarrow$ the set of vertices of G ▷ Vertices are enumerated starting from 1
for all $v_i \in V$ **do**
  $\ell_i \leftarrow N[v_i]$ as a list
end for
$L \leftarrow$*emptystack* ▷ L is a LIFO stack; any item in L is a list ℓ_i
$k \leftarrow 0$
while $\bigcup L \neq V$ **do** ▷ $\bigcup L$ is the union of all the lists ℓ_i in L
  if there exists $i > k$ such that $(\bigcup L) \cap \ell_i = \emptyset$ **then**
    Let i be the min index that satisfies the condition
    Push ℓ_i onto L
    $k \leftarrow i$
  else
    Pop (remove) the top item ℓ_j from L
    $k \leftarrow j$
  end if
end while
return the set $\{v_i : \ell_i \in L\}$ (that is a PDS)

15.5.1.3 Group C

Available information: no information other than the encrypted message. Group C only knows the encryption algorithm and does not know that a PDS exists in the graph nor how it can be used to decrypt. This group is asked (implicitly) to search for possible flaws in the cryptosystem without necessarily searching for the private key.

Strategy: starting from the encrypted message, we form a linear system as follows. For each vertex v, of public value p_v and neighbourhood $N[v] = [v, v_1, ...v_k]$, we write the equation $x_v + x_{v_1} + ... + x_{v_k} = p_v$ where x_i is the secret value of vertex i. This equation translates the encryption step that allowed passing from private values to public values. We thus build a system of linear equations with as many equations and unknowns as there are vertices in G. The solution of the linear system is the tuple of all secret values $[x_1, x_2, ..., x_n]$, whose sum is the plaintext message m.

The linear system can be formed using the graph's adjacency matrix G or by writing the linear equations for each vertex by hand. We highlight that, in this activity, there is a correspondence between the adjacency matrix (one of the standard ways to represent the graph data structure in informatics [Cormen et al., 2022, p. 549]) and the matrix equation that can be used to solve the linear system associated with the encrypted message on the graph. For example, the graph in Figure 15.1 can be represented by the following adjacency matrix (note that, as said, the diagonal is all 1s because, even if edges from each node to itself are not drawn, each node is a neighbor of itself in the PDS definition).

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	1	0	0	0	0	0	0	1	1	0	0	0	1
B	0	1	0	0	0	1	1	0	0	0	0	1	0
C	0	0	1	1	0	0	0	0	0	1	1	0	0
D	0	0	1	1	0	0	0	1	1	0	0	0	0
E	0	0	0	0	1	1	1	0	0	1	0	0	0
F	0	1	0	0	1	1	0	0	0	0	0	0	1
G	0	1	0	0	1	0	1	0	0	0	1	0	0
H	1	0	0	1	0	0	0	1	0	0	1	0	0
I	1	0	0	1	0	0	0	0	1	1	0	0	0
J	0	0	1	0	1	0	0	0	1	1	0	0	0
K	0	0	1	0	0	0	1	1	0	0	1	1	0
L	0	1	0	0	0	0	0	0	0	0	1	1	1
M	1	0	0	0	0	1	0	0	0	0	0	1	1

This is precisely the matrix A in the matrix equation $A\mathbf{x} = \mathbf{b}$ (that represents the linear system of equations that can be used to find the secret values given the public values on the graph) where

$$A = \begin{pmatrix} 1&0&0&0&0&0&0&1&1&0&0&0&1 \\ 0&1&0&0&0&1&1&0&0&0&0&1&0 \\ 0&0&1&1&0&0&0&0&0&1&1&0&0 \\ 0&0&1&1&0&0&0&1&1&0&0&0&0 \\ 0&0&0&0&1&1&1&0&0&1&0&0&0 \\ 0&1&0&0&1&1&0&0&0&0&0&0&1 \\ 0&1&0&0&1&0&1&0&0&0&1&0&0 \\ 1&0&0&1&0&0&0&1&0&0&1&0&0 \\ 1&0&0&1&0&0&0&0&1&1&0&0&0 \\ 0&0&1&0&1&0&0&0&1&1&0&0&0 \\ 0&0&1&0&0&0&1&1&0&0&1&1&0 \\ 0&1&0&0&0&0&0&0&0&0&1&1&1 \\ 1&0&0&0&0&1&0&0&0&0&0&1&1 \end{pmatrix} \quad \mathbf{x} = \begin{pmatrix} A \\ B \\ C \\ D \\ E \\ F \\ G \\ H \\ I \\ J \\ K \\ L \\ M \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 5 \\ 8 \\ 4 \\ 4 \\ 12 \\ 11 \\ 10 \\ 3 \\ 5 \\ 6 \\ 3 \\ 5 \\ 2 \end{pmatrix}.$$

Groups A and B can also be tempted to use linear systems too (even if, for group A, this means not using the private key, which is available).

Unfolding this strategy requires interpreting the cryptosystem as a linear system and examining its resolution. Note that the solution to the problem does not necessitate the resolution of the linear system but just finding the sum of all secret values; this can be done by finding the rows corresponding to the PDS nodes (for example, using the third strategy of group B). The concepts that come into play when elaborating this strategy are matrices, linear systems and their resolution, and also the correspondence of the adjacency matrix with the system's matrix.

15.5.2 Didactical variables

This section identifies the didactical variables to choose relevant values for our learning objectives. These variables have been identified through the study, design, and development of the problem and pre-experimented with volunteer students.

Access to information. In our didactical situation, the main didactical variable is the access to information which differs for the three groups. We have analyzed in the previous section the possible strategies that correspond to different values of this variable.

The type of the graph G. It should be hard to find the PDS in the graph G. So one should exclude certain types of graphs for which it is known that the PDS problem is not hard. For example, if the graph is a tree, a fast algorithm exists that solves the PDS problem [Klostermeyer, 2015, p. 107]. The PDS problem is hard for planar graphs, but we observed that using non-planar graphs makes the problem visually more difficult for the participants. Therefore, we decided to use a non-planar graph.

The size of the graph $\|V\|$. The graph must satisfy specific criteria that make it usable when dealing with humans. More precisely, it must be large enough so that an exhaustive search of the PDS will be hard or tedious. At the same time, it must be small enough so that writing the linear system generated would still be possible for participants.

The graph's maximum degree and the difference of degrees between vertices. A significant difference of degrees between the vertices of the graph potentially influences the starting point and the execution of Algorithm 1; the participants tend to consider that the vertices that have a degree 'too low' or 'too high' have "special" properties and they usually start the algorithm from those nodes. In order to avoid this effect, it is desirable to use a graph "almost" regular. Note that if the graph is regular (i.e., all nodes have the same degree k), there is an easy solution to get the plaintext message that does not require finding the PDS. If we add all the linear system equations and divide by $k+1$, we get the plaintext message (because each m_i will be added precisely $k+1$ times in encryption).

Moreover note that the size of the PDS $\|S\|$ is always in the interval $\frac{\|V\|}{\Delta+1} \leqslant \|S\| \leqslant \frac{\|V\|}{2}$ where V is the set of vertices of the graph G and Δ is the maximum degree of the vertices of G. For a given number of vertices, if the size of the PDS is close to the minimum value, the vertices that belong to the PDS have more neighbors. For our experimentation of the didactical situation, we have chosen a graph with 22 vertices and $\|S\| = 4$.

The plaintext message and its composition. The plaintext message is a positive integer number. This number is subsequently decomposed into the values m_i (secret values) such that $\Sigma_{i=1}^{n} m_i$ with $\|V\| = n$ the size of the graph. Then $m_i \in \mathbb{Z}$; this can be repeated. We have chosen a decomposition in m_i where the absolute value for all m_i is small not to add cognitive difficulty for the participants. In our case, a number between 20 and 100 is a reasonable choice.

Using (or not) a computer algebra system. We chose to give the possibility of using a computer algebra system to solve the linear system. Its use was optional and only proposed if the participants had independently come up with the idea of writing the linear system.

15.5.3 Learning potential

Following the strategies presented in this section, students have to: i) translate the PDS properties of the graph into properties of lists, matrices, and the visual graph representation; ii) do an exhaustive structured search in the matrix or list space or the graph representation (intuitively understanding the idea of LIFO stacks and backtracking); iii) understand and justify why their strategies are correct.

When dealing with a linear system, they may try to reduce it, determine if there is a unique solution or many, and reflect on the complexity of solving linear systems.

Therefore, in this situation – because of the retroactions with the milieu – students have to mobilize concepts, methods, and practices from mathematics and informatics to overcome the obstacles they encounter. Doing so, they also need to move between different semiotic representations [Duval, 1995; 2017] of the interdisciplinary *boundary objects* involved (i.e.,

"artifacts [that] can fulfill a specific function in bridging intersecting practices" [Akkerman and Bakker, 2011, p. 134]; see 7.3), such as matrices and graphs. We believe this allows students to grasp the challenges of public-key cryptography and better understand the concepts, methods, and practices involved in the situations and their interdisciplinarity as well.

15.6 Realization, observation and data collection

In 2021, we piloted an early implementation of the cryptography module entirely online via video conference, which particularly impacted the modes of interaction. In particular, group activity suffered. It was hampered by the less natural interactions (further complicated by the use of English, which was not the native language of any of the participants) and the inability to physically work together on the graphs (even though they were available online, in collaborative editors). However, this preliminary implementation helped us improve the didactical situation. Indeed, we deepened the *a priori* analysis, developed a design more consistent with the preliminary analysis, and better defined the practical organization (timing, mode, materials).

We present the implementation of the cryptography didactical situation, exactly as described in section 15.4, which took place as part of the 2022 school for pre-service teachers, this time held in person (see 15.2.1).

The experimentation took place during a one-day (6 hours) session that included a preliminary presentation on symmetric and asymmetric-key cryptography (and its use and relevance in our society), the didactical situation (3 hours), and a collective reflection on the interdisciplinary aspects that emerged from the groups' work, led by the instructors. The teaching material for the entire module, including the situation, is available at https://identitiesproject.eu/cryptography/. Here we focus on the didactical situation itself.

The didactical situation (about 3 hours) constitutes its active learning part. The core of the didactical situation is an autonomous group activity (about 1 hour): a decryption challenge on which each group is then asked to report back to the other groups.

The participating student teachers were organized into the three groups (A, B, and C) needed for implementing the didactical situation, as described in 15.5.1. The groups of 4 or 5 were composed so that the members were teachers of all the different disciplines (mathematics, informatics, physics, chemistry, and various engineering branches) and nationalities (French, Greek, Italian, and Spanish) and balanced by gender. All participants had a bachelor's or master's degree and were enrolled in a master's program (or equivalent course, depending on the national regulations) to become high-school teachers in STEM disciplines.

After the high-level introduction to symmetric and asymmetric cryptography, the didactical situation began. All three groups were given the same encrypted (with the PDS cryptosystem) message (Fig. 15.4) to decrypt in one hour but starting with different information. The groups were also asked to pay attention to their solving strategies and keep track of difficulties and results so that they could later present their group's work to everyone (10 minutes of presentation and 5 of Q&As). English was used both for interacting with participants and presenting their groups' work.

During this first experiment of our didactical situation, the participants were quickly

engaged in solving the problem. The choices of didactical variables and the organization of the environment stood appropriate: in particular, few interventions by the researchers were necessary during the groups' autonomous work. The mathematics and informatics works in the different groups were generally consistent with the expectations of the *a priori* analysis, and the choices of didactical variables had the expected effects.

One researcher was associated with each group to observe mainly the development of decryption strategies and also the use of the different disciplinary languages (from mathematics and informatics but also the other disciplines of the group members), and the communication dynamics of the groups. This observation aimed to verify whether the implementation of the situation had provided developments consistent with the *a priori* analysis to capture the students' interactions with different disciplinary and interdisciplinary concepts, objects, and methods between different semiotic representations. To support the observation, the researchers relied on a grid, a product of the *a priori* analysis phase (also informed by the 2021 preliminary online experimentation). Such a grid helps the researchers observe for each group three main dimensions: group work and communication dynamics, strategies for solving the decryption problem, and linguistic and epistemological interdisciplinary elements. The grid is provided in appendix B.1. In addition to the observations collected by the researchers, all the sessions were filmed.

An informed consent explaining the objectives of the research, the data collection tools, and the commitment to process data in a pseudo-anonymous manner was provided and signed by all participants.

15.7 A posteriori analysis

The a priori analysis (15.5) was supported by the observations collected during the implementation: the participants tried almost all of the problem-solving strategies we envisioned and described. The student teachers did not formulate the strategies exactly as they were presented in the a priori analysis. Also, not all the groups were able to solve the problem despite using these strategies. However, they tried parts of all the envisioned strategies and seemed to understand (at least partially) the main ideas of how and why these strategies are correct. However, the data collected showed that all groups tried (parts of) all the expected strategies. Moreover, the participants' presentations and following discussions demonstrate their intuitive understanding of the strategies and why they are correct. In this regard, the successful outcome of the instructor-led institutionalization benefited from each group's experience being shared with everyone through the presentations and deepened in the discussions.

More specifically, group C was given the encrypted message and no other information (see 15.5.1 for the information given to each group). The participants solved the problem by formulating a system of linear equations (22 equations and 22 variables) and solving it with an online automatic linear solver[5]. The solver was suggested by the observing researcher only after the group had formulated the system of equations and decided that they would try to

[5]Built by us in Python (https://lodi.ml/solver) with the SymPy library (www.sympy.org).

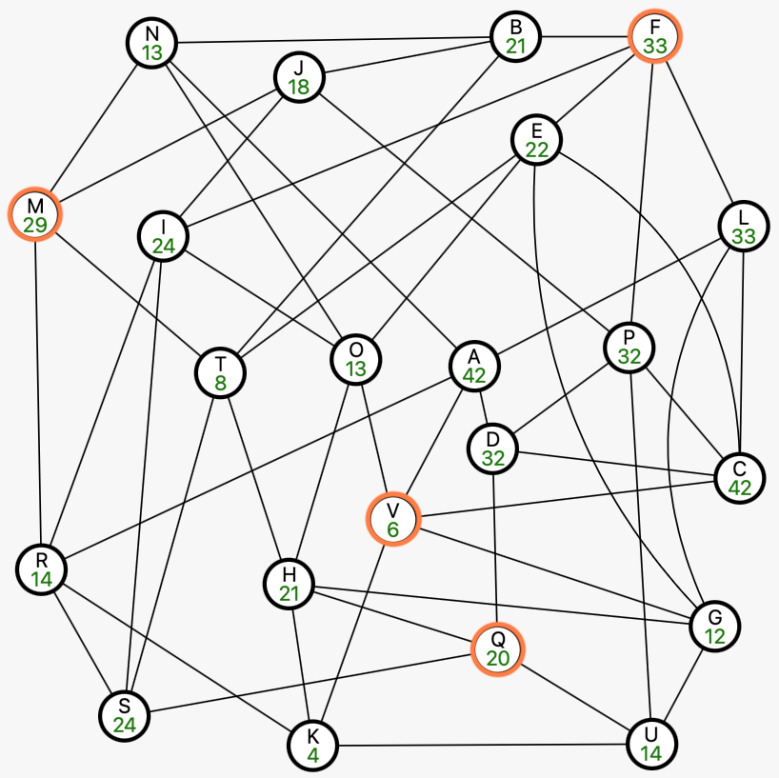


Figure 15.4: The graph with the public values in green (i.e., the encrypted message) chosen for our experimentation. In orange, the PDS of the graph (i.e., the private key).

solve it. The opportunity to use the software tool strongly influenced their strategy to solve the problem. The size of this linear system makes it difficult to solve it by hand. Without an automatic solver available, the students would have probably tried to solve the problem by another strategy, such as reducing the system of equations. They were the only group to solve the didactical situation, finding the plaintext message encrypted in the graph (an integer number). Also, they used the cryptographic language correctly.

Group A was given the definition of *perfect dominating set* and the actual PDS on the graph, and they had to figure out how to use this information to decrypt the secret message. The researcher observing did not explain the definition of PDS, nor the decryption algorithm was presented. The participants had to figure out by themselves how to use the PDS to decrypt the secret message. This scenario brought them to write down the linear system by interpreting the encryption algorithm and trying to make the connection between the system and the PDS definition. We observed that understanding the PDS definition presents several difficulties. First, the PDS definition is complex and structured in three steps. First, there is the notion of *domination*; second, the notion of *dominating set*; third, the notion of *perfect dominating set*. *Domination* is a symmetrical property (i.e., if node a dominates node b, then b also dominates a), while in natural language, domination is usually non-symmetric.

Comprehending and using this definition was not easy for the students dealing with this property for the first time. In addition, domination was defined by the following: 'A vertex v of a graph G dominates vertex u if either $v = u$ or there is an edge from v to u.' We observed that 'if $v = u$' was not interpreted as 'vertex v dominates itself', as was intended. Instead, the participants thought this was related to the public values written on the nodes: their interpretation was that a vertex dominates the vertices with the same public value. This may be related to mathematics's different uses of the equality symbol. In mathematics, it is common to refer to length, width, and measure of quantities in phrases such as *the length of side v is 3 cm, and we write $v = 3$*; the equality symbol is used to give the value 3 to the variable v, i.e., a typical assignment from an informatics perspective. We also use it to express the equality of values, so we can write $u = v$ if they have the same value. Moreover, the equality symbol in algebra is often used to express that two different letters (like u and v) refer to the same object, as we did in defining a PDS. Furthermore, although we thought group A had the "easiest" task since they had the private key at their disposal (i.e., the graph's PDS), their task revealed less trivial and straightforward than expected. Finding how to use the PDS in order to decrypt requires three main steps:

1. understanding the definition of PDS
2. formulating the linear system based on the encrypted message
3. translating the PDS properties (on the graph) into properties related to the equations of the PDS nodes

The last step required a change of semiotic registers and was particularly tricky for the participants. We observed that they spent much time understanding the definition of the PDS and formulating and reducing the linear system but experienced difficulty connecting the two.

Group B was given the PDS definition and the decryption algorithm (without revealing the graph's PDS). As was expected, the participants started looking for the PDS in the graph. They partially tried all three algorithms presented in the a priori analysis. Note that all three algorithms are not polynomial: they are a 'structured' exhaustive search of the PDS in the graph. Therefore, the objective of the situation is not to solve the problem but to translate the PDS properties into algorithmic steps on the three different registers used (the graph, the lists, and the adjacency matrix). In that sense, the students succeeded the task. More precisely, group B started with the list algorithm: they formed the list of neighbors for every node, interpreted the PDS properties with the task of finding a number of lists with empty intersections whose union covers the graph, and started comparing lists. In order to make a more efficient comparison, they decided to take into consideration the lists' size, too. Given that the graph has 22 nodes, if they had a union of lists with size X, they only considered lists with a cardinal of size $\leq 22 - X$ for the following step. As a next step, instead of ordering the lists to facilitate the comparison, they decided to use a matrix, which led to forming the graph's adjacency matrix. As one of the students stated, "with the matrix, we see the connections between the nodes more easily". Algorithm 1 came up in the last part of the situation in a different form. Instead of choosing a starting node (assumed in the PDS)

and erasing its neighbors and its neighbors' neighbors, the students decided to note all paths of length 2 starting from a node (assumed in the PDS) and then all the paths of length 3, which would give them the possible candidates for the second node in the PDS. This idea was not pursued because of the limited time allotted for autonomous group work.

Institutionalization of the solution strategies that emerged and the cryptographic elements involved was done at the end of the implementation. The goal was to transpose some elements of the problem solutions in order to complete the description of the cryptosystem involved and present the cryptographic concepts brought into play. We observed that almost all participants participated in the conversation and the final discussion about interdisciplinarity. They all seemed to have developed an intuitive understanding of the problem-solving techniques and the main ideas behind the strategies that emerged from the groups' work. Although there were some language misunderstandings during the problem-solving process, we observed that the students could properly use the terminology used about graphs, cryptography, and linear system resolution during the discussion. We conjecture that even if their respective group was not successful in decrypting the message, listening to all the groups' presentations (and Q&As that followed) and the instructor-led institutionalization helped the participants form a viable mental model of the elements involved.

To conclude, the groups' work resulting from the implementation supports our a priori analysis of the didactical situation. The participants were strongly involved in problem solving, and our observations (regarding the retroactions and the strategies used) indicate that our organization of the *milieu* and our choices in terms of *didactical variables* proved effective. Indeed, the different configurations of the didactical variables across the groups produced the results predicted by the a priori analysis for those configurations. Also, researchers needed few and small interventions to support the groups' autonomous work.

15.8 Discussion

In this chapter, we presented the study of a *didactical situation* on cryptography between informatics and mathematics, designed, implemented, and analyzed using the *Didactical Engineering* methodology within the *Theory of Didactical Situations*.

The situation's implementation showed a learning potential of fundamental concepts, methods, and ideas not only of cryptography but also of mathematics and informatics. Given the nature of the designed problem-solving activity, the participants need to conceive and develop strategies to solve the problem. To develop these strategies, they need to explore and understand (at least intuitively) several concepts and methods from mathematics and informatics (such as backtracking, LIFO stacks, the adjacency matrix of a graph, and matrices for modeling a linear system of equations). The participants must also move between semiotic registers by interpreting the properties of the PDS definition written in set theory language, graphically, or using lists.

The choices of the values for the didactical variables are essential in this sense: for example, the graph (its size, the vertices degrees, and its graphical representation) does not allow the participants to find the PDS by trial and error and reveals the hardness of the problem while still allowing them to form a linear system by hand; also, the number of PDSs

in the graph is closely related to the existence of a (unique) solution of the linear system. The participants are restricted in their retroactions by those elements and, therefore, need to explore the strategies in order to solve the problem.

Moreover, we conjecture that our didactical situation has the learning potential to introduce topics like the complexity and correctness of algorithms, as well as to work on graphs, dominating sets, linear systems, and matrices and their representations in mathematics and informatics. To illustrate that, we paraphrase and report some questions the participants discussed within their groups while trying to solve the problem.

- *Is there always a perfect dominating set in a graph? And a dominating set?*
- *How complex is solving a linear system?*
- *Are the graph algorithm and the list algorithm more efficient than the brute force solution?*
- *What is the relationship between the linear system and the PDS?*
- *Why is decoding with a PDS correct?*

15.8.1 Future work

The research work of the IDENTITIES project (of which the current work is part) is still ongoing (see 7.1). In this chapter, we have analyzed our observations of the implementation of the didactical situation on cryptography. We conjecture that students were able to grasp the challenges of public-key cryptography and develop a better understanding of the interdisciplinary objects involved. Nevertheless, we still have to refine this analysis rigorously: transcribe the audio and process the videos to identify and analyze all the steps of the problem-solving procedure in detail. Following, we have to identify all interdisciplinary boundary objects (see 7.3) that come into play and analyze them from an interdisciplinary point of view, for example, by using the Akkerman and Bakker [2011] framework on interdisciplinarity.

A future direction for this work would be to implement our didactical situation with different values for the didactical variables and also to adapt the activity for, and experiment it with, high school students.

15.8.2 Conclusions

To conclude, we successfully used the Didactical Engineering research methodology to design a teaching activity that enables students to explore the idea and the complexity of public-key cryptosystems while interacting with the informatics and mathematics interdisciplinary objects involved in that activity and the related disciplinary concepts. Therefore, to overcome the obstacles students encounter in this didactical situation, they must mobilize concepts, methods, and practices of mathematics and informatics, moving between semiotic representations of interdisciplinary objects.

The specific analysis of the interdisciplinary interactions between pre-service STEM teachers and the model of interdisciplinarity that can emerge from this kind of activity is part of the larger project and will be analyzed in future works.

Part V

Conclusions, Appendix and Bibliography

Chapter 16

Conclusions and Future Works

16.1 Introductory programming

We developed a proposal for a learning model to support the learning of introductory programming by drawing on the educational literature that we found most promising, particularly about notional machines and Productive Failure. Delving into both, we left notional machines aside but only from an operational point of view; their flexibility as educational devices has always served us well in the conception and design of subsequent research initiatives and interventions. In particular, analyzing Productive Failure led us to identify what we had recognized in our experience as informatics educators: the necessity mechanism, a learning mechanism with intriguing educational potential.

Future work. *That of notional machines remains an exciting research topic to explore both theoretically and concretely, to use them not only to model computational systems related to programming but also to model informatics systems in other areas (e.g., cryptosystems).*

The study of Productive Failure and PS-I approaches, in general, helped us to precisely define the necessity mechanism from which we developed our learning design. The constant comparison with Productive Failure allowed us to recognize the specificities of our Necessity Learning Design due both to the specific requirements of introductory programming and to the nature of student solutions, i.e., programs that are "living", interrogable artifacts. These insights have enabled us to define a *necessity sequence* precisely.

Then, the further exploration of abstraction in the perspective of learning programming allowed us to recognize the inevitable changes in the abstraction level that a student undergoes in an introductory programming path. The difficulty of these abstraction changes (both up and down) clarified further the ideal moments to use NLD and better defined its educational scenarios and requirements.

The school experimentation of Necessity Learning Design that followed brought many new insights, revealing its potential and also notable limitations.

Future works. *The concrete use of NLD has revealed a flaw in the design of one of our necessity sequences. In general, this pushes for the need to test them to assess their real consistency with the goals and requirements of our learning design.*

In addition, the data collected from the school experimentation (whose preliminary light analysis has been presented) need to be systematically analyzed to give more precise answers to our research questions, particularly on the impact of NLD use on learning.

More generally, the attempt to balance student autonomy and scaffolding has guided our other research efforts, yielding interesting preliminary results in pilot experiments in primary schools and the CS1 course we teach. In all these initiatives, whenever possible, we have always tried to stimulate – though in less structured ways than those of NLD – the necessity mechanism to motivate and prime students to learn the next thing. Empirically, we have always detected a positive motivational and cognitive response in the students involved.

16.2 Informatics for all

While, on the one hand, we tried to distill the big ideas of cryptography, drawing on the principles and modes of the Big Ideas of Science and CS education, on the other hand, we already recognized some transformative principles and ideas of cryptography. We constructed a cryptography course for non-informatics high school students to help them grasp the relevance of such principles and ideas and their essential scientific elements through hands-on and autonomous yet carefully scaffolded activities.

Future work. *The process of expert interviews, reflections, and literature analysis that has led us to have a tentative first draft of possible big ideas of cryptography needs to be completed. The next step will be a more stable draft of about ten big ideas to be submitted to a broader community of stakeholders, along with a questionnaire to survey their opinions, which is also yet to be developed.*

The cryptography course was designed and developed with a non-professional but cultural and citizenship perspective, proposing an approach to principles and ways of informatics less technical and demanding than learning to program.

The same approach (including the cryptography context) was also adopted in developing a Didactical Situation for pre-service teachers. The intervention allowed participants to experience and recognize some relevant concepts in informatics (e.g., computational complexity), effectively showing the interdisciplinarity of cryptography between mathematics and informatics.

Future work. *A more accurate qualitative analysis of the group work recordings filmed during the Didactical Situation implementation remains to be done. Such analysis could give more precise answers about the actual interdisciplinary potential of the asymmetric cryptography activities we developed.*

Overall, the data analysis so far confirmed our intuition that less traditional (and less technical) access to informatics can help students better understand the role of informatics in our lives and grasp its most essential principles and ways of thinking. This approach seems promising for developing computational thinking through the study of informatics (and not as a set of soft skills that can be acquired separately).

Appendix A

Material of Necessity School Experimentation

A.1 Instructional material

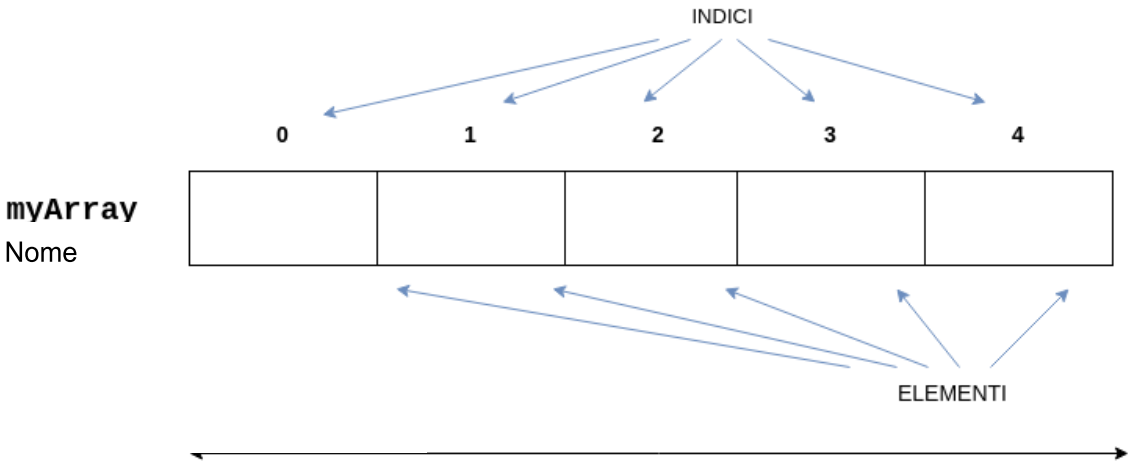
The instructional material for introducing arrays, as described in 10.1.3.4, follows.

Introduzione agli array

Un array è una sequenza di elementi dello stesso tipo. Ad un array è identificato da un nome (come per le variabili), unico per tutta la sequenza.
Per dichiarare un array è necessario specificare il tipo dei suoi elementi (hanno tutti lo stesso tipo) e il numero dei suoi elementi (cioè la lunghezza della sequenza).

Nel seguente frammento di codice viene mostrato un esempio di inizializzazione e dichiarazione di un array:

```
// dichiarazione senza inizializzazione, con lunghezza 5
// (in questo caso obbligatorio specificare la lunghezza)
int myArray[5];
```



Lunghezza dell'array (numero dei suoi elementi)

```
// dichiarazione e inizializzazione con una sequenza di lunghezza 5
// di soli zero (anche in questo caso obbligatorio specificare la lunghezza)
int myArray[5] = {};
```

```
// dichiarazione e inizializzazione con una sequenza specifica, il
compilatore determina la lunghezza (4) a partire dal numero di elementi con
cui l'array viene inizializzato
int myNewArray[] = {42, -3, 16, 1};
```

La lunghezza dell'array è fissata al momento della dichiarazione e non può cambiare durante l'esecuzione del programma.

Il solo nome dell'array lo rappresenta tutto, ma si può anche accedere ai singoli elementi in base alla loro posizione nell'array; la posizione è anche chiamata indice dell'elemento.
Il primo elemento dell'array ha posizione 0.

Il nome dell'array e una posizione consentono di accedere (leggere e/o modificare) all'elemento dell'array a quella specifica posizione. Per accedere ai singoli elementi si usa l'operatore di selezione `[ ]`.
Gli elementi così selezionati `nomeArray[pos]` possono essere trattati come "semplici" variabili.
Quindi è possibile:

- leggere e usare il valore dell'elemento in posizione pos

```
// stampa del primo elemento di myArray
cout << myNewArray[0]; //stampa 42

// somma del terzo e del quarto elemento di myNewArray
int somma = myNewArray[2] + myNewArray[3]; //somma vale 17
```

- modificare il valore memorizzato in posizione pos (se `nomeArray[pos]` è a *sinistra* di un assegnamento)

```
// modifica del secondo elemento di myNewArray
// dopo questa istruzione myNewArray è {42, 100, 16, 1}
myNewArray[1] = 100;
```

Esempio completo

```
int myNewArray[] = {42, -3, 16, 1};
myNewArray[1] = 100;
// ciclo che stampa tutti gli elementi di myNewArray

for(int i=0; i<4; i++) {
    cout << myNewArray[i] << endl;
}
```

Questo codice stampa:

```
42
100
16
1
```

Questo esempio mostra come sia possibile usare l'indice di un *ciclo for* (con i che va da 0 alla lunghezza meno uno dell'array) per accedere a tutti gli elementi di un array. L'indice rappresenta la posizione degli elementi (ricorda: la prima posizione è 0).

Esempi utili

3 esempi in un unico programma (si affrontano uno alla volta in sequenza): https://onlinegdb.com/gDJVsyTSm

1. *stampa* tutti gli elementi di un array preceduti dalla loro posizione
2. *raddoppia* tutti gli elementi con valore dispari di un array di interi
3. stampa tutti gli elementi di un array con una *funzione*

Punti da ricordare

Array: sequenza di elementi dello stesso tipo

- La sequenza è identificata da un nome unico
- Il tipo dell’array (cioè il tipo di tutti i suoi elementi) va specificato all'atto della creazione
- La lunghezza della sequenza va specificata all'atto della creazione*
- La lunghezza *non* può cambiare

Elementi di un array accessibili per posizione (chiamata anche indice)

- Le posizioni partono da 0
- Si usa il nome della sequenza insieme alle parentesi quadre con la posizione dell'elemento a cui si vuole accedere
- Gli elementi così selezionati possono essere trattati come “semplici” variabili
    - leggere il valore presente a quella posizione
    - modificare il valore memorizzato in quella posizione, se a sinistra di un assegnamento
- Si può usare un ciclo (es. ciclo for) per accedere a tutti gli elementi di un array (es. usando l’indice del ciclo come posizione degli elementi)

A.2 Learning Assessment

The learning assessment about arrays introductory knowledge and skills, as described in 10.1.3.5, follows.

Esercizio 1: rispondi alle seguenti domande

Per accedere ad un elemento di un array devo usare… (1 punto)
(una sola risposta possibile)

A. la posizione dell'elemento
B. il valore dell'elemento
C. il nome dell'elemento
D. l'indirizzo dell'elemento
E. nessuna delle precedenti

Cosa non può MAI mancare nella dichiarazione di un array? (1.5 punto)
(più di una risposta possibile)

A. il suo nome
B. la sua lunghezza
C. il tipo dei suoi elementi
D. i suoi elementi
E. il primo elemento
F. nessuno di questi è indispensabile

Quali tra questi array sono ammissibili? (1.5 punto)
(più di una risposta possibile)

A. Un array che contiene tutti i numeri interi da -10 a 100
B. Un array che contiene tutte le lettere (char) di un cognome
C. Un array che contiene nome, cognome (stringhe) e anno di nascita (intero)
D. Un array che contiene float e char
E. Un array che contiene il numero di studenti di ciascuna classe di una scuola

In quale di queste situazioni è opportuno usare un array? (2 punti)
(più di una risposta possibile)

A. Ricevo in input un certo numero (non noto a priori) di valori e ne devo calcolare la media
B. Ricevo in input 1000 valori (che possono ripetersi) e devo tenere traccia della frequenza di ciascun valore
C. Devo tenere traccia delle temperature per ogni giorno del mese
D. Per un periodo indefinito, ogni giorno ricevo il numero di giocatori attivi su Fortnite e devo sempre sapere il numero di giocatori più alto
E. Ricevo e memorizzo l'anagrafica di uno studente: nome (stringa), cognome (stringa), età (intero), genere (carattere)

Esercizio 2 (4 punti)

Sviluppare una funzione '`somma_pari`' che prende in input un array di interi e la sua lunghezza. La funzione restituisce (come int) la somma dei soli numeri dell'array che sono pari (divisibili per 2). Nel main di esempio sotto, con l'array {6,12,3,4,11,5}, l'output deve essere:

```
La somma dei pari e' 22
```

```
#include <iostream>
using namespace std;

int somma_pari(int numeri[], int lun) {
    // il tuo codice QUI

}

/* Questo main è solo un esempio di programma che usa la funzione
che devi sviluppare tu. Non serve modificarlo. */
int main() {
    int numeri[] = {6,12,3,4,11,5};
    int lun = 6;
    cout << "La somma dei pari e' " << somma_pari(numeri,lun) << endl;
}
```

Esercizio 3 (5 punti)

Scrivere il main di un programma che tiene traccia delle frequenze con cui il giorno di Natale capita nei vari giorni della settimana a partire dall'anno 0 (compreso) fino al 2022 (escluso).

I giorni della settimana sono rappresentati da numeri interi:
0 domenica, 1 lunedì, 2 martedì, 3 mercoledì, 4 giovedì, 5 venerdì, 6 sabato

Per sapere in che giorno della settimana cade Natale in un certo anno, usare la funzione 'calcola_giorno_natale' già fornita (e che non deve essere modificata!)
Ad esempio

```
calcola_giorno_natale(2021)
```

restituisce l'intero 6 (che indica sabato) perché nel 2021 Natale è caduto di sabato

Alla fine, il programma deve stampare per ogni giorno della settimana (0…6) quante volte è stato Natale quel giorno per tutti gli anni dall'anno 0 all'anno 2021 (inclusi).

L'output deve essere simile a questo (le frequenze sono quelle giuste).

```
0: 293
1: 283
2: 294
3: 288
4: 288
5: 293
6: 283
```

L'output indica che, dall'anno 0 all'anno 2021 inclusi, Natale è capitato di domenica (indicata con 0) 293 volte, di lunedì (1) 283 volte,... di sabato (6) 283 volte.

```
#include <iostream>
using namespace std;

/* Questa funzione restituisce il giorno della settimana in cui capita
```

```
il giorno di Natale nell'anno 'anno' preso in input; restituisce
0 per domenica, 1 per lunedì, ..., 6 per sabato. NON modificarla!
(PS non c'è bisogno che tu ne capisca il funzionamento intero) */
int calcola_giorno_natale(int anno) {
    int k = anno%400, anchor = 3;
    if(k >= 0 && k < 100) anchor = 2;
    else if(k >= 100 && k < 200) anchor = 0;
    else if(k >= 200 && k < 300) anchor = 5;
    int y = anno % 100;
    int doomsday = ((y/12 + y%12 + (y%12)/4)%7 + anchor) % 7;
    int g_natale = (doomsday-1);
    if(g_natale == -1) g_natale = 6;
    return g_natale;
}

int main() {
    // il tuo codice QUI

    return 0;
}
```

Esercizio 4 (extra)

Scrivere una funzione '`media_array`' che prende in input un array di float e un intero che rappresenta la lunghezza di quell'array.
La funzione deve calcolare la media degli elementi dell'array **maggiori o uguali** a zero.
I valori minori di zero NON devono essere considerati nel calcolo della media.
Inoltre, la funzione modifica l'array sostituendo tutti i valori negativi presenti nell'array con il valore -9999.
La funzione deve restituire (come float) la media così calcolata.

Nel main di esempio sotto, con l'array {10.1, 0, 0.0, -6.4, 5, -7.3, 13}, l'output deve essere:

```
Media:5.62
```

E l'array modificato sarà: {10.1, 0, 0, -9999, 5, -9999, 13}

```
#include <iostream>
using namespace std;

// la tua funzione QUI

/* Questo main è solo un esempio di programma che usa la funzione
che devi sviluppare tu.
Se vuoi, dopo la chiamata di funzione, puoi aggiungere la stampa
dell'array per verificare che sia stato modificato come richiesto. */
int main() {
    float valori[] = {10.1, 0, 0.0, -6.4, 5, -7.3, 13};
    int lun = 7;
    cout << "Media: " << media_array(valori,lun) << endl;
}
```

A.3 Programming exercises in C++

A.3.1 Approach exercises

A.3.1.1 First approach exercise

```
/*
Completare il main di un programma che effettua 1 milione di lanci
di una monenta e stampa il numero finale di teste e croci ottenute.

Un esempio di output:
teste 500028 - croci 499972

La funzione che simula il lancio di una moneta - lancio_moneta() -
restituendo 0 (testa) oppure 1 (croce), è già fornita ed è sufficiente
usarla correttamente.

INTERESSANTE: verificare che i numeri finali di teste e croci
siano simili; più è grande il numero di lanci, più i due numeri
si avvicinano (il che mostra che la probabilità di ottenere testa
o croce è la stessa). */

#include <iostream>
using namespace std;

int lancio_moneta() {
  return (rand()%2);
}

int main() {
  srand(time(NULL));

  /*** codice studenti da QUI ***/
  int teste, croci;
  teste = croci = 0;
  int lancio;

  for(int i=0; i<1000000; i++) {
    lancio = lancio_moneta();
    if (lancio == 0) teste++;
    else if (lancio == 1) croci++; //potrebbe anche essere solo 'else'
  }

  cout << ``teste '' << teste << `` - '';
```

```
  cout << ‘‘croci ’’ << croci << endl;
  /*** FINE codice studenti ***/
}
```

A.3.1.2 Second approach exercise

```
/*
Completare il main di un programma che effettua 1 milione
di lanci di un dado a 6 facce e stampa il numero finale di volte
che è uscita ciascuna faccia

Un esempio di output:
167164 166339 166702 166594 166566 166635

Un esempio alternativo di output:
1: 167164
2: 166339
3: 166702
4: 166594
5: 166566
6: 166635

La funzione che simula il lancio del dado - lancio_dado() - restituendo
un numero da 1 a 6, è già fornita ed è sufficiente usarla correttamente.

INTERESSANTE: verificare che la frequenza con cui esce ciascuna faccia
(cioè i numeri da 1 a 6) è sostanzialmente uguale
alle frequenze delle altre facce;
più è grande il numero di lanci, più le frequenze sono simili (il che
mostra che tutte le facce hanno la stessa probabilità di uscire).
*/

#include <iostream>

using namespace std;

int lancio_dado() {
  return (rand()%6)+1;
}

int main() {
  srand(time(NULL));

  /*** codice studenti da QUI ***/
```

```
  int frq1, frq2, frq3, frq4, frq5, frq6;
  frq1 = frq2 = frq3 = frq4 = frq5 = frq6 = 0;
  int lancio;

  for(int i=0; i<1000000; i++) {
    lancio = lancio_dado();
    if (lancio == 1) frq1++;
    else if (lancio == 2) frq2++;
    else if (lancio == 3) frq3++;
    else if (lancio == 4) frq4++;
    else if (lancio == 5) frq5++;
    else if (lancio == 6) frq6++; //potrebbe essere anche solo 'else'
  }

  cout << frq1 << `` '';
  cout << frq2 << `` '';
  cout << frq3 << `` '';
  cout << frq4 << `` '';
  cout << frq5 << `` '';
  cout << frq6 << endl;
  /*** FINE codice studenti ***/
}
```

A.3.1.3 Third approach exercise

```
/*
Completare il main di un programma che chiede 20 volte
all'utente l'input di un numero intero tra 1 e 12 (rappresenta il mese
di nascita di 20 studenti di una classe). Il programma deve poi stampare,
mese per mese, il numero di studenti nati in quel mese.


Un esempio di output:
gen:2, feb:4, mar:4, apr:0, mag:1, giu:1, lug:2, ago:3, set:0, ott:0, nov:2, dic:1


Un esempio di output alternativo:
2, 4, 4, 0, 1, 1, 2, 3, 0, 0, 2, 1

Si usi la funzione già fornita ( input_mese_nascita() )
che chiede all'utente di inserire un numero tra 1 e 12 e restituisce tale numero.
La funzione continua a chidere l'inserimento finché il dato non è corretto.
La funzione è già fornita ed è sufficiente usarla correttamente.
*/
```

```
#include <iostream>
using namespace std;

int input_mese_nascita() {
  int mese = 0;
  do {
    cout << ‘‘Il tuo mese di nascita (da 1 a 12): ’’;
    cin >> mese;
    if (mese<=0 || mese>12)
      cout << ‘‘NON VALIDO! Inserisci un intero tra 1 e 12’’ << endl;
  } while (mese<=0 || mese>12);
  return mese;
}

int main() {

  /*** codice studenti da QUI ***/
  int gen, feb, mar, apr, mag, giu;
  int lug, ago, set, ott, nov, dic;
  gen = feb = mar = apr = mag = giu = 0;
  lug = ago = set = ott = nov = dic = 0;
  int mese;

  for(int i=0; i<20; i++) {
    mese = input_mese_nascita();
    if (mese == 1) gen++;
    else if (mese == 2) feb++;
    else if (mese == 3) mar++;
    else if (mese == 4) apr++;
    else if (mese == 5) mag++;
    else if (mese == 6) giu++;
    else if (mese == 7) lug++;
    else if (mese == 8) ago++;
    else if (mese == 9) set++;
    else if (mese == 10) ott++;
    else if (mese == 11) nov++;
    else if (mese == 12) dic++; //potrebbe essere anche solo ’else’
  }

  cout << ‘‘gen:’’ << gen << ‘‘, ’’;
  cout << ‘‘feb:’’ << feb << ‘‘, ’’;
  cout << ‘‘mar:’’ << mar << ‘‘, ’’;
```

```
  cout << ``apr:'' << apr << ``, '';
  cout << ``mag:'' << mag << ``, '';
  cout << ``giu:'' << giu << ``, '';
  cout << ``lug:'' << lug << ``, '';
  cout << ``ago:'' << ago << ``, '';
  cout << ``set:'' << set << ``, '';
  cout << ``ott:'' << ott << ``, '';
  cout << ``nov:'' << nov << ``, '';
  cout << ``dic:'' << dic << endl;
  /*** FINE codice studenti ***/
}

/* (un esempio di output)
gen:2, feb:4, mar:4, apr:0, mag:1, giu:1, lug:2, ago:3, set:0, ott:0, nov:2, dic:1
*/
```

A.3.2 PS-I exercise

```
/*
Completa il main di un programma che effettua dieci milioni di estrazioni
del lotto e stampa il numero finale di volte che è uscito ciascun numero.

La funzione che simula l'estrazione di un numero del lotto, restituendo
un intero compreso tra 0 e 89, è già fornita.

Un esempio di output:
    0: 111080
    1: 111224
    2: 110810
    3: 110573
    ...
    88: 111208
    89: 110468


INTERESSANTE: verificare che la frequenza con cui esce ciascun numero
(cioè i numeri da 0 a 89) è sostanzialmente uguale alle frequenze
di tutti gli altri numeri;
più è grande il numero di lanci, più le frequenze sono simili
(il che mostra che, ad ogni estrazione, tutti i numeri hanno
la stessa probabilità di uscire).
*/

#include <iostream>
```

```
using namespace std;

/* simula l'estrazione dei numeri del lotto;
nel nostro mondo di informatici, il lotto
prevede 90 numeri interi, compresi tra 0 e 89 */
int estrazione_numero_lotto() {
    return (rand() % 90);
}


int main() {
    srand(time(NULL));

    /* codice studenti da QUI */



    /* FINE codice studenti */
}
```

A.3.2.1 More challenging version

```
/*
Completa il main di un programma che effettua dieci milioni
di estrazioni del lotto per ciascuna delle 4 ruote (Bologna, Firenze,
Roma, Palermo) e, per ogni ruota, stampa il numero finale
di volte che è uscito ciascun numero.

La funzione che simula l'estrazione di un numero del lotto, restituendo
un intero compreso tra 0 e 89, è già fornita.
E' fornita anche la funzione che stampa ogni elemento di un array,
preceduto dal suo indice.

Un esempio di output:
    BOLOGNA
    0: 111080
    ...
    89: 110468

    FIRENZE
    0: 111019
    ...
    89: 111326
```

```
    ROMA
    0: 111094
    ...
    89: 111081

    PALERMO
    0: 110912
    ...
    89: 110457

EXTRA
E' fornita anche una funzione che dato un array, lo stampa
sotto forma di istogramma.
Sperimentarne l'ultilizzo in relazione alle estrazioni del lotto.

*/

#include <iostream>
using namespace std;

/* simula l'estrazione dei numeri del lotto;
nel nostro mondo di informatici, il lotto
prevede 90 numeri interi, compresi tra 0 e 89 */
int estrazione_numero_lotto() {
    return (rand() % 90);
}

/* stampa tutti gli elementi di un array,
preceduti dal loro indice. */
void print_array(int array[], int dim) {
    for (int i = 0; i < dim; i++) {
        cout << i << ``: '' <<array[i] << endl;
    }
}

/* stampa un istogramma a barre orizzontali in cui gli asterischi,
per ogni valore intero contenuto nell'array,
rappresentano un certo numero di estrazioni. */
void print_istogramma(int array[], int dim) {
    int peso = 10000; //quante estrazioni rappresenta un solo asterisco
    for (int i = 0; i < dim; i++) {
        cout << i << ``:\t'';
        int asterischi = array[i] / peso;
```

```
        for (int j = 0; j < asterischi; j++) {
            cout << ‘‘*’’;
        }
        cout << endl;
    }
    cout << endl;
}


int main() {
    srand(time(NULL));

    /* codice studenti da QUI */



    /* FINE codice studenti */
}
```

A.3.3 Consolidation exercises

A.3.3.1 First consolidation exercise

```
/*
Completare il main di un programma che effettua 1 milione di lanci
di un dado a 6 facce e stampa il numero totale di volte che è uscita
ciascuna faccia. Usare gli array.

Un esempio di output:
167164 166339 166702 166594 166566 166635

Un esempio alternativo di output:
1: 167164
2: 166339
3: 166702
4: 166594
5: 166566
6: 166635

La funzione 'lancio_dado()' che simula il lancio di un dado a 6 facce
(restituisce un numero intero tra 1 e 6) è già fornita.

Se il programma funziona puoi osservare la
LEGGE DEI GRANDI NUMERI
```

```
Ad ogni lancio, le facce hanno tutte la stessa probabilità
di uscire. Più aumentano i lanci, più le frequenze sono simili.
*/

#include <iostream>
using namespace std;

int lancio_dado() {
  return (rand()%6)+1;
}

int main() {
  srand(time(NULL));

  /*** codice studenti da QUI ***/
  int frequenze[6] = {};
  int lancio;

  for(int i=0; i<1000000; i++) {
    lancio = lancio_dado();
    // NELLA CORREZIONE
    // soluzione con array più semplice e leggibile
    frequenze[lancio-1]++; //frequenze[lancio-1] += 1
  }

  /*** FINE codice studenti ***/

  // stampa più semplice e leggibile con gli array
  for(int i=0; i<6; i++) {
    cout << i << ``: '' << frequenze[i] << endl;
  }

}
```

A.3.3.2 Second consolidation exercise

```
/* Nel main vedi un esempio di programma che definisce e istanzia
un qualsiasi array 'numeri' di 'lun' numeri interi.

Il main chiama la funzione 'modifica_numeri'
(che devi implementare tu e di cui c'è già l'intestazione)
e la funzione deve modifica l'array in questo modo:

    per ogni numero dell'array:
```

```
    - se è maggiore di zero, raddoppia il numero
    - altrimenti, mette uno zero al suo posto

Dopo aver modificato l'array, la funzione stampa tutti
i numeri dell'array modificati.

Nell'esempio {4,7,-3,-4,5,6,0}, l'output deve essere:

    8 14 0 0 10 12 0
*/

#include <iostream>
using namespace std;

void modifica_numeri(int numeri[], int lun) {
    /* codice studenti da QUI */
    for(int i=0; i<lun; i++) {
        if(numeri[i]>0)
            numeri[i] *= 2;
        else
            numeri[i] = 0;
    }

    for(int i=0; i<lun; i++) {
        cout << numeri[i] << `` '';
    }
    cout << endl;
    /* FINE codice studenti */
}

int main() {
    int numeri[] = {4,7,-3,-4,5,6,0};
    int lun = 7;
    modifica_numeri(numeri, lun);
}
```

A.3.3.3 Third consolidation exercise

```
/*
Sviluppare una funzione 'conta_positivi' che prende in input un array di interi
 e la sua lunghezza.

La funzione conta quanti numeri dell'array sono maggiori o uguali a 0
```

```
e restituisce il conteggio dei positivi (un intero).

Nell'esempio {16,-2,-3,-4,11,5,0,-7}, l'output deve essere:

    Nell'array ci sono 4 num positivi
*/

#include <iostream>
using namespace std;

// LA TUA FUNZIONE QUI
/* codice studenti da QUI */
int conta_positivi(int numeri[], int lun) {
    int conta = 0;
    for(int i=0; i<lun; i++) {
        if(numeri[i]>=0) conta++;
    }
    return conta;
}
/* FINE codice studenti */

int main() {
    int numeri[] = {16,-2,-3,-4,11,5,0,-7};
    int lun = 8;
    cout<<``Nell'array ci sono ''<< conta_positivi(numeri,lun)<<`` num positivi'';
}
```

A.3.3.4 Fourth consolidation exercise

```
/*
Sviluppare una funzione 'accumula_array' che prende in input un array di interi
e la sua lunghezza.

La funzione sostituisce ad ogni valore dell'array la somma del valore stesso
con tutti i suoi precedenti. Il primo elemento resta invariato.

Dopo aver modificato l'array, la funzione ne stampa tutti i valori modificati.

Nell'esempio {2,5,3,4,0,1}, l'output deve essere:

    2 7 10 14 14 15

 2   il primo elemento resta invariato
 7 = 5 + 2
```

```
10 = 3 + 7
14 = 4 + 14
14 = 0 + 14
15 = 1 + 14
*/

#include <iostream>
using namespace std;

// LA TUA FUNZIONE QUI
/* codice studenti da QUI */
void accumula_array(int numeri[], int lun) {
    int somma = 0;
    int tmp = 0;
    for(int i=0; i<lun; i++) {
        tmp = numeri[i];
        numeri[i] += somma;
        somma += tmp;
    }

    for(int i=0; i<lun; i++) {
        cout<<numeri[i]<<‘‘ ’’;
    }
    cout<<endl;
}
/* FINE codice studenti */

int main() {
    int numeri[] = {2,5,3,4,0,1};
    int lun = 6;
    accumula_array(numeri, lun);
    return 0;
}
```

A.4 Student Questionnaires

A.4.1 Pre-experimentation Questionnaire (common)

The questionnaire, as described in 10.1.2.2 and 10.1.2.3, follows.

Informatica fino ad ora

Siamo due ricercatori dell'università di Bologna:
michael.lodi@unibo.it
marco.sbaraglia@unibo.it

Per circa una settimana osserveremo le vostre lezioni di informatica.
Quello che osserviamo ci serve a riflettere sulla didattica dell'informatica, non serve a valutarvi e nemmeno viene condiviso con i vostri proff.

Le vostre risposte verranno utilizzare in modo anonimo.
Per analizzarle ci serve il consenso firmato da voi, o dai vostri genitori se siete minorenni.
Vi forniremo l'informativa insieme al modulo da compilare per il consenso.

Quando vi chiederemo le vostre opinioni e percezioni - cercheremo di disturbarvi il meno possibile - siate il più possibile sinceri e diretti. Nessuno vi giudicherà per quello che dite, più sinceri siete, più ci aiutate nel nostro lavoro.

Se vi è sorto qualche dubbio su questa cosa, chiedeteci pure fin da adesso.
Se i vostri genitori dovessero avere dubbi, chiarite che si tratta di semplici domande sulla vostra esperienza di studenti, come quelle a cui state per rispondere.

*Campo obbligatorio

1. Email *

Per ciascuna delle coppie che trovi sotto, scegli il numero che meglio descrive come ti senti rispetto alla materia Informatica.

Riferito alla materia 'Informatica' di quest'anno, da settembre fino ad adesso.

2. *

Contrassegna solo un ovale.

	Motivato / Non motivato
1	
2	
3	
4	
5	
6	
7	

3. *

Contrassegna solo un ovale.

	Interessato / Non interessato
1	
2	
3	
4	
5	
6	
7	

4. *

Contrassegna solo un ovale.

	Coinvolto / Non coinvolto
1	
2	
3	
4	
5	
6	
7	

5. *

Contrassegna solo un ovale.

	Non stimolato							Stimolato
	1	2	3	4	5	6	7	

6. *

Contrassegna solo un ovale.

	Non voglio studiarla							Voglio studiarla
	1	2	3	4	5	6	7	

7. *

Contrassegna solo un ovale.

	Ispirato							Non ispirato
	1	2	3	4	5	6	7	

8. *

Contrassegna solo un ovale.

	Non messo alla prova							Messo alla prova
	1	2	3	4	5	6	7	

9. *

Contrassegna solo un ovale.

Non energizzato

1
2
3
4
5
6
7

Energizzato

10. *

Contrassegna solo un ovale.

Non entusiasta

1
2
3
4
5
6
7

Entusiasta

11. *

Contrassegna solo un ovale.

Elettrizzato

1
2
3
4
5
6
7

Non elettrizzato

12. *

Contrassegna solo un ovale.

Carico

1
2
3
4
5
6
7

Non carico

13. *

Contrassegna solo un ovale.

Non affascinato

1
2
3
4
5
6
7

affascinato

14. Quanto stai imparando in Informatica quest'anno? *

1 non ho imparato niente -- 10 ho imparato più di quanto abbia imparato in ogni altra materia

Contrassegna solo un ovale.

1
2
3
4
5
6
7
8
9
10

15. Fino ad ora, quanto pensi che avresti potuto imparare quest'anno in Informatica * se la didattica fosse stata ottimale (attività ed esercizi chiari e interessanti, ecc.)?

Contrassegna solo un ovale.

1
2
3
4
5
6
7
8
9
10

A.4.1.1 In-progress checkpoints (common)

The questionnaire, as described in 10.1.2.3, follows.

Checkpoint 1

Rispondi alle due domande in modo molto sintetico, anche con una sola parola

*Campo obbligatorio

1. Email *

2. Come ti senti dopo questa attività (esercizio estrazione del lotto)? *
   ...in pochissime parole, anche una sola può bastare!

3. Come hai trovato questa attività (esercizio estrazione del lotto)? *
   ...in pochissime parole, anche una sola può bastare!

A.4.2 Post-experimentation Questionnaires

The questionnaires, as described in 10.1.2.2 and 10.1.2.3, follow.

A.4.2.1 Post-experimentation Questionnaire for the experimental class

Informatica dopo l'intro agli array

NOTA BENE
Per ciascuna delle coppie che trovi sotto, scegli il numero che meglio descrive come ti senti rispetto alla materia Informatica dopo l'introduzione agli array.

*Campo obbligatorio

1. Email *

_____

2. *

Contrassegna solo un ovale.

Motivato

1 ◯
2 ◯
3 ◯
4 ◯
5 ◯
6 ◯
7 ◯

Non motivato

3. *

Contrassegna solo un ovale.

Interessato

1 ◯
2 ◯
3 ◯
4 ◯
5 ◯
6 ◯
7 ◯

Non interessato

4. *

Contrassegna solo un ovale.

Coinvolto

1 ◯
2 ◯
3 ◯
4 ◯
5 ◯
6 ◯
7 ◯

Non coinvolto

5. *

Contrassegna solo un ovale.

Non stimolato

1 ◯
2 ◯
3 ◯
4 ◯
5 ◯
6 ◯
7 ◯

Stimolato

6. *

Contrassegna solo un ovale.

Non voglio studiarla		
1		
2		
3		
4		
5		
6		
7		
Voglio studiarla		

7. *

Contrassegna solo un ovale.

Ispirato		
1		
2		
3		
4		
5		
6		
7		
Non ispirato		

8. *

Contrassegna solo un ovale.

Non messo alla prova		
1		
2		
3		
4		
5		
6		
7		
Messo alla prova		

9. *

Contrassegna solo un ovale.

Non energizzato		
1		
2		
3		
4		
5		
6		
7		
Energizzato		

10. *

Contrassegna solo un ovale.

Non entusiasta

1
2
3
4
5
6
7

Entusiasta

11. *

Contrassegna solo un ovale.

Elettrizzato

1
2
3
4
5
6
7

Non elettrizzato

12. *

Contrassegna solo un ovale.

Carico

1
2
3
4
5
6
7

Non carico

13. *

Contrassegna solo un ovale.

Non affascinato

1
2
3
4
5
6
7

affascinato

14. Quanto hai imparato in Informatica in quest'ultimo periodo di introduzione agli array? *

1 non ho imparato niente – 10 ho imparato più di quanto abbia imparato in ogni altra materia

Contrassegna solo un ovale.

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

15. Quanto pensi che avresti potuto imparare in Informatica, in quest'ultimo periodo di introduzione agli array, se la didattica fosse stata ottimale (attività ed esercizi chiari e interessanti, ecc.)? *

Contrassegna solo un ovale.

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

16. Quali aspetti hai apprezzato del modo in cui sono stati introdotti gli array (farti provare a risolvere il problema del lotto prima di spiegare gli array)? *

17. Quali aspetti invece non ti sono piaciuti? *

18. Come ti sei sentitə durante la fase in cui non sapevi risolvere il problema del lotto (perché ancora non conoscevi ancora array)? *

19. Ti ha aiutatə a imparare gli array oppure ti ha ostacolatə? Perché? *

A.4.2.2 Post-experimentation Questionnaire for the control class (shorter)

Informatica dopo l'intro agli array

NOTA BENE
Per ciascuna delle coppie che trovi sotto, scegli il numero che meglio descrive come ti senti rispetto alla materia Informatica dopo l'introduzione agli array.

*Campo obbligatorio

1. Email *

_____

2. *

Contrassegna solo un ovale.

Motivato

1 ◯
2 ◯
3 ◯
4 ◯
5 ◯
6 ◯
7 ◯

Non motivato

3. *

Contrassegna solo un ovale.

Interessato

1 ◯
2 ◯
3 ◯
4 ◯
5 ◯
6 ◯
7 ◯

Non interessato

4. *

Contrassegna solo un ovale.

Coinvolto

1 ◯
2 ◯
3 ◯
4 ◯
5 ◯
6 ◯
7 ◯

Non coinvolto

5. *

Contrassegna solo un ovale.

Non stimolato

1 ◯
2 ◯
3 ◯
4 ◯
5 ◯
6 ◯
7 ◯

Stimolato

6. *

Contrassegna solo un ovale.

Non voglio studiarla		
1		
2		
3		
4		
5		
6		
7		
Voglio studiarla		

7. *

Contrassegna solo un ovale.

Ispirato		
1		
2		
3		
4		
5		
6		
7		
Non ispirato		

8. *

Contrassegna solo un ovale.

Non messo alla prova		
1		
2		
3		
4		
5		
6		
7		
Messo alla prova		

9. *

Contrassegna solo un ovale.

Non energizzato		
1		
2		
3		
4		
5		
6		
7		
Energizzato		

10. *

Contrassegna solo un ovale.

Non entusiasta

1
2
3
4
5
6
7

Entusiasta

11. *

Contrassegna solo un ovale.

Elettrizzato

1
2
3
4
5
6
7

Non elettrizzato

12. *

Contrassegna solo un ovale.

Carico

1
2
3
4
5
6
7

Non carico

13. *

Contrassegna solo un ovale.

Non affascinato

1
2
3
4
5
6
7

affascinato

14. Quanto hai imparato in Informatica in quest'ultimo periodo di introduzione agli array? *

1 non ho imparato niente – 10 ho imparato più di quanto abbia imparato in ogni altra materia

Contrassegna solo un ovale.

15. Quanto pensi che avresti potuto imparare in Informatica, in quest'ultimo periodo di introduzione agli array, se la didattica fosse stata ottimale (attività ed esercizi chiari e interessanti, ecc.)? *

Contrassegna solo un ovale.

Appendix B

Material of the Didactical Situation on Interdisciplinary Cryptography

B.1 Researcher Observation Grid

The researcher observation grid used during the Didactical Situation, as described in 15.6, follows.

Observer *<Name>* **Start time** *<hh:mm>*

Group *<ID>* (*<milieu short description>*) **End time** *<hh:mm>*

Participants (prospective teachers)

- **P1** *<Name Surname> – <discipline>*
- **P2** *<Name Surname> – <discipline>*
- **P3** *<Name Surname> – <discipline>*
- **P4** *<Name Surname> – <discipline>*
- **P5** *<Name Surname> – <discipline>*

Group work						
	P1	**P2**	**P3**	**P3**	**P5**	*Notes*
Works mainly together with the group						
Works in subgroups *specify subgroups & different approaches*						
Works alone *specify approach(es)*						
Constantly communicates with others						
Does not (try to) communicate						
Not understood by other students						
Other / notes:						
Problem-solving strategies						
	P1	**P2**	**P3**	**P3**	**P5**	*Notes*
Heuristic algorithm (partitions of the graph)						
proposed						
followed						
abandoned						
Algorithm using lists (each list is a neighborhood of a PDS node)	*(alt. desc.: find a subset of lists of neighbors that include each node exactly once)*					
proposed						
followed						
abandoned						

	P1	P2	P3	P3	P5	*Notes*
Algorithm 1 using the adjacency matrix: find a linear combination that has 1 everywhere	*(alt. desc.: use the adjacency matrix to represent the graph and then to find a subset of rows whose sum is exactly a list of 1s)*					
proposed						
followed						
abandoned						
Algorithm 2 using the adjacency matrix: create a linear system of equations (to be solved by a linear solver)						
proposed						
followed						
abandoned						
Other strategy (specify):						
proposed						
followed						
abandoned						
Other / notes:						

Boundary objects

	P1	P2	P3	P3	P5	*Notes*
Talks about the adjacency matrix as a way to represent and solve a system of equations						
Talks about the adjacency matrix as a way to represent the graph (1 for neighbor)						
Other / notes:						

Linguistic aspects

	P1	P2	P3	P3	P5	*Notes*
Talks about the problem using a cryptography language, e.g., PDS as the (private) key						
Talks about the problem using own discipline language						
Talks about the problem using OTHER discipline languages						
Other / notes:						

Interdisciplinary aspects

	P1	P2	P3	P3	P5	*Notes*
Effort in being understood by students of other disciplines						
Remain in / go back to their “disciplinary confort zone”						
Other / notes:						

Disciplinary thinking / acting

	P1	P2	P3	P3	P5	*Notes*
"First make it work, then make it nice" (CS) approach						
Comfort in doing a lot of computations (CS) (e.g., a lot of equations without simplifying, brute force for sublists)						
Search for an elegant representation/strategy before trying to solve (Math/Phys?)						
Discomfort in doing a lot of computations (search for simplification/abstraction) (Math/Phys?)						
Other / notes:						

Bibliography

Joint Task Force on Computing Curricula ACM/IEEE-CS. 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. ACM Press and IEEE Computer Society Press, USA. https://doi.org/10.1145/2534860 [Cited on pages 18 and 22]

B. Adelson and E. Soloway. 1985. The Role of Domain Expenence in Software Design. *IEEE Transactions on Software Engineering* SE-11, 11 (1985), 1351–1360. https://doi.org/10.1109/TSE.1985.231883 [Cited on page 31]

Sanne F. Akkerman and Arthur Bakker. 2011. Boundary Crossing and Boundary Objects. *Review of Educational Research* 81, 2 (June 2011), 132–169. https://doi.org/10.3102/0034654311404435 [Cited on pages 95, 96, 262, and 267]

Giora Alexandron, Michal Armoni, Michal Gordon, and David Harel. 2012. The Effect of Previous Programming Experience on the Learning of Scenario-Based Programming. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '12)*. Association for Computing Machinery (ACM), New York, NY, USA, 151–159. https://doi.org/10.1145/2401796.2401821 [Cited on page 129]

V. H. Allan and M. V. Kolesar. 1997. Teaching Computer Science: A Problem Solving Approach That Works. *SIGCUE Outlook* 25, 1–2 (jan 1997), 2–10. https://doi.org/10.1145/274375.274376 [Cited on page 15]

Amjad Altadmri and Neil C.C. Brown. 2015. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '15)*. Association for Computing Machinery (ACM), New York, NY, USA, 522–527. https://doi.org/10.1145/2676723.2677258 [Cited on page 54]

David Alvargonzález. 2011. Multidisciplinarity, Interdisciplinarity, Transdisciplinarity, and the Sciences. *International Studies in the Philosophy of Science* 25, 4 (Dec. 2011), 387–403. https://doi.org/10.1080/02698595.2011.623366 [Cited on page 97]

Rachid Anane and Mohammad T. Alshammari. 2020. A Dynamic Visualisation of the DES Algorithm and a Multi-Faceted Evaluation of Its Educational Value. In *Proceedings of*

the 25th ACM Conference on Innovation & Technology in Computer Science Education (Trondheim, Norway) *(ITiCSE '20)*. Association for Computing Machinery (ACM), New York, NY, USA, 370–376. https://doi.org/10.1145/3341525.3387386 [Cited on pages 91 and 249]

Aivar Annamaa. 2015. Thonny,: A Python IDE for Learning Programming. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '15)*. Association for Computing Machinery (ACM), New York, NY, USA, 343. https://doi.org/10.1145/2729094.2754849 [Cited on page 184]

Leo Apostel, Guy Berger, Asa Brigs, and Guy Michaud. 1972. *Interdisciplinarity Problems of Teaching and Research in Universities*. Technical Report. Organisation for Economic Cooperation and Development, Paris (France). Centre for Educational Research and Innovation. 307 pages. [Cited on page 98]

Michèle Artigue. 1994. Didactical engineering as a framework for the conception of teaching products. *Didactics of mathematics as a scientific discipline* 13 (1994), 27–39. [Cited on page 99]

Michéle Artigue. 2014. Didactic Engineering in Mathematics Education. In *Encyclopedia of Mathematics Education*. Springer Netherlands, 159–162. https://doi.org/10.1007/978-94-007-4978-8_44 [Cited on pages 99 and 100]

Michèle Artigue. 2020. Didactic Engineering in Mathematics Education. In *Encyclopedia of Mathematics Education*, Stephen Lerman (Ed.). Springer International Publishing, Cham, Switzerland, 202–206. https://doi.org/10.1007/978-3-030-15789-0_44 [Cited on pages 99, 100, and 246]

J. W. Atwood and E. Regener. 1981. Teaching Subsets of Pascal. In *Proceedings of the Twelfth SIGCSE Technical Symposium on Computer Science Education* (St. Louis, Missouri, USA) *(SIGCSE '81)*. Association for Computing Machinery (ACM), New York, NY, USA, 96–103. https://doi.org/10.1145/800037.800969 [Cited on page 55]

Lindsay Baker, Stella Ng, and Farah Friesen. 2019. *Paradigms of Education. An Online Supplement.* Retrieved November 8, 2022 from https://www.paradigmsofeducation.com [Cited on pages 36, 38, 39, and 40]

Doug Baldwin. 1996. Discovery Learning in Computer Science. *SIGCSE Bull.* 28, 1 (mar 1996), 222–226. https://doi.org/10.1145/236462.236544 [Cited on pages 81 and 82]

Doug Baldwin and Johannes A. G. M. Koomen. 1992. Using Scientific Experiments in Early Computer Science Laboratories. In *Proceedings of the Twenty-Third SIGCSE Technical Symposium on Computer Science Education* (Kansas City, MO, USA) *(SIGCSE '92)*. Association for Computing Machinery (ACM), New York, NY, USA, 102–106. https://doi.org/10.1145/134510.134532 [Cited on page 82]

T. Balman. 1981. Computer assisted teaching of FORTRAN. *Computers & Education* 5, 2 (1981), 111–123. https://doi.org/10.1016/0360-1315(81)90020-8 [Cited on page 55]

Bianca Vienni Baptista and Julie Thompson Klein. 2022. *Institutionalizing Interdisciplinarity and Transdisciplinarity*. Routledge. https://doi.org/10.4324/9781003129424 [Cited on page 94]

E Barelli, B Barquero, O Romero, M R Aguada, J Giménez, C Pipitone, G Sala-Sebastià, A Nipyrakis, A Kokolaki, I Metaxas, E Michailidi, D Stavrou, I Bartzia, M Lodi, M Sbaraglia, S Modeste, S Martini, V Durand-Guerrier, V Bagaglini, S Satanassi, P Fantini, S Kapon, L Branchetti, and O Levrini. 2022. Disciplinary identities in interdisciplinary topics: challenges and opportunities for teacher education. In *Proceedings of ESERA 2021* (Braga, Portugal) *(European Science Education Research Association 2021 Biannual Conference)*, G S Carvalho, A S Afonso, and & Z Anastácio (Eds.), Vol. 13. 934–943. https://esera2021.org/en/content/eproceedings/conference-proceedings/conf-proceedings.html [Cited on pages 94 and 248]

Erik Barendsen and Carsten Schulte. 2018. Perspectives on Computing Curricula. In *Computer Science Education. Perspectives on Teaching and Learning in School*, S. Sentance, E. Barendsen, and C. Schulte (Eds.). Bloomsbury Academic, London, United Kingdom, Chapter 7, 77–90. https://doi.org/10.5040/9781350057142.ch-007 [Cited on page 22]

Berta Barquero and Marianna Bosch. 2015. Didactic Engineering as a Research Methodology: From Fundamental Situations to Study and Research Paths. In *Task Design In Mathematics Education*. Springer International Publishing, Cham, Switzerland, 249–272. https://doi.org/10.1007/978-3-319-09629-2_8 [Cited on page 99]

Howard S. Barrows. 1996. Problem-based learning in medicine and beyond: A brief overview. *New Directions for Teaching and Learning* 1996, 68 (1996), 3–12. https://doi.org/10.1002/tl.37219966804 [Cited on page 47]

Walter L. Bateman. 1990. *Open to Question: The Art of Teaching and Learning by Inquiry*. Jossey-Bass Inc., San Francisco, CA, USA. [Cited on page 81]

Aminah Bibi Bawamohiddin and Rozilawati Razali. 2017. Problem-based Learning for Programming Education. *International Journal on Advanced Science, Engineering and Information Technology* 7, 6 (Dec. 2017), 2035. https://doi.org/10.18517/ijaseit.7.6.2232 [Cited on pages 47 and 48]

Piraye Bayman and Richard E. Mayer. 1988. Using conceptual models to teach BASIC computer programming. *Journal of Educational Psychology* 80, 3 (Sept. 1988), 291–298. https://doi.org/10.1037/0022-0663.80.3.291 [Cited on page 25]

Michael J Beatty and Shelly K Payne. 1985. Is construct differentiation loquacity?: A motivational perspective. *Human Communication Research* 11 (1985), 605–612. [Cited on page 160]

Theresa Beaubouef and John Mason. 2005. Why the High Attrition Rate for Computer Science Students: Some Thoughts and Observations. *ACM SIGCSE Bulletin* 37, 2 (jun 2005), 103–106. https://doi.org/10.1145/1083431.1083474 [Cited on page 15]

Byron Weber Becker. 2001. Teaching CS1 with Karel the Robot in Java. *SIGCSE Bull.* 33, 1 (feb 2001), 50–54. https://doi.org/10.1145/366413.364536 [Cited on page 55]

Tim Bell, Jason Alexander, Isaac Freeman, and Mick Grimley. 2009. Computer Science Unplugged: School Students Doing Real Computing Without Computers. *The New Zealand Journal of Applied Computing and Information Technology* 13, 1 (2009), 20–29. [Cited on pages 83, 84, 85, and 230]

Tim Bell, Peter Andreae, and Anthony Robins. 2014. A Case Study of the Introduction of Computer Science in NZ Schools. 14, 2, Article 10 (jun 2014), 31 pages. https://doi.org/10.1145/2602485 [Cited on page 75]

Tim Bell and Caitlin Duncan. 2018. Teaching Computing in Primary Schools. In *Computer Science Education. Perspectives on Teaching and Learning in School*, S. Sentance, E. Barendsen, and C. Schulte (Eds.). Bloomsbury Academic, London, United Kingdom, Chapter 10, 131–150. https://doi.org/10.5040/9781350057142.ch-010 [Cited on pages 76, 77, 79, and 80]

Tim Bell, Harold Thimbleby, Mike Fellows, Ian Witten, Neil Koblitz, and Matthew Powell. 2003. Explaining cryptographic systems. *Computers & Education* 40, 3 (2003), 199–215. https://doi.org/10.1016/S0360-1315(02)00102-1 [Cited on pages 92, 246, 249, 252, and 253]

Tim Bell, Paul Tymann, and Amiram Yehudai. 2018. The Big Ideas in Computer Science for K-12 Curricula. *Bulletin of EATCS* 1, 124 (2018). [Cited on pages 71, 75, 76, 77, and 217]

Tim Bell and Jan Vahrenhold. 2018. CS Unplugged—How Is It Used, and Does It Work? In *Adventures Between Lower Bounds and Higher Altitudes: Essays Dedicated to Juraj Hromkovič on the Occasion of His 60th Birthday*, Hans-Joachim Böckenhauer, Dennis Komm, and Walter Unger (Eds.). Springer International Publishing, Cham, Switzerland, 497–521. https://doi.org/10.1007/978-3-319-98355-4_29 [Cited on pages 84 and 85]

Tim Bell, Ian Witten, and Michael Fellows. 2015. *Public Key Encryption*. CS Unplugged. Retrieved July 24, 2022 from https://classic.csunplugged.org/activities/public-key-encryption/ [Cited on pages 252 and 253]

Carlo Bellettini, Violetta Lonati, Dario Malchiodi, Mattia Monga, Anna Morpurgo, Mauro Torelli, and Luisa Zecca. 2014. Informatics Education in Italian Secondary Schools. *ACM Transactions on Computing Education* 14, 2, Article 15 (June 2014), 6 pages. https://doi.org/10.1145/2602490 [Cited on page 221]

Jens Bennedsen and Michael E. Caspersen. 2006. Abstraction Ability as an Indicator of Success for Learning Object-Oriented Programming? *ACM SIGCSE Bulletin* 38, 2 (jun 2006), 39–43. https://doi.org/10.1145/1138403.1138430 [Cited on page 29]

Jens Bennedsen and Michael E. Caspersen. 2007. Failure Rates in Introductory Programming. *ACM SIGCSE Bulletin* 39, 2 (jun 2007), 32–36. https://doi.org/10.1145/1272848.1272879 [Cited on page 15]

Jens Bennedsen and Michael E. Caspersen. 2019. Failure Rates in Introductory Programming: 12 Years Later. *ACM Inroads* 10, 2 (apr 2019), 30–36. https://doi.org/10.1145/3324888 [Cited on pages 15 and 21]

John B Biggs and Kevin F Collis. 1982. *Evaluating the Quality of Learning: the SOLO Taxonomy (Structure of the Observed Learning Outcome)*. Elsevier. https://doi.org/10.1016/c2013-0-10375-3 [Cited on page 25]

Gro Bjerknes and Tone Bratteteig. 1995. User Participation and Democracy: A Discussion of Scandinavian Research on System Development. *Scandinavian Journal of Information Systems* 7, 1 (Jan. 1995). https://aisel.aisnet.org/sjis/vol7/iss1/1 [Cited on page 58]

Elizabeth Bjork and Robert Bjork. 2011. Making things hard on yourself, but in a good way: Creating desirable difficulties to enhance learning. *Psychology and the Real World: Essays Illustrating Fundamental Contributions to Society* (01 2011), 56–64. [Cited on pages 49, 119, and 120]

Susanne Bødker, Pelle Ehn, Dan Sjögren, and Yngve Sundblad. 2000. Co-operative Design—perspectives on 20 years with 'the Scandinavian IT Design Model'. In *proceedings of NordiCHI*, Vol. 2000. 22–24. [Cited on pages 58 and 196]

Charles C. Bonwell and James A. Eison. 1991. Active Learning: Creating Excitement in the Classroom. 1991 ASHE-ERIC Higher Education Reports. [Cited on pages 34 and 35]

Richard Bornat, Saeed Dehnadi, and Simon. 2008. Mental Models, Consistency and Programming Aptitude. In *Proceedings of the Tenth Conference on Australasian Computing Education - Volume 78* (Wollongong, NSW, Australia) *(ACE '08)*. Australian Computer Society, Inc., AUS, 53–61. [Cited on page 16]

M. Bosch, J. Gascón, A. Mercier, and C. Magnolias. 2005. *La praxéologie comme unité d'analyse des processus didactiques* (1 ed.). 107–122. [Cited on page 100]

Nigel Bosch and Sidney K. D'Mello. 2013. Sequential Patterns of Affective States of Novice Programmers. *CEUR Workshop Proceedings* 1009 (01 2013), 1–10. [Cited on page 15]

Matt Bower. 2009. Learning Computing Online – Key Findings From Students. In *Proceedings of EdMedia + Innovate Learning 2009*, George Siemens and Catherine Fulford (Eds.). Association for the Advancement of Computing in Education (AACE), Honolulu, HI, USA, 4166–4175. https://www.learntechlib.org/p/32082 [Cited on pages 57, 58, and 191]

Matt Bower. 2011. Synchronous collaboration competencies in web-conferencing environments – their impact on the learning process. *Distance Education* 32, 1 (2011), 63–83. https://doi.org/10.1080/01587919.2011.565502 [Cited on page 58]

Matt Bower, Gregor Kennedy, Barney Dalgarno, Mark JW Lee, and Jacqueline Kenney. 2014. *Blended synchronous learning: A handbook for educators*. Australian Government, Office for Learning and Teaching, Department of Education, Sydney, NSW, Australia. [Cited on pages 57, 58, 189, and 191]

Garry D. Brewer. 1999. The Challenges of Interdisciplinarity. *Policy Sciences* 32, 4 (1999), 327–337. https://doi.org/10.1023/a:1004706019826 [Cited on pages 94 and 95]

Guy Brousseau and Nicolas Balacheff. 1997. *Theory of didactical situations in mathematics: didactique des mathématiques, 1970-1990*. Number v. 19 in Mathematics education library. Kluwer Academic Publishers, Dordrecht; Boston. [Cited on pages 98 and 99]

Guy Brousseau, Bernard Sarrazy, and Jarmila Novotná. 2020. Didactic Contract in Mathematics Education. In *Encyclopedia of Mathematics Education*, Stephen Lerman (Ed.). Springer International Publishing, Cham, Switzerland, 197–202. https://doi.org/10.1007/978-3-030-15789-0_46 [Cited on page 99]

Guy Brousseau and Virginia Warfield. 2020. Didactic Situations in Mathematics Education. In *Encyclopedia of Mathematics Education*, Stephen Lerman (Ed.). Springer International Publishing, Cham, Switzerland, 206–213. https://doi.org/10.1007/978-3-030-15789-0_47 [Cited on page 99]

Christopher Brown, Frederick Crabbe, Rita Doerr, Raymond Greenlaw, Chris Hoffmeister, Justin Monroe, Donald Needham, Andrew Phillips, Anthony Pollman, Stephen Schall, John Schultz, Steven Simon, David Stahl, and Sarah Standard. 2012. Anatomy, Dissection, and Mechanics of an Introductory Cyber-Security Course's Curriculum at the United States Naval Academy. In *Proceedings of the 17th ACM Conference on Innovation & Technology in Computer Science Education* (Haifa, Israel) *(ITiCSE '12)*. Association for Computing Machinery (ACM), New York, NY, USA, 303–308. https://doi.org/10.1145/2325296.2325367 [Cited on pages 90 and 249]

Neil C. C. Brown, Sue Sentance, Tom Crick, and Simon Humphreys. 2014. Restart: The Resurgence of Computer Science in UK Schools. *ACM Transactions on Computing Education* 14, 2 (2014), 9:1–9:22. [Cited on page 75]

Rachael L Brown. 2020. Why philosophers and scientists should work together. *The Biologist* (2020). [Cited on page 94]

Valerie A. Brown, John A. Harris, and Jacqueline Y. Russell. 2010. *Tackling wicked problems through the transdisciplinary imagination*. Earthscan, London, United Kingdom. OCLC: 669503940. [Cited on page 94]

Kim B. Bruce. 2004. Controversy on How to Teach CS 1: A Discussion on the SIGCSE-Members Mailing List. *ACM SIGCSE Bulletin* 36, 4 (jun 2004), 29–34. https://doi.org/10.1145/1041624.1041652 [Cited on pages 18 and 22]

Jerome S. Bruner. 1961. The Act of Discovery. *Harvard Educational Review* 31, 1 (1961). [Cited on pages 41 and 81]

Peter Brusilovsky, Eduardo Calabrese, Jozef Hvorecky, Anatoly Kouchnirenko, and Philip Miller. 1997. *Education and Information Technologies* 2, 1 (1997), 65–83. https://doi.org/10.1023/a:1018636507883 [Cited on page 55]

P. Brusilovsky and Others. 1994. *Teaching programming to novices: a review of approaches and tools*. Technical Report. https://eric.ed.gov/?id=ED388228 ERIC Number: ED388228. [Cited on page 55]

Suzanne Fox Buchele. 2013. Two Models of a Cryptography and Computer Security Class in a Liberal Arts Context. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery (ACM), New York, NY, USA, 543–548. https://doi.org/10.1145/2445196.2445360 [Cited on page 90]

Winslow Burleson and Rosalind W Picard. 2004. Affective agents: Sustaining motivation to learn through failure and a state of stuck. In *Workshop on Social and Emotional Intelligence in Learning Environments*. MIT Media Lab. [Cited on page 15]

Susanne Bødker, Christian Dindler, and Ole Sejer Iversen. 2017. Tying Knots: Participatory Infrastructuring at Work. *Computer Supported Cooperative Work (CSCW)* 26, 1-2 (April 2017), 245–273. https://doi.org/10.1007/s10606-017-9268-y [Cited on page 208]

Sara Capecchi, Michael Lodi, Violetta Lonati, and Sbaraglia Marco. 2022. *Materials of "Castle and Stairs to Learn Iteration: UMC Co-design"*. https://codesignumc.web.app/ [Cited on pages 198, 199, 201, and 202]

Sara Capecchi, Michael Lodi, Violetta Lonati, and Marco Sbaraglia. 2023. Castle and Stairs to Learn Iteration: Co-designing a UMC Learning Module with Teachers. (2023). Not yet published. [Cited on page 195]

Michael E. Caspersen. 2018. Teaching Programming. In *Computer Science Education. Perspectives on Teaching and Learning in School*, S. Sentance, E. Barendsen, and C. Schulte (Eds.). Bloomsbury Academic, London, United Kingdom, Chapter 9, 109–130. https://doi.org/10.5040/9781350057142.ch-009 [Cited on pages 13, 21, 23, 24, 45, 46, and 128]

Michael E. Caspersen and Michael Kölling. 2009. STREAM: A First Programming Process. *ACM Transactions on Computing Education* 9, 1, Article 4 (mar 2009), 29 pages. https://doi.org/10.1145/1513593.1513597 [Cited on page 23]

Francisco Enrique Vicente Castro and Kathi Fisler. 2017. Designing a multi-faceted SOLO taxonomy to track program design skills through an entire course. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*. Association for Computing Machinery (ACM), New York, NY, USA. https://doi.org/10.1145/3141880.3141891 [Cited on page 27]

Christina Chalmers, Merilyn Carter, Tom Cooper, and Rod Nason. 2017. Implementing "Big Ideas" to Advance the Teaching and Learning of Science, Technology, Engineering, and Mathematics (STEM). *International Journal of Science and Mathematics Education* 15, S1 (Feb. 2017), 25–43. https://doi.org/10.1007/s10763-017-9799-1 [Cited on page 71]

Yves Chevallard and Marianna Bosch. 2020. Anthropological Theory of the Didactic (ATD). In *Encyclopedia of Mathematics Education*, Stephen Lerman (Ed.). Springer International Publishing, Cham, Switzerland, 53–61. https://doi.org/10.1007/978-3-030-15789-0_100034 [Cited on page 101]

Nancy L Chick, Aeron Haynie, Regan AR Gurung, and AR Regan. 2009. From generic to signature pedagogies: teaching disciplinary understandings. In *Exploring signature pedagogies: Approaches to disciplinary habits of min*. Stylus Publishing, 1–16. https://books.google.co.il/books?id=0SWec-nwL4EC [Cited on page 67]

Bernard C. K. Choi and Anita W. P. Pak. 2007. Multidisciplinarity, interdisciplinarity, and transdisciplinarity in health research, services, education and policy: 2. Promotors, barriers, and strategies of enhancement. *Clinical &amp Investigative Medicine* 30, 6 (Dec. 2007), 224. https://doi.org/10.25011/cim.v30i6.2950 [Cited on page 95]

David M Christophel. 1990. The relationships among teacher immediacy behaviors, student motivation, and learning. *Communication Education* 39, 4 (1990), 323–340. [Cited on page 160]

Code.org. 2022. *Artist pre-reader*. Retrieved January 4, 2023 from https://studio.code.org/projects/artist_k1/new [Cited on pages 197, 198, 199, and 201]

Code.org. n.d. *Code.org*. https://code.org/ [Cited on pages 196, 198, and 207]

Code.org, CSTA, & ECEP Alliance. 2020. *2020 State of Computer Science Education: Illuminating Disparities*. Technical Report. https://advocacy.code.org/2020_state_of_cs.pdf [Cited on page 15]

Code.org, CSTA, & ECEP Alliance. 2022. *2022 State of Computer Science Education: Understanding Our National Imperative*. Technical Report. https://advocacy.code.org/2022_state_of_cs.pdf [Cited on page 15]

Merijke Coenraad, Jen Palmer, Donna Eatinger, David Weintrop, and Diana Franklin. 2022. Using participatory design to integrate stakeholder voices in the creation of a culturally relevant computing curriculum. *International Journal of Child-Computer Interaction* 31 (2022), 100353. https://doi.org/10.1016/j.ijcci.2021.100353 [Cited on pages 59 and 208]

John W. Coffey. 2010. Web Conferencing Software in University-Level, E-Learning-Based, Technical Courses. *Journal of Educational Technology Systems* 38, 3 (2010), 367–381. https://doi.org/10.2190/ET.38.3.f [Cited on page 58]

Timothy Colburn and Gary Shute. 2007. Abstraction in Computer Science. *Minds and Machines* 17, 2 (2007), 169–184. https://doi.org/10.1007/s11023-007-9061-7 [Cited on page 31]

Thomas H. Cormen, Charles Eric Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to algorithms* (fourth ed.). The MIT Press, Cambridge, Massachusetts, USA. [Cited on pages 251, 258, and 259]

Malcolm Corney, Donna Teague, and Richard N. Thomas. 2010. Engaging Students in Programming. In *Proceedings of the Twelfth Australasian Conference on Computing Education - Volume 103* (Brisbane, Australia) *(ACE '10)*. Australian Computer Society, Inc., AUS, 63–72. [Cited on pages 15 and 16]

Isabella Corradini, Michael Lodi, and Enrico Nardelli. 2017a. Computational Thinking in Italian Schools: Quantitative Data and Teachers' Sentiment Analysis after Two Years of "Programma Il Futuro". In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) *(ITiCSE '17)*. Association for Computing Machinery (ACM), New York, NY, USA, 224–229. https://doi.org/10.1145/3059009.3059040 [Cited on pages 196 and 198]

Isabella Corradini, Michael Lodi, and Enrico Nardelli. 2017b. Conceptions and Misconceptions about Computational Thinking among Italian Primary School Teachers. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. Association for Computing Machinery (ACM), New York, NY, USA, 136–144. https://doi.org/10.1145/3105726.3106194 [Cited on page 66]

Isabella Corradini, Michael Lodi, and Enrico Nardelli. 2018. An Investigation of Italian Primary School Teachers' View on Coding and Programming. In *Informatics in Schools. Fundamentals of Computer Science and Software Engineering. Lecture Notes in Computer Science (ISSEP 2018)*, Sergei N. Pozdniakov and Valentina Dagienė (Eds.), Vol. 11169. Springer International Publishing, Cham, Switzerland, 228–243. [Cited on page 66]

Tom Crick, Cathryn Knight, Richard Watermeyer, and Janet Goodall. 2020. The Impact of COVID-19 and "Emergency Remote Teaching" on the UK Computer Science Education Community. In *United Kingdom & Ireland Computing Education Research Conference. (UKICER '20)*. ACM, New York, NY, USA, 31–37. https://doi.org/10.1145/3416465.3416472 [Cited on page 58]

CS Unplugged. [n.d.]. Principles. Retrieved December 29, 2022 from https://csunplugged.org/en/principles/ [Cited on pages 84 and 240]

CSTA. 2017. *CSTA K-12 Computer Science Standards, rev. 2017*. Technical Report. Computer Science Teachers Association. http://www.csteachers.org/standards [Cited on pages 90, 198, 199, and 248]

Computer Science Teachers Association CSTA. 2016. CSTA K–12 Computer Science Standards [INTERIM]. https://kipdf.com/interim-csta-k-12-computer-science-standards_5b1132977f8b9a2d218b45db.html [Cited on page 22]

Paul Curzon. 2015. Unplugged Computational Thinking for Fun. *KEYCIT 2014 - Key Competencies in Informatics and ICT* 7 (2015), 15–27. [Cited on page 83]

Paul Curzon, Tim Bell, Jane Waite, and Mark Dorling. 2019. Computational Thinking. In *The Cambridge Handbook of Computing Education Research*, Sally A. Fincher and Anthony V.

Robins (Eds.). Cambridge University Press, Cambridge, United Kingdom, Chapter 17, 513–546. https://doi.org/10.1017/9781108654555.018 [Cited on pages 66, 83, 111, and 117]

Paul Curzon and Peter William McOwan. 2017. *The Power of Computational Thinking: Games, Magic and Puzzles to Help You Become a Computational Thinker*. World Scientific Publishing Europe Ltd, London, United Kingdom. [Cited on page 83]

Paul Curzon, Peter W McOwan, James Donohue, Seymour Wright, and William Marsh. 2018. Teaching of Concepts. In *Computer Science Education. Perspectives on Teaching and Learning in School*, S. Sentance, E. Barendsen, and C. Schulte (Eds.). Bloomsbury Academic, London, United Kingdom, Chapter 8, 91–108. https://doi.org/10.5040/9781350057142.ch-008 [Cited on pages 65, 79, 81, and 83]

Paul Curzon, Peter W. McOwan, Nicola Plant, and Laura R. Meagher. 2014. Introducing Teachers to Computational Thinking Using Unplugged Storytelling. In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education (WiPSCE '14)*. Association for Computing Machinery (ACM), New York, NY, USA, 89–92. https://doi.org/10.1145/2670757.2670767 [Cited on page 84]

Dean C. Dauw. 1967. Vocational interests of highly creative computer personnel. *Personnel Journal* 46, 10 (1967), 653–659. [Cited on page 16]

Neil Davidson. 1990. *Cooperative Learning in Mathematics: A Handbook for Teachers*. Addison-Wesley Innovative Division, Menlo Park, CA, USA. [Cited on page 82]

Michael de Raadt, Richard Watson, and Mark Toleman. 2005. *Textbooks: under inspection*. Technical Report. University of Southern Queensland, Toowoomba, Australia. https://eprints.usq.edu.au/167/ [Cited on page 23]

FadiP. Deek, Howard Kimmel, and James A. McHugh. 1998. Pedagogical Changes in the Delivery of the First-Course in Computer Science: Problem Solving, Then Programming. *Journal of Engineering Education* 87, 3 (July 1998), 313–320. https://doi.org/10.1002/j.2168-9830.1998.tb00359.x [Cited on page 48]

Saeed Dehnadi and Richard Bornat. 2006. The camel has two humps (working title). (2006). http://www.eis.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf [Cited on page 16]

Jill Denner and Shannon Campe. 2018. Equity and Inclusion in Computer Science Education. In *Computer Science Education. Perspectives on Teaching and Learning in School*, S. Sentance, E. Barendsen, and C. Schulte (Eds.). Bloomsbury Academic, London, United Kingdom, Chapter 14, 189–206. https://doi.org/10.5040/9781350057142.ch-014 [Cited on page 14]

Peter J. Denning. 2013. The Science in Computer Science. *Commun. ACM* 56, 5 (may 2013), 35–38. https://doi.org/10.1145/2447976.2447988 [Cited on page 15]

Peter J. Denning and Craig H. Martell. 2015. *Great Principles of Computing*. The MIT Press, Cambridge, Massachusetts, USA. [Cited on page 75]

Peter J. Denning, Matti Tedre, and Pat Yongpradit. 2017. Misconceptions about Computer Science. *Commun. ACM* 60, 3 (Feb. 2017), 31–33. https://doi.org/10.1145/3041047 [Cited on page 67]

Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. Understanding the Syntax Barrier for Novices. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education* (Darmstadt, Germany) *(ITiCSE '11)*. Association for Computing Machinery (ACM), New York, NY, USA, 208–212. https://doi.org/10.1145/1999747.1999807 [Cited on page 54]

Norman K Denzin and Yvonna S Lincoln. 2017. *The SAGE Handbook of Qualitative Research*. SAGE Publications, Thousand Oaks, CA, USA. [Cited on page 193]

Pranita Deshpande, Cynthia B. Lee, and Irfan Ahmed. 2019. Evaluation of Peer Instruction for Cybersecurity Education. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery (ACM), New York, NY, USA, 720–725. https://doi.org/10.1145/3287324.3287403 [Cited on pages 90 and 249]

John Dewey. 1916. *Democracy and Education : An Introduction to the Philosophy of Education*. Macmillan. [Cited on page 34]

A. A diSessa and H. Abelson. 1986. Boxer: A Reconstructible Computational Medium. *Commun. ACM* 29, 9 (sep 1986), 859–868. https://doi.org/10.1145/6592.6595 [Cited on page 29]

Allen Downey. 2015. *Think Python*. O'Reilly Media, Sebastopol, CA. [Cited on page 131]

Allison Druin. 2002. The role of children in the design of new technology. *Behaviour & Information Technology* 21, 1 (Jan. 2002), 1–25. https://doi.org/10.1080/01449290110108659 [Cited on page 58]

Benedict Du Boulay. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73. https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9 [Cited on pages 28, 109, and 110]

Caitlin Duncan. 2019. *Computer science and computational thinking in primary schools*. Ph.D. Dissertation. University of Canterbury. http://hdl.handle.net/10092/17160 [Cited on page 67]

Raymond Duval. 1995. *Sémiosis et pensée humaine: registres sémiotiques et apprentissages intellectuels*. Peter Lang, Berne, Switzerland. [Cited on pages 253, 255, and 261]

Raymond Duval. 2017. *Understanding the Mathematical Way of Thinking – The Registers of Semiotic Representations*. Springer International Publishing, Cham, Switzerland. https://doi.org/10.1007/978-3-319-56910-9 [Cited on pages 253, 255, and 261]

Sidney D'Mello and Art Graesser. 2012. Dynamics of affective states during complex learning. *Learning and Instruction* 22, 2 (2012), 145–157. https://doi.org/10.1016/j.learninstruc.2011.10.001 [Cited on page 15]

ECforALL. n.d. *Act 1 Curriculum*. https://impactconectar.wixsite.com/website/act-1 [Cited on page 198]

Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, Kate Sanders, and Carol Zander. 2006. Putting Threshold Concepts into Context in Computer Science Education. *ACM SIGCSE Bulletin* 38, 3 (jun 2006), 103–107. https://doi.org/10.1145/1140123.1140154 [Cited on page 15]

Joseph Elarde. 2016. Toward Improving Introductory Programming Student Course Success Rates: Experiences with a Modified Cohort Model to Student Success Sessions. *J. Comput. Sci. Coll.* 32, 2 (dec 2016), 113–119. [Cited on page 16]

Richard J. Enbody and William F. Punch. 2010. Performance of Python CS1 Students in Mid-Level Non-Python CS Courses. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (Milwaukee, Wisconsin, USA) *(SIGCSE '10)*. Association for Computing Machinery, New York, NY, USA, 520–523. https://doi.org/10.1145/1734263.1734437 [Cited on page 54]

Richard J. Enbody, William F. Punch, and Mark McCullen. 2009. Python CS1 as Preparation for C++ CS2. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education* (Chattanooga, TN, USA) *(SIGCSE '09)*. Association for Computing Machinery, New York, NY, USA, 116–120. https://doi.org/10.1145/1508865.1508907 [Cited on page 54]

Yrjö Engeström, Ritva Engeström, and Merja Kärkkäinen. 1995. Polycontextuality and boundary crossing in expert cognition: Learning and problem solving in complex work activities. *Learning and Instruction* 5, 4 (Jan. 1995), 319–336. https://doi.org/10.1016/0959-4752(95)00021-6 [Cited on pages 95 and 96]

Nathan L. Ensmenger. 2010. *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise*. The MIT Press, Cambridge, Massachusetts, USA. https://doi.org/10.7551/mitpress/9780262050937.001.0001 [Cited on pages 14 and 16]

Henry A. Etlinger. 1990. A Retrospective on an Early Software Projects Course. *SIGCSE Bull.* 22, 1 (feb 1990), 72–77. https://doi.org/10.1145/319059.319087 [Cited on page 82]

Joint Research Centre European Commission. 2017. *DigComp 2.1: the digital competence framework for citizens with eight proficiency levels and examples of use.* Publications Office. https://doi.org/10.2760/38842 [Cited on pages 89 and 216]

Katrina Falkner and Judy Sheard. 2019. Pedagogic Approaches. In *The Cambridge Handbook of Computing Education Research*, Sally A. Fincher and Anthony V. Robins (Eds.). Cambridge University Press, Cambridge, United Kingdom, Chapter 15, 445–480. https://doi.org/10.1017/9781108654555.016 [Cited on pages 24, 28, 36, 37, 38, and 50]

Katrina Falkner, Rebecca Vivian, and Nickolas Falkner. 2014. The Australian Digital Technologies Curriculum: Challenge and Opportunity. In *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148* (Auckland, New Zealand) *(ACE '14)*. Australian Computer Society, Inc., AUS, 3–12. [Cited on page 75]

Michel Fayol and Patrick Lemaire. 1989. Une étude expérimentale du fonctionnement distinctif de la virgule dans des phrases: perspective génétique. *Études de Linguistique Appliquée; Paris* 73 (1989), 71–80. https://search.proquest.com/docview/1307660874/citation/AD75E7B1D3194174PQ/1 [Cited on page 54]

Yvon Feaster, Luke Segars, Sally K. Wahba, and Jason O. Hallstrom. 2011. Teaching CS Unplugged in the High School (with Limited Success). In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education* (Darmstadt, Germany) *(ITiCSE '11)*. Association for Computing Machinery (ACM), New York, NY, USA, 248–252. https://doi.org/10.1145/1999747.1999817 [Cited on page 85]

Rachel E Fees, Jennifer A Da Rosa, Sarah S Durkin, Mark M Murray, and Angela L Moran. 2018. Unplugged cybersecurity: An approach for bringing computer science into the classroom. *International Journal of Computer Science Education in Schools* 2, 1 (Feb. 2018), 3–13. https://doi.org/10.21585/ijcses.v2i1.21 [Cited on pages 92 and 249]

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2004. The TeachScheme! Project: Computing and Programming for Every Student. *Computer Science Education* 14, 1 (2004), 55–77. https://doi.org/10.1076/csed.14.1.55.23499 arXiv:https://doi.org/10.1076/csed.14.1.55.23499 [Cited on page 55]

Michael R Fellows and Mark N Hoover. 1991. Perfect domination. *Australas. J Comb.* 3 (1991), 141–150. [Cited on page 250]

Michael R Fellows and Neal Koblitz. 1994. Combinatorially based cryptography for children (and adults). *Congressus Numerantium* 99 (1994), 9–41. [Cited on pages 246, 250, 252, and 253]

Sally Fincher, Johan Jeuring, Craig S. Miller, Peter Donaldson, Benedict du Boulay, Matthias Hauswirth, Arto Hellas, Felienne Hermans, Colleen Lewis, Andreas Mühling, Janice L. Pearce, and Andrew Petersen. 2020. Notional Machines in Computing Education: The Education of Attention. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE-WGR '20)*. Association for Computing Machinery (ACM), New York, NY, USA, 21–50. https://doi.org/10.1145/3437800.3439202 [Cited on pages 29, 110, and 111]

Sally A. Fincher and Anthony V. Robins. 2019. *The Cambridge Handbook of Computing Education Research*. Cambridge University Press, Cambridge, United Kingdom. https://doi.org/10.1017/9781108654555 [Cited on page 41]

Diana Franklin, Merijke Coenraad, Jennifer Palmer, Donna Eatinger, Anna Zipp, Marco Anaya, Max White, Hoang Pham, Ozan Gökdemir, and David Weintrop. 2020a. An Analysis of Use-Modify-Create Pedagogical Approach's Success in Balancing Structure and Student

Agency. In *Proceedings of the 2020 ACM Conference on International Computing Education Research* (Virtual Event, New Zealand) *(ICER '20)*. Association for Computing Machinery (ACM), New York, NY, USA, 14–24. https://doi.org/10.1145/3372782.3406256 [Cited on page 53]

Diana Franklin, Jean Salac, Zachary Crenshaw, Saranya Turimella, Zipporah Klain, Marco Anaya, and Cathy Thomas. 2020b. Exploring Student Behavior Using the TIPP&SEE Learning Strategy. In *Proceedings of the 2020 ACM Conference on International Computing Education Research* (Virtual Event, New Zealand) *(ICER '20)*. Association for Computing Machinery (ACM), New York, NY, USA, 91–101. https://doi.org/10.1145/3372782.3406257 [Cited on pages 53 and 54]

Scott Freeman, Sarah L. Eddy, Miles McDonough, Michelle K. Smith, Nnadozie Okoroafor, Hannah Jordt, and Mary Pat Wenderoth. 2014. Active learning increases student performance in science, engineering, and mathematics. *Proceedings of the National Academy of Sciences* 111, 23 (2014), 8410–8415. https://doi.org/10.1073/pnas.1319030111 arXiv:https://www.pnas.org/doi/pdf/10.1073/pnas.1319030111 [Cited on pages 29, 34, 45, and 115]

Robert Frodeman. 2014. *Sustainable Knowledge*. Palgrave Macmillan UK. https://doi.org/10.1057/9781137303028 [Cited on page 94]

Maurizio Gabbrielli and Simone Martini. 2010. *Programming Languages: Principles and Paradigms*. Springer London, London, United Kingdom. https://doi.org/10.1007/978-1-84882-914-5 [Cited on pages 29, 31, 32, and 33]

Reinaldo AZ Garcia. 1987. *Identifying the academic factors that predict the success of entering freshmen in a beginning computer science course*. Doctoral thesis. http://hdl.handle.net/2346/59483 [Cited on page 15]

D.R. Garrison, Martha Cleveland-Innes, and Tak Shing Fung. 2010. Exploring causal relationships among teaching, cognitive and social presence: Student perceptions of the community of inquiry framework. *The Internet and Higher Education* 13, 1 (2010), 31–36. https://doi.org/10.1016/j.iheduc.2009.10.002 Special Issue on the Community of Inquiry Framework: Ten Years Later. [Cited on page 189]

D. Randy Garrison, Terry Anderson, and Walter Archer. 1999. Critical Inquiry in a Text-Based Environment: Computer Conferencing in Higher Education. *The Internet and Higher Education* 2, 2-3 (March 1999), 87–105. https://doi.org/10.1016/S1096-7516(00)00016-6 [Cited on page 188]

Dedre Gentner. 1983. *Mental models*. Erlbaum, Hillsdale, N.J. [Cited on page 27]

Dedre Gentner. 2002. *Mental models, psychology of*. Elsevier Science, 9683–9687. [Cited on page 27]

W. Wayt Gibbs. 1994. Software's Chronic Crisis. *Scientific American* 271, 3 (Sept. 1994), 86–95. https://doi.org/10.1038/scientificamerican0994-86 [Cited on page 14]

Marleen Gilsing and Felienne Hermans. 2021. Gradual Programming in Hedy: A First User Study. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–9. https://doi.org/10.1109/vl/hcc51201.2021.9576236 [Cited on page 87]

David Ginat and Eti Menashe. 2015. SOLO Taxonomy for Assessing Novices' Algorithmic Design. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '15)*. Association for Computing Machinery (ACM), New York, NY, USA, 452–457. https://doi.org/10.1145/2676723.2677311 [Cited on page 26]

B.G. Glaser and A.L. Strauss. 1967. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Publishing Company. [Cited on page 187]

Inga Glogger-Frey, Corinna Fleischer, Lisa Grüny, Julian Kappich, and Alexander Renkl. 2015. Inventing a solution and studying a worked solution prepare differently for learning from direct instruction. *Learning and Instruction* 39 (Oct. 2015), 72–87. https://doi.org/10.1016/j.learninstruc.2015.05.001 [Cited on page 51]

Google LLC. & Gallup Inc. 2016. *Diversity gaps in computer science: exploring the underrepresentation of girls, Blacks and Hispanics*. Technical Report. http://goo.gl/PG34aH [Cited on page 15]

Andreas Gramm, Malte Hornung, and Helmut Witten. 2012. Email for You (Only?): Design and Implementation of a Context-Based Learning Process on Internetworking and Cryptography. In *Proceedings of the 7th Workshop in Primary and Secondary Computing Education* (Hamburg, Germany) *(WiPSCE '12)*. Association for Computing Machinery (ACM), New York, NY, USA, 116–124. https://doi.org/10.1145/2481449.2481477 [Cited on page 91]

Kathryn E. Gray and Matthew Flatt. 2003. ProfessorJ: A Gradual Introduction to Java through Language Levels. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Anaheim, CA, USA) *(OOPSLA '03)*. Association for Computing Machinery (ACM), New York, NY, USA, 170–177. https://doi.org/10.1145/949344.949394 [Cited on page 55]

Raymond Greenlaw, Christopher Brown, Zachary Dannelly, Andrew Phillips, and Sarah Standard. 2015. Using a Message Board as a Teaching Tool in an Introductory Cyber-Security Course. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '15)*. Association for Computing Machinery (ACM), New York, NY, USA, 308–313. https://doi.org/10.1145/2676723.2677221 [Cited on page 92]

David Gries. 1974. What Should We Teach in an Introductory Programming Course? *ACM SIGCSE Bulletin* 6, 1 (jan 1974), 81–89. https://doi.org/10.1145/953057.810447 [Cited on page 23]

Shuchi Grover and Roy Pea. 2013. Computational Thinking in K–12: A Review of the State of the Field. *Educational Researcher* 42, 1 (2013), 38–43. https://doi.org/10.3102/0013189X12463051 [Cited on page 15]

Till Grüne-Yanoff. 2016. Interdisciplinary success without integration. *European Journal for Philosophy of Science* 6, 3 (March 2016), 343–360. https://doi.org/10.1007/s13194-016-0139-z [Cited on page 94]

Raymonde Guindon, Herb Krasner, and Bill Curtis. 1987. Breakdowns and processes during the early activities of software design by professionals. (12 1987), 65–82. [Cited on page 30]

Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. Association for Computing Machinery (ACM), New York, NY, USA, 579–584. https://doi.org/10.1145/2445196.2445368 [Cited on page 184]

Ayush Gupta, David Hammer, and Edward F. Redish. 2010. The Case for Dynamic Models of Learners' Ontologies in Physics. *Journal of the Learning Sciences* 19, 3 (2010), 285–321. https://doi.org/10.1080/10508406.2010.491751 arXiv:https://doi.org/10.1080/10508406.2010.491751 [Cited on page 29]

Regan A. R. Gurung, Nancy L. Chick, and Aeron Haynie. 2009. *Exploring signature pedagogies: approaches to teaching disciplinary habits of mind* (1st ed ed.). Stylus Publishing, Sterling, Va. OCLC: 309904368. [Cited on page 67]

John Guttag. 2021. *Introduction to Computation and Programming Using Python : With Application to Computational Modeling and Understanding Data*. The MIT Press, Cambridge, Massachusetts, USA. [Cited on page 131]

Mark Guzdial. 2007. What makes programming so hard. http://home.cc.gatech.edu/csl/uploads/6/Guzdial-blog-pieces-on-what-is-CSEd.pdf [Cited on page 16]

Mark Guzdial. 2008. Paving the Way for Computational Thinking. *Commun. ACM* 51, 8 (Aug. 2008), 25–27. https://doi.org/10.1145/1378704.1378713 [Cited on page 128]

Mark Guzdial. 2010. Why is it so hard to learn to program. *Making Software: What Really Works, and Why We Believe It. O'Reilly Media* (2010), 111–124. [Cited on pages 15 and 16]

Mark Guzdial. 2013. Exploring Hypotheses about Media Computation. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research* (San Diego, San California, USA) *(ICER '13)*. Association for Computing Machinery (ACM), New York, NY, USA, 19–26. https://doi.org/10.1145/2493394.2493397 [Cited on page 86]

Mark Guzdial. 2015. Learner-Centered Design of Computing Education: Research on Computing for Everyone. *Synthesis Lectures on Human-Centered Informatics* 8, 6 (Nov. 2015), 1–165. https://doi.org/10.2200/s00684ed1v01y201511hci033 [Cited on page 15]

Mark Guzdial. 2017. Balancing Teaching CS Efficiently with Motivating Students. *Commun. ACM* 60, 6 (May 2017), 10–11. https://doi.org/10.1145/3077227 [Cited on pages 45 and 116]

Mark Guzdial. 2019a. Computing Education as a Foundation for 21st Century Literacy. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery (ACM), New York, NY, USA, 502–503. https://doi.org/10.1145/3287324.3290953 [Cited on page 86]

Mark Guzdial. 2019b. What's generally good for you vs what meets a need: Balancing explicit instruction vs problem/project-based learning in computer science classes. Retrieved January 26, 2023 from https://computinged.wordpress.com/2019/09/16/whats-good-for-you-vs-what-fixes-you-balancing-explicit-instruction-vs-problemproject-based-learning-in-computer-science-classes/ [Cited on pages 43 and 44]

Mark Guzdial. 2020. *How I'm lecturing during emergency remote teaching*. Retrieved January 30, 2023 from https://computinged.wordpress.com/2020/04/06/how-im-lecturing-during-emergency-remote-teaching/ [Cited on page 58]

Mark Guzdial. 2021. *Helping social studies teachers to teach data literacy with Teaspoon languages*. Retrieved December 23, 2022 from https://computinged.wordpress.com/2021/12/22/helping-social-studies-teachers-to-teach-data-literacy-with-teaspoon-languages/ [Cited on pages 56, 86, 88, 229, and 230]

Mark Guzdial, Shriram Krishnamurthi, Juha Sorva, Jan Vahrenhold, Anthony Bagnall, Richard L Cole, Themis Palpanas, Konstantinos Zoumpatianos, Christoph Becker, et al. 2020. Dagstuhl Reports, Volume 9, Issue 7, July 2019, Complete Issue. (2020). https://doi.org/10.4230/DAGREP.9.7 [Cited on page 110]

M. Guzdial, W.M. McCracken, and A. Elliott. 1997. Task specific programming languages as a first programming language. In *Proceedings Frontiers in Education 1997 27th Annual Conference. Teaching and Learning in an Era of Change*. IEEE. https://doi.org/10.1109/fie.1997.632675 [Cited on page 86]

Mark Guzdial and Bahare Naimipour. 2019. Task-Specific Programming Languages for Promoting Computing Integration: A Precalculus Example. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '19)*. Association for Computing Machinery (ACM), New York, NY, USA, Article 21, 5 pages. https://doi.org/10.1145/3364510.3364532 [Cited on pages 85, 86, 87, and 229]

Mark Guzdial and Elliot Soloway. 2002. Teaching the Nintendo Generation to Program. *Commun. ACM* 45, 4 (apr 2002), 17–21. https://doi.org/10.1145/505248.505261 [Cited on page 15]

Wynne Harlen et al. 2010. *Principles and Big Ideas of Science Education*. Association for Science Education, Hatfield, Hertfordshire, United Kingdom. [Cited on page 71]

Wynne Harlen et al. 2015. *Working with Big Ideas of Science Education*. Science Education Programme (SEP) of IAP, Trieste, Italy. [Cited on pages 71, 72, 73, 74, 75, and 217]

Teresa W Haynes, Stephen Hedetniemi, and Peter Slater. 2013. *Fundamentals of domination in graphs*. CRC press, Boca Raton, Florida, USA. [Cited on page 250]

Orit Hazzan, Yael Dubinsky, Larisa Eidelman, Victoria Sakhnini, and Mariana Teif. 2006. Qualitative Research in Computer Science Education. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '06)*. Association for Computing Machinery (ACM), New York, NY, USA, 408–412. https://doi.org/10.1145/1121341.1121469 [Cited on page 185]

Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. 2003. Helium, for Learning Haskell. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell* (Uppsala, Sweden) *(Haskell '03)*. Association for Computing Machinery (ACM), New York, NY, USA, 62–71. https://doi.org/10.1145/871895.871902 [Cited on page 55]

Felienne Hermans. 2020. Hedy: A Gradual Language for Programming Education. In *Proceedings of the 2020 ACM Conference on International Computing Education Research* (Virtual Event, New Zealand) *(ICER '20)*. Association for Computing Machinery (ACM), New York, NY, USA, 259–270. https://doi.org/10.1145/3372782.3406262 [Cited on pages 54, 55, 87, 110, and 111]

Felienne Hermans and Marlies Aldewereld. 2017. Programming is Writing is Programming. In *Companion Proceedings of the 1st International Conference on the Art, Science, and Engineering of Programming* (Brussels, Belgium) *(Programming '17)*. Association for Computing Machinery (ACM), New York, NY, USA, Article 33, 8 pages. https://doi.org/10.1145/3079368.3079413 [Cited on page 55]

Eric Hicks, Quang Tran, Kriangsiri Malasri, Nam Sy Vo, and Vinhthuy Phan. 2020. Active Learning: The Almost Silver Bullet. In *2020 12th International Conference on Knowledge and Systems Engineering (KSE)*. 131–135. https://doi.org/10.1109/KSE50997.2020.9287513 [Cited on page 33]

Rashina Hoda and Peter Andreae. 2014. It's Not Them, It's Us! Why Computer Science Fails to Impress Many First Years. In *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148* (Auckland, New Zealand) *(ACE '14)*. Australian Computer Society, Inc., AUS, 159–162. [Cited on page 18]

R. C. Holt, D. B. Wortman, D. T. Barnard, and J. R. Cordy. 1977. SP/k: A System for Teaching Computer Programming. *Commun. ACM* 20, 5 (may 1977), 301–309. https://doi.org/10.1145/359581.359586 [Cited on page 55]

Trudy Howles. 2009. A study of attrition and the use of student learning communities in the computer science introductory programming sequence. *Computer Science Education* 19, 1 (2009), 1–13. https://doi.org/10.1080/08993400902809312 arXiv:https://doi.org/10.1080/08993400902809312 [Cited on page 15]

Wen-Jung Hsin. 2005. Teaching Cryptography to Undergraduate Students in Small Liberal Art Schools. In *Proceedings of the 2nd Annual Conference on Information Security Curriculum Development* (Kennesaw, Georgia) *(InfoSecCD '05)*. Association for Computing Machinery (ACM), New York, NY, USA, 38–42. https://doi.org/10.1145/1107622.1107632 [Cited on page 90]

Mary A. Hudak and David E. Anderson. 1990. Formal Operations and Learning Style Predict Success in Statistics and Computer Science Courses. *Teaching of Psychology* 17, 4 (1990), 231–234. https://doi.org/10.1207/s15328023top1704_4 arXiv:https://doi.org/10.1207/s15328023top1704$_4$ [$Citedonpage$ 16]

Ann Hulbert. 2007. The Paradox of Play. Are kids today having enough fun? https://slate.com/human-interest/2007/06/the-new-play-movement.html. [Cited on page 53]

Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. 2002. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages & Computing* 13, 3 (2002), 259–290. https://doi.org/10.1006/jvlc.2002.0237 [Cited on page 29]

E. Hunt. 2001. Intelligence: Historical and Conceptual Perspectives. In *International Encyclopedia of the Social &amp Behavioral Sciences*. Elsevier, 7658–7663. https://doi.org/10.1016/b0-08-043076-7/01631-4 [Cited on page 19]

Katri Huutoniemi, Julie Thompson Klein, Henrik Bruun, and Janne Hukkinen. 2010. Analyzing interdisciplinarity: Typology and indicators. *Research Policy* 39, 1 (2010), 79–88. https://doi.org/10.1016/j.respol.2009.09.011 [Cited on page 94]

Petri Ihantola, Juho Leinonen, and Matti Rintala. 2020. *Students' Preferences Between Traditional and Video Lectures: Profiles and Study Success*. Association for Computing Machinery (ACM), New York, NY, USA. https://doi.org/10.1145/3428029.3428561 [Cited on page 57]

Cruz Izu, Amali Weerasinghe, and Cheryl Pope. 2016. A Study of Code Design Skills in Novice Programmers using the SOLO taxonomy. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*. Association for Computing Machinery (ACM), New York, NY, USA. https://doi.org/10.1145/2960310.2960324 [Cited on page 26]

Jerry A Jacobs. 2014. In defense of disciplines. In *In Defense of Disciplines*. University of Chicago Press. [Cited on page 94]

P. N. Johnson-Laird. 1983. *Mental models: towards a cognitive science of language, inference, and consciousness*. Number 6 in Cognitive science series. Harvard University Press, Cambridge, Mass. [Cited on page 27]

Joint Task Force on Cybersecurity Education. 2018. *Cybersecurity Curricula 2017: Curriculum Guidelines for Post-Secondary Degree Programs in Cybersecurity*. Association for Computing Machinery (ACM), New York, NY, USA. https://dl.acm.org/doi/book/10.1145/3184594 [Cited on pages 89, 90, and 248]

B. Joyce, M. Weil, and B. Showers. 1992. *Models of Teaching*. Allyn and Bacon, Boston, MA, USA. [Cited on page 82]

David A. Joyner, Qiaosi Wang, Suyash Thakare, Shan Jing, Ashok Goel, and Blair MacIntyre. 2020. The Synchronicity Paradox in Online Education. In *Proceedings of the Seventh ACM Conference on Learning @ Scale (L@S '20)*. Association for Computing Machinery (ACM), New York, NY, USA, 15–24. https://doi.org/10.1145/3386527.3405922 [Cited on page 191]

K-12 CS Framework. 2016. *K–12 Computer Science Framework*. Technical Report. http://www.k12cs.org [Cited on pages 89 and 216]

Daniel Kahneman. 2011. *Thinking, fast and slow*. FSG - Macmillan, New York, NY, USA. [Cited on page 18]

Sabrina Kalish. 2009. *Effects Of Instructor Immediacy And Student Need For Cognition On Student Motivation And Perceptions Of Learning*. Master's thesis. University of Central Florida. http://purl.fcla.edu/fcla/etd/CFE0002785 [Cited on page 160]

Shulamit Kapon and Sibel Erduran. 2021. Crossing Boundaries – Examining and Problematizing Interdisciplinarity in Science Education. In *Contributions from Science Education Research*. Springer International Publishing, Cham, Switzerland, 265–276. https://doi.org/10.1007/978-3-030-74490-8_21 [Cited on page 97]

Manu Kapur. 2016. Examining Productive Failure, Productive Success, Unproductive Failure, and Unproductive Success in Learning. *Educational Psychologist* 51, 2 (2016), 289–299. https://doi.org/10.1080/00461520.2016.1155457 arXiv:https://doi.org/10.1080/00461520.2016.1155457 [Cited on pages 50, 51, and 116]

Manu Kapur and Katerine Bielaczyc. 2012. Designing for Productive Failure. *Journal of the Learning Sciences* 21, 1 (Jan. 2012), 45–83. https://doi.org/10.1080/10508406.2011.591717 [Cited on pages 49, 50, 51, 111, 116, 120, and 121]

Jonathan Katz and Yehuda Lindell. 2007. *Introduction to Modern Cryptography*. Chapman and Hall/CRC, New York, NY, USA. https://doi.org/10.1201/9781420010756 [Cited on page 250]

Judy Kay, Michael Barg, Alan Fekete, Tony Greening, Owen Hollands, Jeffrey H. Kingston, and Kate Crawford. 2000. Problem-Based Learning for Foundation Computer Science Courses. *Computer Science Education* 10, 2 (aug 2000), 109–128. https://doi.org/10.1076/0899-3408(200008)10:2;1-c;ft109 [Cited on page 48]

Colin J. Kessel and Christopher D. Wickens. 1982. The Transfer of Failure-Detection Skills between Monitoring and Controlling Dynamic Systems. *Human Factors* 24, 1 (1982), 49–60. https://doi.org/10.1177/001872088202400106 arXiv:https://doi.org/10.1177/001872088202400106 [Cited on page 29]

Shaista E. Khilji. 2014. Human aspects of interdisciplinary research. *South Asian Journal of Global Business Research* 3, 1 (Feb. 2014), 2–10. https://doi.org/10.1108/sajgbr-12-2013-0090 [Cited on page 94]

Päivi Kinnunen and Lauri Malmi. 2006. Why Students Drop out CS1 Course?. In *Proceedings of the Second International Workshop on Computing Education Research* (Canterbury, United Kingdom) *(ICER '06)*. Association for Computing Machinery (ACM), New York, NY, USA, 97–108. https://doi.org/10.1145/1151588.1151604 [Cited on page 15]

Paul A. Kirschner, John Sweller, and Richard E. Clark. 2006. Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching. *Educational Psychologist* 41, 2 (June 2006), 75–86. https://doi.org/10.1207/s15326985ep4102_1 [Cited on page 42]

Julie Thompson Klein. 2010. A taxonomy of interdisciplinarity. In *The Oxford handbook of interdisciplinarity*, Robert Frodeman (Ed.). Oxford University Press, Oxford; New York, Chapter 15, 15. [Cited on pages 95 and 97]

William F. Klostermeyer. 2015. A Taxonomy of Perfect Domination. *Journal of Discrete Mathematical Sciences and Cryptography* 18, 1-2 (2015), 105–116. https://doi.org/10.1080/09720529.2014.914288 [Cited on pages 250 and 260]

Donald E. Knuth. 1972. George Forsythe and the Development of Computer Science. *Commun. ACM* 15, 8 (aug 1972), 721–726. https://doi.org/10.1145/361532.361538 [Cited on page 86]

David A. Kolb. 1984. *Experiential learning: experience as the source of learning and development*. Prentice-Hall, Englewood Cliffs, N.J. [Cited on pages 34 and 35]

Michael Kölling. 2003. The curse of hello world. In *Workshop on Learning and Teaching Object-Orientation–Scandinavian Perspectives*. Oslo, Norway. Invited Lecture. [Cited on page 23]

Michael Kölling. 2010. The Greenfoot Programming Environment. *ACM Transactions on Computing Education* 10, 4, Article 14 (nov 2010), 21 pages. https://doi.org/10.1145/1868358.1868361 [Cited on page 16]

Michael Kölling, Neil C. C. Brown, and Amjad Altadmri. 2015. Frame-Based Editing: Easing the Transition from Blocks to Text-Based Programming. In *Proceedings of the Workshop in Primary and Secondary Computing Education* (London, United Kingdom) *(WiPSCE '15)*. Association for Computing Machinery (ACM), New York, NY, USA, 29–38. https://doi.org/10.1145/2818314.2818331 [Cited on page 29]

Abdullah Konak. 2014. A cyber security discovery program: Hands-on cryptography. In *2014 IEEE Integrated STEM Education Conference*. IEEE. https://doi.org/10.1109/isecon.2014.6891029 [Cited on pages 92 and 249]

Abdullah Konak. 2018. Experiential Learning Builds Cybersecurity Self-Efficacy in K-12 Students. *Journal of Cybersecurity Education, Research and Practice* 2018, 1 (July 2018). https://digitalcommons.kennesaw.edu/jcerp/vol2018/iss1/6 [Cited on pages 91 and 249]

Shriram Krishnamurthi and Kathi Fisler. 2019. Programming Paradigms and Beyond. In *The Cambridge Handbook of Computing Education Research*, Sally A. Fincher and Anthony V.

Robins (Eds.). Cambridge University Press, Cambridge, United Kingdom, Chapter 13, 377–413. https://doi.org/10.1017/9781108654555.014 [Cited on page 29]

J. M. Lang. 2007. Did you learn anything? *The Chronicle of Higher Education* C1–C4 (march 2007). [Cited on page 160]

D. Laurillard. 2002. *Rethinking university teaching: A conversational framework for the effective use of learning technologies*. RoutledgeFalmer, London, United Kingdom. [Cited on page 182]

Ákos Lédeczi, Miklós Maróti, Hamid Zare, Bernard Yett, Nicole Hutchins, Brian Broll, Péter Völgyesi, Michael B. Smith, Timothy Darrah, Mary Metelko, Xenofon Koutsoukos, and Gautam Biswas. 2019. Teaching Cybersecurity with Networked Robots. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery (ACM), New York, NY, USA, 885–891. https://doi.org/10.1145/3287324.3287450 [Cited on page 91]

Irene Lee, Fred Martin, and Katie Apone. 2014. Integrating Computational Thinking across the K–8 Curriculum. *ACM Inroads* 5, 4 (dec 2014), 64–71. https://doi.org/10.1145/2684721.2684736 [Cited on page 52]

Irene Lee, Fred Martin, Jill Denner, Bob Coulter, Walter Allan, Jeri Erickson, Joyce Malyn-Smith, and Linda Werner. 2011. Computational Thinking for Youth in Practice. *ACM Inroads* 2, 1 (feb 2011), 32–37. https://doi.org/10.1145/1929887.1929902 [Cited on pages 52, 196, and 197]

David C. Leonard. 2002. *Learning Theories: A to Z*. Greenwood - ABC-CLIO. [Cited on page 36]

J. Paul Leonard. 1930. *The Use of Practice Exercises in Teaching Capitalization and Punctuation*. Vol. 21. https://doi.org/10.1080/00220671.1930.10880030. 186–190 pages. [Cited on page 55]

O. Levrini, L. Branchetti, and P. Fantini. 2019. In *S. Kapon (Chair) & S. Erduran (Discussant), Crossing boundaries – Examining and problematizing interdisciplinarity in science education* (Bologna, Italy) *(European Science Education Research Association 2019 Biannual Conference)*. Invited symposium. [Cited on pages 97 and 98]

S. T. Levy, A. R. Zohar, and I. Dubovi. 2019. Slipping between disciplines: How forming causal explanations may compel crossing disciplinary boundaries. In *S. Kapon (Chair) & S. Erduran (Discussant), Crossing boundaries – Examining and problematizing interdisciplinarity in science education* (Bologna, Italy) *(European Science Education Research Association 2019 Biannual Conference)*. Invited symposium. [Cited on page 97]

Colleen M. Lewis. 2017. Good (and Bad) Reasons to Teach All Students Computer Science. In *New Directions for Computing Education: Embedding Computing Across Disciplines*, Samuel B. Fee, Amanda M. Holland-Minkley, and Thomas E. Lombardi (Eds.). Springer International Publishing, Cham, Switzerland, 15–34. https://doi.org/10.1007/978-3-319-54226-3_2 [Cited on page 67]

Colleen M. Lewis, Michael J. Clancy, and Jan Vahrenhold. 2019a. Pedagogic Approaches. In *The Cambridge Handbook of Computing Education Research*, Sally A. Fincher and Anthony V. Robins (Eds.). Cambridge University Press, Cambridge, United Kingdom, Chapter 27, 773–800. https://doi.org/10.1017/9781108654555.028 [Cited on page 28]

Colleen M. Lewis, Niral Shah, and Katrina Falkner. 2019b. Equity and Diversity. In *The Cambridge Handbook of Computing Education Research*, Sally A. Fincher and Anthony V. Robins (Eds.). Cambridge University Press, Cambridge, United Kingdom, Chapter 16, 481–510. https://doi.org/10.1017/9781108654555.017 [Cited on page 14]

Sarah Lewis. 2015. Qualitative Inquiry and Research Design: Choosing Among Five Approaches. *Health Promotion Practice* 16, 4 (April 2015), 473–475. https://doi.org/10.1177/1524839915580941 [Cited on page 193]

Yeping Li, Alan H. Schoenfeld, Andrea A. diSessa, Arthur C. Graesser, Lisa C. Benson, Lyn D. English, and Richard A. Duschl. 2019. On Thinking and STEM Education. *Journal for STEM Education Research* 2, 1 (Feb. 2019), 1–13. https://doi.org/10.1007/s41979-019-00014-x [Cited on page 67]

Liceo Matematico. [n.d.]. Retrieved December 29, 2022 from https://www.liceomatematico.it/ [Cited on page 222]

John Lidstone and Paul Shield. 2010. Virtual reality or virtually real: blended teaching and learning in a Master's level research methods class. In *Cases on Online and Blended Learning Technologies in Higher Education: Concepts and Practices*, Yukiko Inoue (Ed.). IGI Global,, Hershey, PA, USA, 91–111. [Cited on page 183]

Janet Mei-Chuen Lin and Cheng-Chih Wu. 2007. Suggestions for content selection and presentation in high school computer textbooks. *Computers & Education* 48, 3 (2007), 508–521. [Cited on page 23]

Anke Lindmeier and Andreas Mühling. 2020. Keeping Secrets: K-12 Students' Understanding of Cryptography. In *Proceedings of the 15th Workshop on Primary and Secondary Computing Education* (Virtual Event, Germany) *(WiPSCE '20)*. Association for Computing Machinery (ACM), New York, NY, USA, Article 14, 10 pages. https://doi.org/10.1145/3421590.3421630 [Cited on pages 89, 216, and 249]

Raymond Lister. 2016. Toward a Developmental Epistemology of Computer Programming. In *Proceedings of the 11th Workshop in Primary and Secondary Computing Education* (Münster, Germany) *(WiPSCE '16)*. Association for Computing Machinery, New York, NY, USA, 5–16. https://doi.org/10.1145/2978249.2978251 [Cited on page 54]

Raymond Lister, Tony Clear, Simon, Dennis J. Bouvier, Paul Carter, Anna Eckerdal, Jana Jacková, Mike Lopez, Robert McCartney, Phil Robbins, Otto Seppälä, and Errol Thompson. 2010. Naturally Occurring Data as Research Instrument: Analyzing Examination Responses to Study the Novice Programmer. *ACM SIGCSE Bulletin* 41, 4 (jan 2010), 156–173. https://doi.org/10.1145/1709424.1709460 [Cited on page 17]

Richard Lobb and Jenny Harlow. 2016. Coderunner: A Tool for Assessing Computer Programming Skills. *ACM Inroads* 7, 1 (Feb. 2016), 47–51. https://doi.org/10.1145/2810041 [Cited on page 185]

Michael Lodi. 2020a. Informatical Thinking. *Olympiads In Informatics* 14 (Dec. 2020), 113–132. https://doi.org/10.15388/ioi.2020.09 [Cited on pages 13, 67, 68, 79, and 117]

Michael Lodi. 2020b. *Introducing computational thinking in k-12 education: historical, epistemological, pedagogical, cognitive, and affective aspects*. Ph.D. Dissertation. Alma Mater Studiorum - Università di Bologna. https://doi.org/10.6092/unibo/amsdottorato/9188 [Cited on pages 41 and 83]

Michael Lodi and Simone Martini. 2021. Computational Thinking, Between Papert and Wing. *Science & Education* 30, 4 (Aug. 2021), 883–908. https://doi.org/10.1007/s11191-021-00202-5 [Cited on pages 13, 15, 66, 79, and 117]

Michael Lodi, Simone Martini, and Enrico Nardelli. 2017. Do we really need computational thinking? *Mondo Digitale* 72, Article 2 (Nov. 2017), 15 pages. http://mondodigitale.aicanet.net/2017-5/articoli/MD72_02_abbiamo_davvero_bisogno_del_pensiero_computazionale.pdf In Italian. [Cited on pages 66 and 79]

Micahel Lodi, Simone Martini, and Marco Sbaraglia. 2022a. Crittografia a blocchi al Liceo Matematico. In *Cryptography and Coding Theory Conference 2021*. Collectio Ciphrarum, Vol. 3. Aracne, Roma, Italy, 103–104. https://doi.org/10.53136/979125994981340 In Italian. [Cited on page 218]

Micahel Lodi, Simone Martini, and Marco Sbaraglia. 2023. Programmare per imparare la crittografia al Liceo Matematico. *Rendiconti del Seminario Matematico* 80, 2 (2023). http://www.seminariomatematico.polito.it/rendiconti/ In Italian; in press. [Cited on page 218]

Michael Lodi, Marco Sbaraglia, and Simone Martini. 2021a. Resources of "Big Ideas of Cryptography in K-12". https://bigideascryptok12.bitbucket.io/ [Cited on pages 217, 218, 230, 231, 232, 233, 234, 242, and 243]

Michael Lodi, Marco Sbaraglia, and Simone Martini. 2022b. Cryptography in Grade 10: Core Ideas with Snap! and Unplugged. In *Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education Vol. 1* (Dublin, Ireland) *(ITiCSE '22)*. Association for Computing Machinery (ACM), New York, NY, USA, 7. https://doi.org/10.1145/3502718.3524767 [Cited on pages 216, 217, 218, 221, and 222]

Michael Lodi, Marco Sbaraglia, Stefano Pio Zingaro, and Simone Martini. 2021b. The Online Course Was Great: I Would Attend It Face-to-Face: The Good, The Bad, and the Ugly of IT in Emergency Remote Teaching of CS1. In *Proceedings of the ACM Conference on Information Technology for Social Good* (Roma, Italy) *(GoodIT '21)*. Association for Computing Machinery (ACM), New York, NY, USA, 242–247. https://doi.org/10.1145/3462203.3475902 [Cited on pages 131, 181, and 242]

Katharina Loibl, Ido Roll, and Nikol Rummel. 2017. Towards a Theory of When and How Problem Solving Followed by Instruction Supports Learning. *Educational Psychology Review* 29, 4 (Dec. 2017), 693–715. https://doi.org/10.1007/s10648-016-9379-x [Cited on pages 50, 51, 111, 116, and 120]

Katharina Loibl and Nikol Rummel. 2013. The impact of guidance during problem-solving prior to instruction on students' inventions and learning outcomes. *Instructional Science* 42, 3 (June 2013), 305–326. https://doi.org/10.1007/s11251-013-9282-5 [Cited on page 51]

Michael C. Loui and Maura Borrego. 2019. Engineering Education Research. In *The Cambridge Handbook of Computing Education Research*, Sally A. Fincher and Anthony V. Robins (Eds.). Cambridge University Press, Cambridge, United Kingdom, Chapter 11, 292–322. https://doi.org/10.1017/9781108654555.012 [Cited on pages 216 and 220]

Stephanie Lunn, Maíra Marques Samary, and Alan Peterfreund. 2021. Where is Computer Science Education Research Happening?. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21)*. Association for Computing Machinery (ACM), New York, NY, USA, 288–294. https://doi.org/10.1145/3408877.3432375 [Cited on page 21]

Andrew Luxton-Reilly. 2016. Learning to Program is Easy *(ITiCSE '16)*. Association for Computing Machinery (ACM), New York, NY, USA, 284–289. https://doi.org/10.1145/2899415.2899432 [Cited on page 15]

Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory Programming: A Systematic Literature Review *(ITiCSE 2018 Companion)*. Association for Computing Machinery (ACM), New York, NY, USA, 55–106. https://doi.org/10.1145/3293881.3295779 [Cited on page 21]

Nicholas Lytle, Veronica Cateté, Danielle Boulden, Yihuan Dong, Jennifer Houchins, Alexandra Milliken, Amy Isvik, Dolly Bounajim, Eric Wiebe, and Tiffany Barnes. 2019. Use, Modify, Create: Comparing Computational Thinking Lesson Progressions for STEM Classes. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE '19)*. Association for Computing Machinery (ACM), New York, NY, USA, 395–401. https://doi.org/10.1145/3304221.3319786 [Cited on pages 52 and 208]

Jun Ma, Jun Tao, Jean Mayo, Ching-Kuang Shene, Melissa Keranen, and Chaoli Wang. 2016. AESvisual: A Visualization Tool for the AES Cipher. In *Proceedings of the 21st ACM Conference on Innovation & Technology in Computer Science Education* (Arequipa, Peru) *(ITiCSE '16)*. Association for Computing Machinery (ACM), New York, NY, USA, 230–235. https://doi.org/10.1145/2899415.2899425 [Cited on pages 91 and 249]

Miles MacLeod. 2016. What makes interdisciplinarity difficult? Some consequences of domain specificity in interdisciplinary practice. *Synthese* 195, 2 (Oct. 2016), 697–720. https://doi.org/10.1007/s11229-016-1236-4 [Cited on page 94]

Lauri Malmi. 2020. COMPUTING EDUCATION RESEARCH The New Normal of Teaching Computer Science. *ACM Inroads* 11, 4 (Nov. 2020), 17–19. https://doi.org/10.1145/3433692 [Cited on page 58]

Linda Mannila, Mia Peltomäki, and Tapio Salakoski. 2006. What about a simple language? Analyzing the difficulties in learning to program. *Computer Science Education* 16, 3 (2006), 211–227. https://doi.org/10.1080/08993400600912384 arXiv:https://doi.org/10.1080/08993400600912384 [Cited on page 54]

Lauren E. Margulieux, Brian Dorn, and Kristin A. Searle. 2019. Learning Sciences for Computing Education. In *The Cambridge Handbook of Computing Education Research*, Sally A. Fincher and Anthony V. Robins (Eds.). Cambridge University Press, Cambridge, United Kingdom, Chapter 8, 208–230. https://doi.org/10.1017/9781108654555.009 [Cited on page 42]

Maria A. Martinez, Narcis Sauleda, and Güenter L. Huber. 2001. Metaphors as blueprints of thinking about teaching and learning. *Teaching and Teacher Education* 17, 8 (Nov. 2001), 965–977. https://doi.org/10.1016/s0742-051x(01)00043-9 [Cited on page 37]

Simone Martini. 2016a. Several Types of Types in Programming Languages. In *History and Philosophy of Computing (IFIP Advances in Information and Communication Technology)*, Fabio Gadducci and Mirko Tavosanis (Eds.). Springer International Publishing, Cham, Switzerland, 216–227. https://doi.org/10.1007/978-3-319-47286-7_15 [Cited on page 127]

Simone Martini. 2016b. Types in Programming Languages, between Modelling, Abstraction, and Correctness. In *CiE 2016: Pursuit of the Universal (LNCS)*, Arnold Beckmann, Laurent Bienvenu, and Nataša Jonoska (Eds.), Vol. 9709. Springer International Publishing, Cham, Switzerland, 164–169. https://doi.org/10.1007/978-3-319-40189-8_17 [Cited on page 127]

Simone Martini. 2020. The Standard Model for Programming Languages: The Birth of a Mathematical Theory of Computation. In *Recent Developments in the Design and Implementation of Programming Languages (OpenAccess Series in Informatics (OASIcs))*, Frank S. de Boer and Jacopo Mauro (Eds.), Vol. 86. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1–13. https://doi.org/10.4230/OASIcs.Gabbrielli.8 [Cited on page 123]

Michael R. Matthews. 1997. Introductory Comments on Philosophy and Constructivism in Science Education. *Science & Education* 6, 1 (01 Jan. 1997), 5–14. https://doi.org/10.1023/A:1008650823980 [Cited on page 42]

Alasdair McAndrew. 2008. Teaching Cryptography with Open-Source Software. *SIGCSE Bull.* 40, 1 (March 2008), 325–329. https://doi.org/10.1145/1352322.1352247 [Cited on page 91]

Brendan McCane, Claudia Ott, Nick Meek, and Anthony Robins. 2017. Mastery Learning in Introductory Programming. In *Proceedings of the Nineteenth Australasian Computing Education Conference* (Geelong, VIC, Australia) *(ACE '17)*. Association for Computing Machinery (ACM), New York, NY, USA, 1–10. https://doi.org/10.1145/3013499.3013501 [Cited on page 18]

Jeffrey J. McConnell. 1996. Active Learning and Its Use in Computer Science. *SIGCUE Outlook* 24, 1–3 (jan 1996), 52–54. https://doi.org/10.1145/1013718.237526 [Cited on pages 34 and 35]

Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-Year CS Students. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education* (Canterbury, United Kingdom) *(ITiCSE-WGR '01)*. Association for Computing Machinery (ACM), New York, NY, USA, 125–180. https://doi.org/10.1145/572133.572137 [Cited on page 25]

James C McCroskey, Ari Sallinen, Judith M Fayer, Virginia P Richmond, and Robert A Barraclough. 1996. Nonverbal immediacy and cognitive learning: A cross-cultural investigation. *Communication Education* 45 (1996), 200–211. [Cited on page 160]

Andrew Mcgettrick, Roger Boyle, Roland Ibbett, John Lloyd, Gillian Lovegrove, and Keith Mander. 2005. Grand Challenges in Computing: Education—A Summary. *Comput. J.* 48, 1 (jan 2005), 42–48. https://doi.org/10.1093/comjnl/bxh064 [Cited on pages 21, 23, and 24]

Tanya J. McGill and Simone E. Volet. 1997. A Conceptual Framework for Analyzing Students' Knowledge of Programming. *Journal of Research on Computing in Education* 29, 3 (1997), 276–297. https://doi.org/10.1080/08886504.1997.10782199 arXiv:https://doi.org/10.1080/08886504.1997.10782199 [Cited on pages 24, 25, and 28]

Laura Meagher. 2017. Teaching London Computing Follow-up Evaluation through Interviews with Teachers. (2017), 1–10. https://www.london.gov.uk/sites/default/files/lsef_legacy_interviews_evaluation_report-final.pdf [Cited on page 83]

Antonio Jose Mendes, Luis Paquete, Amilcar Cardoso, and Anabela Gomes. 2012. Increasing student commitment in introductory programming learning. In *2012 Frontiers in Education Conference Proceedings*. IEEE. https://doi.org/10.1109/fie.2012.6462486 [Cited on page 15]

Jan Meyer and Ray Land. 2006. *Overcoming Barriers to Student Understanding*. Routledge. https://doi.org/10.4324/9780203966273 [Cited on page 18]

Lorenzo Miani. 2021. *Highlighting interdisciplinarity between physics and mathematics in historical papers on special relativity: design of blended activities for pre-service teacher education*. Master's thesis. http://amslaurea.unibo.it/23544/ Master thesis in Physics, University of Bologna. [Cited on page 248]

MIUR. 2010. Regolamento Licei del 16/02/2010. Retrieved December 29, 2022 from https://archivio.pubblica.istruzione.it/riforma_superiori/nuovesuperiori/doc/Regolamento_licei_definitivo_16.02.2010.pdf [Cited on page 221]

Mahnaz Moallem, Woei Hung, and Nada Dabbagh (Eds.). 2019. *The Wiley Handbook of Problem-Based Learning*. John Wiley & Sons, Hoboken, NJ. https://doi.org/10.1002/9781119173243 [Cited on page 47]

Simon Modeste. 2016. Impact of Informatics on Mathematics and Its Teaching. In *History and Philosophy of Computing*, Fabio Gadducci and Mirko Tavosanis (Eds.). Springer International Publishing, Cham, Switzerland, 243–255. [Cited on page 248]

Thomas K. Moore. 1993. Scientific Investigation in a Breadth-First Approach to Introductory Computer Science. *SIGCSE Bull.* 25, 1 (mar 1993), 63–67. https://doi.org/10.1145/169073.169350 [Cited on page 82]

Michael Morgan, Jane Sinclair, Matthew Butler, Neena Thota, Janet Fraser, Gerry Cross, and Jana Jackova. 2018. Understanding International Benchmarks on Student Engagement: Awareness and Research Alignment from a Computer Science Perspective. In *Proceedings of the 2017 ITiCSE Conference on Working Group Reports* (Bologna, Italy) *(ITiCSE-WGR '17)*. Association for Computing Machinery (ACM), New York, NY, USA, 1–24. https://doi.org/10.1145/3174781.3174782 [Cited on page 15]

Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide. 2002. Exploring the Role of Visualization and Engagement in Computer Science Education. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education* (Aarhus, Denmark) *(ITiCSE-WGR '02)*. Association for Computing Machinery (ACM), New York, NY, USA, 131–152. https://doi.org/10.1145/960568.782998 [Cited on page 29]

Enrico Nardelli, Luca Forlizzi, Michael Lodi, Violetta Lonati, Claudio Mirolo, Mattia Monga, Alberto Montresor, and Anna Morpurgo. 2017. *Proposal for a national Informatics curriculum in the Italian school*. Technical Report. CINI. https://www.consorzio-cini.it/images/PROPOSAL-Informatics-curriculum-Italian-school.pdf [Cited on pages 195, 198, and 199]

Enrico Nardelli, Francesco Lacchia, Veronica Rossano, Enrichetta Gentile, Luca Forlizzi, Giovanna Melideo, Sara Capecchi, Ilenia Fronza, Tullio Vardanega, Renzo Davoli, Michael Lodi, Marco Sbaraglia, Violetta Lonati, Mattia Monga, and Anna Morpurgo. 2023. Learning Iteration for Grades 2-3: Puzzles vs. UMC in Code.org. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education* (Toronto, ON, Canada) *(SIGCSE '23)*. Association for Computing Machinery (ACM), 1. https://doi.org/10.1145/3545947.3576312 In press. [Cited on page 197]

Greg L. Nelson and Amy J. Ko. 2018. On Use of Theory in Computing Education Research. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. Association for Computing Machinery (ACM), New York, NY, USA. https://doi.org/10.1145/3230977.3230992 [Cited on page 149]

Greg L. Nelson, Benjamin Xie, and Amy J. Ko. 2017. Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. Association for Computing Machinery (ACM), New York, NY, USA, 2–11. https://doi.org/10.1145/3105726.3106178 [Cited on page 29]

R Newman, R Gatward, and M Poppleton. 1970. Paradigms for teaching computer programming in higher education. *WIT Transactions on Information and Communication Technologies* 7 (1970). [Cited on page 15]

Elena Novak and Javed Khan. 2022. A Research-Practice Partnership Approach for Co-Designing a Culturally Responsive Computer Science Curriculum for Upper Elementary Students. *TechTrends* (apr 2022), 1–12. https://doi.org/10.1007/s11528-022-00730-z [Cited on page 59]

Esko Nuutila, Seppo Törmä, Päivi Kinnunen, and Lauri Malmi. 2008. Learning Programming with the PBL Method — Experiences on PBL Cases and Tutoring. In *Reflections on the Teaching of Programming: Methods and Implementations*, Jens Bennedsen, Michael E. Caspersen, and Michael Kölling (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, Germany, 47–67. https://doi.org/10.1007/978-3-540-77934-6_5 [Cited on pages 47 and 48]

OECD. 2019. *Education at a Glance 2019*. 497 pages. https://doi.org/https://doi.org/10.1787/f8d7880d-en [Cited on page 94]

Michael J. O'Grady. 2012. Practical Problem-Based Learning in Computing Education. *ACM Transactions on Computing Education* 12, 3 (jul 2012), 1–16. https://doi.org/10.1145/2275597.2275599 [Cited on page 48]

Armanda Maria C. Amorim Oliveira, Simone C. dos Santos, and Vinicius Cardoso Garcia. 2013. PBL in teaching computing: An overview of the last 15 years. In *2013 IEEE Frontiers in Education Conference (FIE)*. IEEE, Oklahoma City, Oklahoma. https://doi.org/10.1109/fie.2013.6684830 [Cited on pages 47 and 48]

Jairo Ortiz-Revilla, Agustín Adúriz-Bravo, and Ileana M. Greca. 2020. A Framework for Epistemological Discussion on Integrated STEM Education. *Science &amp Education* 29, 4 (June 2020), 857–880. https://doi.org/10.1007/s11191-020-00131-9 [Cited on page 94]

Fred Paas, Alexander Renkl, and John Sweller. 2003. Cognitive Load Theory and Instructional Design: Recent Developments. *Educational Psychologist* 38, 1 (2003), 1–4. https://doi.org/10.1207/S15326985EP3801_1 arXiv:https://doi.org/10.1207/S15326985EP3801$_1 x [Citedonpage 18]$

John F. Pane, Chotirat Ratanamahatana, and Brad A. Myers. 2001. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies* 54, 2 (2001), 237–264. https://doi.org/10.1006/ijhc.2000.0410 [Cited on page 134]

Seymour Papert. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., New York, NY, USA. [Cited on pages 55, 66, 86, 131, 229, and 230]

Seymour Papert and Idit Harel. 1991. Situating Constructionism. In *Constructionism*, Seymour Papert and Idit Harel (Eds.). Ablex Publishing Corporation, Norwood, NJ, Chapter 1. [Cited on pages 66, 83, and 230]

Roy D. Pea. 1986. Language-Independent Conceptual "Bugs" in Novice Programming. *Journal of Educational Computing Research* 2, 1 (Feb. 1986), 25–36. https://doi.org/10.2190/689t-1r2a-x4w4-29j2 [Cited on page 27]

Nik Peachey. 2017. Synchronous Online Teaching. In *Digital Language Learning and Teaching*, M. Carrier et al. (Eds.). Routledge, New York, NY, USA. [Cited on pages 57, 58, and 190]

Wu Peng. 2010. Practice and experience in the application of problem-based learning in computer programming course. In *2010 International Conference on Educational and Information Technology*. IEEE, Chongqing, China. https://doi.org/10.1109/iceit.2010.5607778 [Cited on page 47]

Schwartz S. Perkins, D. N. and R. Simmons. 1988. *Instructional strategies for the problems of novice programmers*. L. Erlbaum Associates, Hillsdale, N.J, 153–178. [Cited on page 27]

Alan J. Perlis. 1962. The Computer in the University. In *Computers and the World of the Future*, Martin Greenberger (Ed.). The MIT Press, Cambridge, Massachusetts, USA. [Cited on page 86]

E.J. Pharo, A. Davison, K. Warr, M. Nursey-Bray, K. Beswick, E. Wapstra, and C. Jones. 2012. Can teacher collaboration overcome barriers to interdisciplinary learning in a disciplinary university? A case study using climate change. *Teaching in Higher Education* 17, 5 (Oct. 2012), 497–507. https://doi.org/10.1080/13562517.2012.658560 [Cited on page 94]

Jean Piaget. 1973. *To Understand is to Invent: The Future of Education*. Grossman Publishers, New York, NY, USA. [Cited on page 38]

L Pino-Fan, Ismenia Guzmán, Raymond Duval, and Vicenç Font. 2015. The theory of registers of semiotic representation and the onto-semiotic approach to mathematical cognition and instruction: linking looks for the study of mathematical understanding. In *Proceedings of the 39th Conference of the International Group for the Psychology of Mathematics Education*, Vol. 4. PME, Hobart, Australia, 33–40. [Cited on page 253]

Martinha Piteira and Carlos Costa. 2013. Learning Computer Programming: Study of Difficulties in Learning Programming. In *Proceedings of the 2013 International Conference on Information Systems and Design of Communication* (Lisboa, Portugal) *(ISDOC '13)*. Association for Computing Machinery (ACM), New York, NY, USA, 75–80. https://doi.org/10.1145/2503859.2503871 [Cited on page 54]

Jan L. Plass, Roxana Moreno, and Roland Brünken (Eds.). 2010. *Cognitive Load Theory*. Cambridge University Press. https://doi.org/10.1017/cbo9780511844744 [Cited on page 18]

Leo Porter and Daniel Zingaro. 2014. Importance of Early Performance in CS1: Two Conflicting Assessment Stories. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (Atlanta, Georgia, USA) *(SIGCSE '14)*. Association for Computing Machinery (ACM), New York, NY, USA, 295–300. https://doi.org/10.1145/2538862.2538912 [Cited on page 18]

Leo Porter, Daniel Zingaro, and Raymond Lister. 2014. Predicting Student Success Using Fine Grain Clicker Data. In *Proceedings of the Tenth Annual Conference on International Computing Education Research* (Glasgow, Scotland, United Kingdom) *(ICER '14)*. Association for Computing Machinery (ACM), New York, NY, USA, 51–58. https://doi.org/10.1145/2632320.2632354 [Cited on page 18]

Scott R. Portnoff. 2018. The Introductory Computer Programming Course is First and Foremost a Language Course. *ACM Inroads* 9, 2 (apr 2018), 34–52. https://doi.org/10.1145/3152433 [Cited on page 55]

Michael Prince. 2004. Does Active Learning Work? A Review of the Research. *Journal of Engineering Education* 93, 3 (July 2004), 223–231. https://doi.org/10.1002/j.2168-9830.2004.tb00809.x [Cited on pages 34, 45, and 115]

Michael J. Prince and Richard M. Felder. 2006. Inductive Teaching and Learning Methods: Definitions, Comparisons, and Research Bases. *Journal of Engineering Education* 95, 2 (April 2006), 123–138. https://doi.org/10.1002/j.2168-9830.2006.tb00884.x [Cited on pages 40, 41, 47, and 82]

Noa Ragonis and Mordechai Ben-Ari. 2005. A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education* 15, 3 (Sept. 2005), 203–221. https://doi.org/10.1080/08993400500224310 [Cited on page 28]

Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. https://doi.org/10.1145/1592761.1592779 [Cited on page 55]

John W. Rice. 2012. The Gamification of Learning and Instruction: Game-Based Methods and Strategies for Training and Education. *Int. J. Gaming Comput. Mediat. Simul.* 4, 4 (Oct. 2012), 81–83. https://doi.org/10.4018/jgcms.2012100106 [Cited on page 37]

Virginia P Richmond. 1990. Communication in the classroom: Power and motivation. *Communication Education* 39 (1990), 181–195. [Cited on page 160]

Virginia P Richmond, John S Gorham, and James C McCroskey. 1987. The relationship between selected immediacy behaviors and cognitive learning. *Communication Yearbook* 10 (1987), 574–590. [Cited on page 160]

Robert S. Rist. 1989. Schema Creation in Programming. *Cognitive Science* 13, 3 (July 1989), 389–414. https://doi.org/10.1207/s15516709cog1303_3 [Cited on pages 30 and 31]

Robert S. Rist. 1991. Knowledge Creation and Retrieval in Program Design: A Comparison of Novice and intermediate Student Programmers. *Human–Computer Interaction* 6, 1 (1991), 1–46. https://doi.org/10.1207/s15327051hci0601_1 arXiv:https://doi.org/10.1207/s15327051hci0601$_1$ [$Citedonpages30and31$]

Robert S Rist. 2005. Learning to Program: Schema Creation, Application, and Evaluation. In *Computer Science Education Research*, Sally Fincher and Marian Petre (Eds.). Taylor & Francis, Chapter 5, 175–197. [Cited on pages 30 and 31]

Eric Roberts. 2001. An Overview of MiniJava. *SIGCSE Bull.* 33, 1 (feb 2001), 1–5. https://doi.org/10.1145/366413.364525 [Cited on page 55]

Anthony Robins. 2010. Learning edge momentum: a new account of outcomes in CS1. *Computer Science Education* 20, 1 (2010), 37–71. https://doi.org/10.1080/08993401003612167 arXiv:https://doi.org/10.1080/08993401003612167 [Cited on pages 16, 17, and 46]

Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education* 13, 2 (2003), 137–172. https://doi.org/10.1076/csed.13.2.137.14200 arXiv:https://doi.org/10.1076/csed.13.2.137.14200 [Cited on pages 14 and 23]

Anthony V Robins. 2018. *Outcomes in introductory programming*. Technical Report OUCS-2018-02. Department of Computer Science, University of Otago. https://www.otago.ac.nz/computer-science/otago685184.pdf [Cited on page 16]

Anthony V. Robins. 2019. Novice Programmers and Introductory Programming. In *The Cambridge Handbook of Computing Education Research*, Sally A. Fincher and Anthony V. Robins (Eds.). Cambridge University Press, Cambridge, United Kingdom, Chapter 12, 327–376. https://doi.org/10.1017/9781108654555.013 [Cited on pages 16, 17, 18, 21, 28, 45, and 46]

Anthony V. Robins. 2022. Dual Process Theories: Computing Cognition in Context. *ACM Transactions on Computing Education* 22, 4, Article 41 (sep 2022), 31 pages. https://doi.org/10.1145/3487055 [Cited on pages 18 and 19]

Anthony V. Robins, Lauren E. Margulieux, and Briana B. Morrison. 2019. Cognitive Sciences for Computing Education. In *The Cambridge Handbook of Computing Education Research*, Sally A. Fincher and Anthony V. Robins (Eds.). Cambridge University Press, Cambridge, United Kingdom, Chapter 9, 231–275. https://doi.org/10.1017/9781108654555.010 [Cited on page 38]

Jeremy Roschelle and William R. Penuel. 2006. Co-Design of Innovations with Teachers: Definition and Dynamics. In *Proceedings of the 7th International Conference on Learning Sciences* (Bloomington, Indiana, USA) *(ICLS '06)*. International Society of the Learning Sciences, 606–612. [Cited on page 59]

Kjell Erik Rudestam and Judith Schoenholtz-Read. 2010. The Flourishing of Adult Online Education. *Handbook of Online Learning* (2010). [Cited on pages 57 and 191]

Yvan Russell. 2022. Three Problems of Interdisciplinarity. *Avant* 13 (08 2022), 1–19. https://doi.org/10.26913/avant [Cited on page 94]

Philip Sadler and Gerhard Sonnert. 2018. The Path to College Calculus: The Impact of High School Mathematics Coursework. *Journal for Research in Mathematics Education* 49, 3 (May 2018), 292–329. https://doi.org/10.5951/jresematheduc.49.3.0292 [Cited on page 87]

Jean Salac, Cathy Thomas, Chloe Butler, Ashley Sanchez, and Diana Franklin. 2020. TIPP&SEE: A Learning Strategy to Guide Students through Use - Modify Scratch Activities. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery (ACM), New York, NY, USA, 79–85. https://doi.org/10.1145/3328778.3366821 [Cited on pages 53 and 205]

Kate Sanders, Jonas Boustedt, Anna Eckerdal, Robert McCartney, and Carol Zander. 2017. Folk Pedagogy: Nobody Doesn't Like Active Learning. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. Association for Computing Machinery (ACM), New York, NY, USA, 145–154. https://doi.org/10.1145/3105726.3106192 [Cited on pages 33 and 35]

John R. Savery and Thomas M. Duffy. 1995. Problem Based Learning: An Instructional Model and Its Constructivist Framework. *Educational Technology* 35, 5 (1995), 31–38. http://www.jstor.org/stable/44428296 [Cited on page 29]

Marco Sbaraglia. 2021. A Necessity-Driven Learning Design for Computer Science. In *Proceedings of the 26th ACM Conference on Innovation & Technology in Computer Science Education V. 2* (Virtual Event, Germany) *(ITiCSE '21)*. Association for Computing Machinery (ACM), New York, NY, USA, 664–665. https://doi.org/10.1145/3456565.3460017 [Cited on pages 109, 216, 220, and 243]

Marco Sbaraglia, Michael Lodi, and Simone Martini. 2021a. A Necessity-Driven Ride on the Abstraction Rollercoaster of CS1 Programming. *Informatics in Education* 20, 4 (dec 2021), 641–682. https://doi.org/10.15388/infedu.2021.28 [Cited on pages 115 and 216]

Marco Sbaraglia, Michael Lodi, Stefano Pio Zingaro, and Simone Martini. 2021b. The Good, The Bad, and The Ugly of a Synchronous Online CS1. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 2 (ITiCSE '21)*. Association for Computing Machinery (ACM), 1. https://doi.org/10.1145/3456565.3460075 [Cited on page 181]

Marco Sbaraglia, Michael Lodi, Stefano Pio Zingaro, and Simone Martini. 2021c. *Questionnaire*. Retrieved January 7, 2023 from https://figshare.com/s/ac683cb7fecd743794f6 [Cited on page 185]

Cathryne Schmitz, C.H. Stinson, and Channelle James. 2010. Community and environmental sustainability: Collaboration and interdisciplinary education. *Critical Social Work* 11 (01 2010), 83–100. [Cited on page 94]

Donald A. Schön. 1995. Educating the reflective legal practitioner. *Clinical L. Rev.* 2 (1995), 231. [Cited on page 34]

Carsten Schulte. 2008. Block Model: An Educational Model of Program Comprehension as a Tool for a Scholarly Approach to Teaching. In *Proceedings of the Fourth International Workshop on Computing Education Research (ICER '08)*. Association for Computing Machinery (ACM), New York, NY, USA, 149–160. https://doi.org/10.1145/1404520.1404535 [Cited on pages 25 and 28]

Robert M. Schumacher and Mary P. Czerwinski. 1992. *Mental Models and the Acquisition of Expert Knowledge*. Springer-Verlag, Berlin, Heidelberg, Germany, 61–79. [Cited on page 29]

M. Schvartzer, T. Peer, and S. Kapon. 2019. Learning physics through Maker projects - Between disciplinary authenticity and personal relevance. In *S. Kapon (Chair) & S. Erduran (Discussant), Crossing boundaries – Examining and problematizing interdisciplinarity in science education* (Bologna, Italy) *(European Science Education Research Association 2019 Biannual Conference)*. Invited symposium. [Cited on page 97]

Daniel L. Schwartz and John D. Bransford. 1998. A Time For Telling. *Cognition and Instruction* 16, 4 (Dec. 1998), 475–5223. https://doi.org/10.1207/s1532690xci1604_4 [Cited on pages 48, 116, and 119]

Daniel L. Schwartz and Taylor Martin. 2004. Inventing to Prepare for Future Learning: The Hidden Efficiency of Encouraging Original Student Production in Statistics Instruction. *Cognition and Instruction* 22, 2 (June 2004), 129–184. https://doi.org/10.1207/s1532690xci2202_1 [Cited on pages 49 and 51]

Dino Schweitzer and Jeff Boleng. 2009. Designing Web Labs for Teaching Security Concepts. *J. Comput. Sci. Coll.* 25, 2 (Dec. 2009), 39–45. [Cited on page 91]

Dino Schweitzer and Wayne Brown. 2009. Using Visualization to Teach Security. *Journal of Computing Sciences in Colleges* 24, 5 (May 2009), 143–150. [Cited on pages 91 and 249]

Andreas Schwill. 1994. Fundamental ideas of computer science. *Bulletin - European Association for Theoretical Computer Science* 53 (1994), 274–274. [Cited on pages 74, 75, and 76]

Sue Sentance, Erik Barendsen, and Carsten Schulte (Eds.). 2018. *Computer Science Education. Perspectives on Teaching and Learning in School*. Bloomsbury Academic. https://doi.org/10.5040/9781350057142 [Cited on page 22]

Sue Sentance and Jane Waite. 2017. PRIMM: Exploring Pedagogical Approaches for Teaching Text-Based Programming in School. In *Proceedings of the 12th Workshop on Primary and Secondary Computing Education* (Nijmegen, Netherlands) *(WiPSCE '17)*. Association for Computing Machinery (ACM), New York, NY, USA, 113–114. https://doi.org/10.1145/3137065.3137084 [Cited on page 52]

Sue Sentance, Jane Waite, and Maria Kallia. 2019a. Teachers' Experiences of Using PRIMM to Teach Programming in School. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery (ACM), New York, NY, USA, 476–482. https://doi.org/10.1145/3287324.3287477 [Cited on pages 53 and 54]

Sue Sentance, Jane Waite, and Maria Kallia. 2019b. Teaching computer programming with PRIMM: a sociocultural perspective. *Computer Science Education* 29, 2-3 (2019), 136–176. https://doi.org/10.1080/08993408.2019.1608781 [Cited on page 52]

Judy Sheard and Dianne Hagan. 1998. Our Failing Students: A Study of a Repeat Group. *ACM SIGCSE Bulletin* 30, 3 (aug 1998), 223–227. https://doi.org/10.1145/290320.283550 [Cited on page 15]

Bruce L. Sherin. 2001. *International Journal of Computers for Mathematical Learning* 6, 1 (2001), 1–61. https://doi.org/10.1023/a:1011434026437 [Cited on page 86]

Ben Shneiderman. 1977. Teaching programming: A spiral approach to syntax and semantics. *Computers & Education* 1, 4 (Jan. 1977), 193–197. https://doi.org/10.1016/0360-1315(77)90008-2 [Cited on pages 55, 56, 110, 111, 120, and 128]

SIGCSE Special Projects. [n.d.]. Special Project Grants. Retrieved December 30, 2022 from https://sigcse.org/programs/special/ [Cited on page 215]

Xavier Simms and Hongmei Chi. 2011. Enhancing Cryptography Education via Visualization Tools. In *Proceedings of the 49th Annual Southeast Regional Conference* (Kennesaw, Georgia) *(ACM-SE '11)*. Association for Computing Machinery (ACM), New York, NY, USA, 344–345. https://doi.org/10.1145/2016039.2016139 [Cited on pages 91 and 249]

Jean Simon. 1973. *La Langue écrite de l'enfant*. Presses universitaires de France. [Cited on page 54]

Tanmay Sinha and Manu Kapur. 2019. When productive failure fails. In *Proceedings of the 41st Annual Meeting of the Cognitive Science Society (CogSci 2019): Creativity + Cognition + Compulation*. COGSCI, Montreal, Canada, 2811–2817. [Cited on pages 50, 110, and 121]

Tanmay Sinha, Manu Kapur, Robert West, Michele Catasta, Matthias Hauswirth, and Dragan Trninic. 2021. Differential benefits of explicit failure-driven and success-driven scaffolding in problem-solving prior to instruction. *Journal of Educational Psychology* 113, 3 (April 2021), 530–555. https://doi.org/10.1037/edu0000483 [Cited on pages 51, 110, 111, and 121]

Steven S. Skiena. 2020. *Combinatorial Search*. Springer International Publishing, Cham, Switzerland, 281–306. https://doi.org/10.1007/978-3-030-54256-6_9 [Cited on page 256]

James D. Slotta and Michelene T. H. Chi. 2006. Helping Students Understand Challenging Topics in Science Through Ontology Training. *Cognition and Instruction* 24, 2 (2006), 261–289. https://doi.org/10.1207/s1532690xci2402_3 arXiv:https://doi.org/10.1207/s1532690xci2402$_3$ [$Citedonpage\ 29$]

Neil Smith, Yasemin Allsop, Helen Caldwell, David Hill, Yota Dimitriadi, and Andrew Paul Csizmadia. 2015. Master Teachers in Computing: What Have We Achieved?. In *Proceedings of the Workshop in Primary and Secondary Computing Education (WiPSCE '15)*. Association for Computing Machinery (ACM), New York, NY, USA, 21–24. https://doi.org/10.1145/2818314.2818332 [Cited on page 83]

Philip Smith and Geoffrey Webb. 1995. Reinforcing a Generic Computer Model for Novice Programmers. In *Proceedings of the 12nd Australasian Society for Computers in Learning in Tertiary Education (ASCILITE '95)*. [Cited on page 27]

E. Soloway. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. 29, 9 (Sept. 1986), 850–858. https://doi.org/10.1145/6592.6594 [Cited on page 142]

Joel Sommers. 2010. Educating the next Generation of Spammers. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (Milwaukee, Wisconsin, USA) *(SIGCSE '10)*. Association for Computing Machinery (ACM), New York, NY, USA, 117–121. https://doi.org/10.1145/1734263.1734302 [Cited on pages 90 and 249]

Juha Sorva. 2012. *Visual program simulation in introductory programming education*. Doctoral thesis. School of Science. http://urn.fi/URN:ISBN:978-952-60-4626-6 [Cited on pages 29, 39, and 41]

Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *ACM Transactions on Computing Education* 13, 2, Article 8 (jul 2013), 31 pages. https://doi.org/10.1145/2483710.2483713 [Cited on pages 28 and 110]

Juha Sorva. 2018. Misconceptions and the Beginner Programmer. In *Computer Science Education. Perspectives on Teaching and Learning in School*, S. Sentance, E. Barendsen, and C. Schulte (Eds.). Bloomsbury Academic, London, United Kingdom, Chapter 13, 171–188. https://doi.org/10.5040/9781350057142.ch-013 [Cited on page 28]

J. A Spencer and R. K Jordan. 1999. Learner centred approaches in medical education. *BMJ* 318, 7193 (May 1999), 1280–1283. https://doi.org/10.1136/bmj.318.7193.1280 [Cited on page 82]

David Statter and Michal Armoni. 2020. Teaching Abstraction in Computer Science to 7th Grade Students. *ACM Trans. Comput. Educ.* 20, 1, Article 8 (Jan. 2020), 37 pages. https://doi.org/10.1145/3372143 [Cited on page 128]

Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *ACM Trans. Comput. Educ.* 13, 4, Article 19 (nov 2013), 40 pages. https://doi.org/10.1145/2534973 [Cited on page 54]

STEM Task Force. 2014. *Innovate: a blueprint for science, technology, engineering, and mathematics in California public education*. Technical Report. Californians Dedicated to Education Foundation, Dublin, CA, USA. https://www.cde.ca.gov/pd/ca/sc/documents/innovate.pdf [Cited on page 248]

Anselm L. Strauss and Juliet M. Corbin. 1998. *Basics of qualitative research: techniques and procedures for developing grounded theory*. Sage Publications, Thousand Oaks, CA, USA. [Cited on pages 186 and 187]

Lucy Suchman. 1993. Working relations of technology production and use. *Computer Supported Cooperative Work* 2, 1-2 (March 1993), 21–39. https://doi.org/10.1007/bf00749282 [Cited on pages 95 and 96]

Patrick Suppes. 1974. The Place of Theory in Educational Research. *Educational Researcher* 3, 6 (1974), 3–10. https://doi.org/10.3102/0013189X003006003 arXiv:https://doi.org/10.3102/0013189X003006003 [Cited on page 35]

John Sweller. 1988. Cognitive load during problem solving: Effects on learning. *Cognitive Science* 12, 2 (1988), 257–285. https://doi.org/10.1016/0364-0213(88)90023-7 [Cited on page 18]

John Sweller. 1994. Cognitive load theory, learning difficulty, and instructional design. *Learning and Instruction* 4, 4 (1994), 295–312. https://doi.org/10.1016/0959-4752(94)90003-5 [Cited on page 18]

John Sweller, Jeroen J. G. van Merrienboer, and Fred G. W. C. Paas. 1998. Cognitive Architecture and Instructional Design. *Educational Psychology Review* 10, 3 (1998), 251–296. https://doi.org/10.1023/a:1022193728205 [Cited on page 19]

Keith S. Taber. 2012. Constructivism as educational theory: Contingency in learning, and optimally guided instruction. In *Educational theory*, Hassaskhah Jaleh (Ed.). Nova, New York, NY, USA, 39–61. [Cited on pages 43, 44, 192, 209, and 242]

O.S. Tan, R.D. Parsons, S.L. Hinson, and D. Sardo-Brown. 2003. *Educational Psychology: A Practitioner-Researcher Approach*. Thomson Educational Publishing, Singapore. [Cited on page 48]

Cara Tang and Christian Servin. 2020. COMMUNITY COLLEGE CORNER Challenges and Opportunities during COVID: A Community College Perspective. *ACM Inroads* 11, 4 (Nov. 2020), 12–16. https://doi.org/10.1145/3429984 [Cited on page 58]

Rivka Taub, Michal Armoni, and Mordechai Ben-Ari. 2012. CS Unplugged and Middle-School Students' Views, Attitudes, and Intentions Regarding CS. *ACM Transactions on Computing Education* 12, 2, Article 8 (April 2012), 29 pages. https://doi.org/10.1145/2160547.2160551 [Cited on page 85]

Matti Tedre. 2018. The Nature of Computing as a Discipline. In *Computer Science Education. Perspectives on Teaching and Learning in School*, S. Sentance, E. Barendsen, and C. Schulte (Eds.). Bloomsbury Academic, London, United Kingdom, Chapter 2, 5–18. https://doi.org/10.5040/9781350057142.ch-002 [Cited on page 15]

Josh Tenenberg. 2019. Qualitative Methods for Computing Education. In *The Cambridge Handbook of Computing Education Research*, Sally A. Fincher and Anthony V. Robins (Eds.). Cambridge University Press, Cambridge, United Kingdom, 173–207. https://doi.org/10.1017/9781108654555.008 [Cited on pages 182, 185, 186, 188, and 193]

The Committee on European Computing Education (CECE). 2017. *Informatics Education in Europe: Are We all in the Same Boat?* Technical Report. ACM Europe & Informatics Europe. http://www.informatics-europe.org/component/phocadownload/category/10-reports.html?download=60:cece-report [Cited on page 15]

Renate Thies and Jan Vahrenhold. 2013. On Plugging "Unplugged" into CS Classes. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery (ACM), New York, NY, USA, 365–370. https://doi.org/10.1145/2445196.2445303 [Cited on page 85]

Sigmund Tobias and Thomas M. Duffy (Eds.). 2009. *Constructivist instruction: Success or failure?* Routledge. [Cited on pages 42 and 242]

Ivan Tomek, Tomasz Muldner, and Saleem Khan. 1985. PMS—A program to make learning Pascal easier. *Computers & Education* 9, 4 (1985), 205–211. https://doi.org/10.1016/0360-1315(85)90009-0 [Cited on page 55]

David Touretzky, Fred Martin, Deborah Seehorn, Cynthia Breazeal, and Tess Posner. 2019. Special Session: AI for K-12 Guidelines Initiative. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery (ACM), New York, NY, USA, 492–493. https://doi.org/10.1145/3287324.3287525 [Cited on page 217]

Ari Tuhkala. 2021. A systematic literature review of participatory design studies involving teachers. *European Journal of Education* 56, 4 (Dec. 2021), 641–659. https://doi.org/10.1111/ejed.12471 [Cited on page 59]

Claude F. Turner, Blair Taylor, and Siddharth Kaza. 2011. Security in Computer Literacy: A Model for Design, Dissemination, and Assessment. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (Dallas, TX, USA) *(SIGCSE '11)*. Association for Computing Machinery (ACM), New York, NY, USA, 15–20. https://doi.org/10.1145/1953163.1953174 [Cited on pages 90 and 249]

Raymond Turner. 2021. Computational Abstraction. *Entropy* 23, 2 (2021). https://doi.org/10.3390/e23020213 [Cited on page 31]

UChicago STEM Education. n.d. Scratch Encore. https://www.canonlab.org/scratch-encore [Cited on page 198]

UK Department of Education. 2013. *National curriculum in England: computing programmes of study*. https://www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study [Cited on pages 89 and 216]

U.S. Bureau of Labor Statistics. 2015. Table 4. employment by Major Occupational Group, 2014 and projected 2024. Retrieved October 31, 2022 from https://www.bls.gov/news.release/ecopro.t04.htm [Cited on page 14]

Ian Utting, Allison Elliott Tew, Mike McCracken, Lynda Thomas, Dennis Bouvier, Roger Frye, James Paterson, Michael Caspersen, Yifat Ben-David Kolikant, Juha Sorva, and Tadeusz Wilusz. 2013. A Fresh Look at Novice Programmers' Performance and Their Teachers' Expectations. In *Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education-Working Group Reports* (Canterbury, England, United Kingdom) *(ITiCSE -WGR '13)*. Association for Computing Machinery (ACM), New York, NY, USA, 15–32. https://doi.org/10.1145/2543882.2543884 [Cited on page 16]

Dirk van der Linden, Awais Rashid, Emma Williams, and Bogdan Warinschi. 2018. Safe Cryptography for All: Towards Visual Metaphor Driven Cryptography Building Blocks. In *Proceedings of the 1st International Workshop on Security Awareness from Design to Deployment* (Gothenburg, Sweden) *(SEAD '18)*. Association for Computing Machinery (ACM), New York, NY, USA, 41–44. https://doi.org/10.1145/3194707.3194709 [Cited on page 91]

Kurt VanLehn, Stephanie Siler, R. Charles Murray, Takashi Yamauchi, and William Baggett. 2003. Why Do Only Some Events Cause Learning During Human Tutoring? *Cognition and Instruction* 21 (09 2003), 209–249. https://doi.org/10.1207/S1532690XCI2103_01 [Cited on pages 49 and 122]

Marianne Verhallen and Simon Verhallen. 1994. *Woorden leren, woorden onderwijzen*. CPS. [Cited on page 55]

Eelco Visser. 2015. Understanding software through linguistic abstraction. *Science of Computer Programming* 97 (2015), 11–16. https://doi.org/10.1016/j.scico.2013.12.001 Special Issue on New Ideas and Emerging Results in Understanding Software. [Cited on page 127]

Willemien Visser. 1987. Strategies in Programming Programmable Controllers: A Field Study on a Professional Programmer. In *Empirical Studies of Programmers: Second workshop (ESP2)*, G. M. Olson, S. Sheppard, and E. Soloway (Eds.). Ablex, 217–230. https://inria.hal.science/hal-00641376 [Cited on page 30]

Joke Voogt, Petra Fisser, Jon Good, Punya Mishra, and Aman Yadav. 2015. Computational thinking in compulsory education: Towards an agenda for research and practice. *Education and Information Technologies* 20, 4 (Dec. 2015), 715–728. https://doi.org/10.1007/s10639-015-9412-6 [Cited on pages 67 and 243]

Valdemar Švábenský, Jan Vykopal, and Pavel Čeleda. 2020. What Are Cybersecurity Education Papers About? A Systematic Literature Review of SIGCSE and ITiCSE Conferences. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery (ACM), New York, NY, USA, 2–8. https://doi.org/10.1145/3328778.3366816 [Cited on pages 90 and 249]

Lev S. Vygotsky. 1978. *Mind in Society*. Harvard University Press, Cambridge, MA, USA. [Cited on pages 41 and 43]

Jacques Wainer and Eduardo C. Xavier. 2018. A Controlled Experiment on Python vs C for an Introductory Programming Course: Students' Outcomes. *ACM Trans. Comput. Educ.* 18, 3, Article 12 (aug 2018), 16 pages. https://doi.org/10.1145/3152894 [Cited on page 54]

Christopher Watson and Frederick W.B. Li. 2014. Failure Rates in Introductory Programming Revisited. In *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education* (Uppsala, Sweden) *(ITiCSE '14)*. Association for Computing Machinery (ACM), New York, NY, USA, 39–44. https://doi.org/10.1145/2591708.2591749 [Cited on pages 15 and 21]

Bruce F. Webster. 1996. The real software crisis: The shortage of top-notch programmers threatens to become the limiting factor in software development. *Byte Magazine* (1996). https://brucefwebster.com/2013/09/13/the-real-software-crisis-byte-magazine-january-1996/ [Cited on page 16]

Gordon Wells. 1999. *Dialogic inquiry: Towards a socio-cultural practice and theory of education*. Cambridge University Press. [Cited on page 85]

Jacqueline L. Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P. K. Ajith Kumar, and Christine Prasad. 2006. An Australasian Study of Reading and Comprehension Skills in Novice Programmers, Using the Bloom and SOLO Taxonomies. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52* (Hobart, Australia) *(ACE '06)*. Australian Computer Society, Inc., AUS, 243–252. [Cited on page 25]

Wikipedia contributors. 2022. *Diffie–Hellman key exchange — Wikipedia, The Free Encyclopedia*. Retrieved December 23, 2022 from https://en.wikipedia.org/wiki/DiffieHellman_key_exchange [Cited on page 230]

Jeannette M. Wing. 2006. Computational Thinking. *Commun. ACM* 49, 3 (mar 2006), 33–35. https://doi.org/10.1145/1118178.1118215 [Cited on pages 66 and 230]

Jeannette M. Wing. 2008. Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 366, 1881 (2008), 3717–3725. https://doi.org/10.1098/rsta.2008.0118 [Cited on page 230]

J. Winterton, F.D.L. Deist, E. Stringfellow, and European Centre for the Development of Vocational Training. 2006. *Typology of Knowledge, Skills and Competences: Clarification of the Concept and Prototype*. Office for Official Publications of the European Communities. [Cited on page 25]

David Wood, Jerome S. Bruner, and Gail Ross. 1976. The role of tutoring in problem solving. *Journal of Child Psychology and Psychiatry* 17, 2 (April 1976), 89–100. https://doi.org/10.1111/j.1469-7610.1976.tb00381.x [Cited on pages 43 and 85]

Aman Yadav and Ulf Dalvad Berthelsen. 2021. *Computational Thinking in Education*. Routledge. https://doi.org/10.4324/9781003102991 [Cited on pages 86, 88, 229, and 243]

Aharon Yadin. 2013. Using Unique Assignments for Reducing the Bimodal Grade Distribution. *ACM Inroads* 4, 1 (mar 2013), 38–42. https://doi.org/10.1145/2432596.2432612 [Cited on page 16]

Bernard Yett, Nicole Hutchins, Gordon Stein, Hamid Zare, Caitlin Snyder, Gautam Biswas, Mary Metelko, and Ákos Lédeczi. 2020. A Hands-On Cybersecurity Curriculum Using a Robotics Platform. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery (ACM), New York, NY, USA, 1040–1046. https://doi.org/10.1145/3328778.3366878 [Cited on page 91]

Maximilian Zinkus, Oliver Curry, Marina Moore, Zachary Peterson, and Zoë J. Wood. 2019. Fakesbook: A Social Networking Platform for Teaching Security and Privacy Concepts to Secondary School Students. In *Proc. of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery (ACM), New York, NY, USA, 892–898. https://doi.org/10.1145/3287324.3287486 [Cited on page 91]

Acknowledgments

Ringrazio innanzitutto il professor Michael Lodi. Con grande generosità e fiducia, Michael mi ha da subito messo a disposizione le sue notevoli competenze scientifiche e metodologiche, incoraggiandomi e consentendomi di crescere come ricercatore e professionista. Nonostante la disparità di valore ed esperienza, mi ha sempre trattato da pari, permettendomi di esprimermi senza imbarazzi e di contribuire al meglio al nostro lavoro di ricerca. Mi ritengo davvero fortunato di aver trovato in lui un amico.

Ringrazio il mio supervisore, il professor Simone Martini, scienzato, educatore e persona di grande valore, il cui esempio è per me sprone e la cui conoscenza motivo d'orgoglio. La sua presenza discreta ma costante, la sua chiarezza e correttezza sono state un supporto fondamentale per affrontare questo percorso e ingredienti di crescita professionale e umana.

Thanks also to Professors Valentina Dagienė and Francisco Castro for the time they devoted to a thorough and generous review of my thesis, leaving me with rich indications that empowered me to improve it.
To Professor Dagienė, special thanks for the hospitality and esteem with which she welcomed me to Lithuania and allowed me to participate in a valuable and beautiful doctoral consortium.

Ringrazio i professori Rebecca Montanari e Gianluigi Zavattaro, per la fiducia e la stima e per le loro indicazioni misurate ma decisive nei momenti chiave di questo percorso che mi hanno consentito di migliorare la qualità del mio lavoro.
Grazie al professor Alessandro Ricci, fonte di ispirazione professionale, ma anche persona eccezionale e amico.
Grazie al mio ex Dirigente Scolastico, ingegner Salvatore Grillo, che ha sostenuto la mia avventura di dottorato, facilitandone la burocrazia e incoraggiandomi.

Grazie alla mia famiglia.
Grazie all'amore costante della nonna Maria, alla fervida stima del nonno Gino e alla forza sicura di mia sorella Silvia.
Grazie tantissimo ai miei genitori per il loro supporto senza fine... grazie all'esuberante sprone della mamma Mara e alla fiducia attenta del babbo Claudio.
Un ringraziamento speciale a Sofia, compagna che con amore e pazienza mi ha accompagnato dappertutto, sacrificando se stessa e donandomi tesori che non sapevo esistessero.
Grazie ad Alice, amica e sorella, che non mi ha mai fatto mancare ascolto e comprensione, supporto e fiducia.