Alma Mater Studiorum
Università di Bologna

**Dottorato di Ricerca in
Matematica**

Ciclo XXV

Settore Concorsuale di afferenza: 01/A2
Settore Scientifico Disciplinare: MAT/02

Permutation classes, sorting algorithms, and lattice paths

Tesi di Dottorato presentata da Luca Ferrari

Coordinatore Dottorato
Prof. Alberto Parmeggiani

Relatore
Prof. Marilena Barnabei

Esame Finale anno 2013

Contents

Introduction

A permutation σ is said to *avoid* a pattern τ if it does not contain any subsequence which is order-isomorphic to τ. The set $Av(T)$ of permutations avoiding a set of patterns T is a *permutation class*, namely, an order ideal of the poset $(\Sigma, \preccurlyeq)$, where Σ is the set of permutations of any length and $\preccurlyeq$ is the pattern containment relation.

Permutation classes are closely related to the action of some sorting procedures, which are realized by using four particular devices. The *deque* is a double-ended linear data structure where the elements are inserted and removed through two gates, placed at both ends of the deque. Four input (I and $\bar{I}$) and output (O and $\bar{O}$) operations are allowed: I and O act on the one end of the deque, while $\bar{I}$ and $\bar{O}$ act on the other. The other three devices can be obtained by appropriate restrictions on the four input/output operations. In the *stack*, for example, the elements go in and out through only one of the gates. In the *input-restricted deque*, all the operations are allowed except for $\bar{I}$, while, conversely, in the *output-restricted deque* the only forbidden operation is $\bar{O}$.

Suppose to take the identity permutation $id = 12 \dots n$ as the input permutation of a device $\mathbb{X}$. We say that a permutation σ is *computed* by $\mathbb{X}$ if there exists a sequence of input/output operations, allowed by $\mathbb{X}$, that transforms id into σ. For instance, the permutation $\sigma = 312$ can be computed by the sequences $III\bar{O}\bar{O}O$ and $I\bar{I}IOOO$, while it is not possible to obtain σ by making use only of the operations I and O. Therefore, σ is computed by all the variants of the deque, but not by the stack.

Donald Knuth, in the first volume of his celebrated book *The art of Computer Programming* [13], observed that the permutations which can be compu-

ted by these data structures can be characterized in terms of pattern avoidance.

After Knuth's results, Tarjan [21] and Pratt [17] studied analogous problems in more complex contexts. In recent years the topic was reopened several times, while often in terms of *sortable permutations* rather than *computable permutations*. A permutation σ is *sorted* by a device $\mathbb{X}$ if there exists a sequence of input/output operations of $\mathbb{X}$ that turns σ into the identity permutation. It is not difficult to show that the permutations that can be sorted by a fixed device are exactly the inverses of those that can be computed.

The idea to sort permutations through such devices suggests to look for a possible deterministic procedure which decides if there exists a sequence of input/output operations which is able to convert a given permutation σ into id. Of course, such a procedure becomes even more interesting if it is efficient in terms of computational complexity: a *brute force* approach is of no interest, and practically inapplicable, in this case. For this reason, we look for a procedure that decides if a given permutation σ is $\mathbb{X}$–sortable or not without testing all the possible $\mathbb{X}$–sequences.

In the procedures that we describe in the thesis, we require that the answer about the sortability of the input permutation σ must be given by just scanning σ and simultaneously deciding, in constant time, which input/output operation has to be performed in order to turn σ into the permutation id. Under this condition, the decision procedures answer the question in time $O(|\sigma|)$, and, what is more, they give a constructive method that finds, for the $\mathbb{X}$–sortable permutations, one of the possible $\mathbb{X}$–sorting sequences for σ. In this sense, we are allowed to call them *sorting procedures*.

Under the previous conditions and other minor rules, we show that there exists an unique way to implement such a procedure for the stack and the restricted deques. Moreover, these procedures can be applied not only on permutations but even on more general input sequences of totally ordered symbols. The stack sorting procedure is very well-known in literature, and often taken for granted. Conversely, it is quite difficult to find a detailed analysis of the procedures for the restricted deques. As regards the deque, we show that it is not possible to realize a procedure that follows the previous restrictions and that is able to sort all the $\mathbb{D}$–sortable permutations.

In chapter 4 we discuss the use of the sorting procedures in some different contexts. It is not difficult to show that, for any given input permutation, we can always obtain the identity permutation through a suitable number of iterations of a sorting procedure. From this point of view, the sorting procedures can be used as base steps of new sorting algorithms. We can also create hybrid algorithms by blending two sorting procedures which are associated to different devices, or mixing a sorting procedure with the base step of a sorting algorithm.

In chapter 4 we analyze in detail the joint action of the stack sorting procedure (S) and the base step of the well-known *bubblesort* algorithm (B), together with their *dual* versions. Despite being inefficient and, therefore, poorly attractive for practical purposes, the bubblesort presents some very interesting features from a more theoretical point of view.

In a single step of bubblesort two consecutive elements of the input permutation are swapped if the smaller follows the greater. Obviously, a single iteration of B is not sufficient, in general, to sort the input permutation. The set of $B-$sortable permutations is a permutation class [1], as the set of $\mathbb{X}-$sortable permutations.

We introduce the dual procedures $\tilde{B}$ and $\tilde{S}$, which are obtained from the original B and S through the action of the reverse-complement operator ρ: $\tilde{B} = \rho \circ B \circ \rho$ and $\tilde{S} = \rho \circ S \circ \rho$. As one might expect, if we mix together some steps of an algorithm and some steps of a completely different one, the action of the resulting hybrid algorithm depends, in general, on the order we use to perform the different steps. Despite this, we prove that - quite surprisingly - the output of an algorithm consisting of some steps S and some steps $\tilde{B}$ depends only on the number of steps of each type, and not on their relative order. Moreover, the same holds for B and $\tilde{B}$, and for B and $\tilde{S}$.

In order to prove these commutation properties, we decompose the procedures B and $\tilde{B}$ into some simpler sub procedures, called *local sorting operators*. Furthermore, we show that the permutations that can be sorted by a fixed number of iterations of B and $\tilde{B}$ can be expressed, once again, in terms of pattern avoidance. More precisely, the basis of the associated permutation class consists of patterns which are *inflations* of the permutation 21.

In the final chapter we give an alternative proof of some enumerative results for the classes of $\mathbb{X}$–sortable permutations, in particular for the two restricted deques. It is well-known that the number of permutations of length n that can be sorted through a restricted deque corresponds to the number of *Schröder paths* of length $2(n-1)$: we refer to Knuth [13] for an analytical proof of this. In the thesis, we show how the $\mathbb{X}$–sorting procedures yield a bijection between sortable permutations and Schröder paths.

Chapter 1

Permutation classes

1.1 Permutations and pattern containment

Definition 1.1.1. *A* permutation of length n *is a one-to-one correspondence from the set* $\{1, 2, \ldots, n\}$ *to itself.*

We will denote by Σ_n the set of all permutations of length n, and by

$$\Sigma = \bigcup_{n=0}^{\infty} \Sigma_n$$

the set of all permutations of any length.

Permutations will be denoted by making use of the one-line representation

$$\sigma = \sigma_1 \sigma_2 \ldots \sigma_n,$$

where σ_i stands for $\sigma(i)$, the image of $i \in \{1, 2, \ldots, n\}$ under σ. We will use the symbol $|\sigma|$ to denote the length of σ.

By definition, in the one-line representation of a permutation we have all the natural numbers from 1 to n, without repetitions. These restrictions are removed when talking about *words*.

Definition 1.1.2. *A* word of length n over the alphabet $\mathcal{A}$ *is a correspondence from* $\{1, 2, \ldots, n\}$ *to* $\mathcal{A}$.

The one-line representation of a word is analogous to the one already defined for permutations.

Definition 1.1.3. *We say that α is a* subsequence *of a permutation σ if there exist indices $1 \leqslant i_1 \leqslant \cdots \leqslant i_t \leqslant n$ such that $\alpha = \sigma_{i_1} \ldots \sigma_{i_t}$.*

In other terms, a subsequence α of a permutation is a word whose one-line notation can be obtained from the one-line notation of σ by choosing some of its elements.

Definition 1.1.4. *We say that two words α and β, of length n, are* order-isomorphic *if, for every $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant n$, we have*

$$\alpha_i < \alpha_j \iff \beta_i < \beta_j \ .$$

Definition 1.1.5. *A permutation $\sigma \in \Sigma_n$ is said to* contain the pattern $\tau \in \Sigma_t$ *if there exist a subsequence α of σ which is order-isomorphic to τ. In this case, we write $\tau \preccurlyeq \sigma$.*

We can represent a permutation $\sigma = \sigma_1\sigma_2\ldots\sigma_n$ by the usual graphical representation: for every image σ_i, we draw a circle in the box at position (i, σ_i) of a square grid (see figure 1.1).

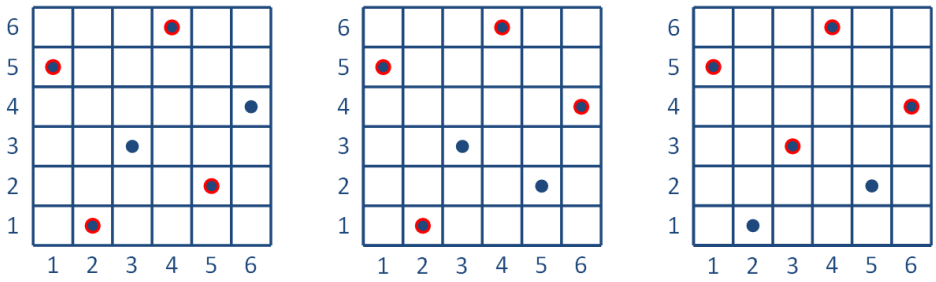


Figure 1.1. *The permutation $\sigma = 513624$ has three occurrences of the pattern $\tau = 3142$: the subsequences 5162, 5164 and 5364 (circled in red).*

Proposition 1.1.6. *The pattern containment relation $\preccurlyeq$ is a partial order relation on Σ.*

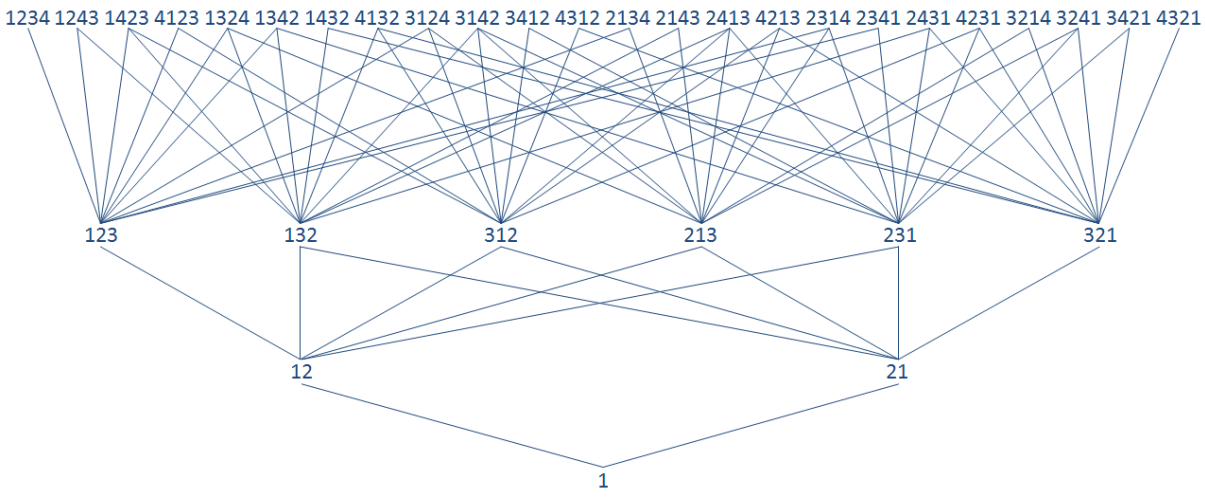


Figure 1.2. *The Hasse diagram of the poset* $(\Sigma, \preccurlyeq)$*, drawn for permutations of length* $n \leqslant 4$*.*

1.2 Pattern avoidance and permutation classes

Definition 1.2.1. *A permutation* σ *is said to* avoid *a pattern* τ *if* σ *does not contain any subsequence that is order-isomorphic to* τ *(*$\tau \not\preccurlyeq \sigma$*).*

The set of permutations of Σ which avoid a pattern τ will be denoted by the symbol

$$Av(\tau).$$

More generally, if T is a set of patterns, we will denote by

$$Av(T) = \bigcap_{\tau \in T} Av(\tau)$$

the set of permutations of Σ that simultaneously avoid all the patterns in T. In particular, we will use the symbol $Av_n(T)$ to denote the set of permutations of length n of $Av(T)$. In literature, the symbols $S_n(T)$ and $\Sigma_n(T)$ are often used instead of $Av_n(T)$.

Definition 1.2.2. *A* permutation class *is a set* $\mathcal{C}$ *of permutations such that*

$$\forall\, \sigma \in \mathcal{C},\ \tau \preccurlyeq \sigma \ \Rightarrow\ \tau \in \mathcal{C}.$$

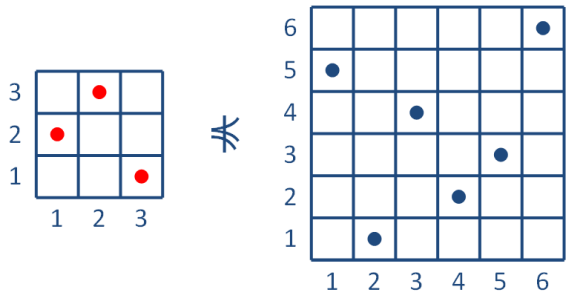


Figure 1.3. *The permutation* $\sigma = 514236$ *does not contain any subsequence which is order-isomorphic to* $\tau = 231$. *Hence,* σ avoids 231.

In other terms, a set of permutations is a permutation class if it is an order ideal of the poset $(\Sigma, \preccurlyeq)$.

Observe that, if σ avoids a set of patterns T, then the same holds for all the patterns of σ. This proves the following proposition.

Proposition 1.2.3. *For every set of patterns* T, *the set* $Av(T)$ *is a permutation class.*

It is not difficult to show that even the converse holds. As we already observed, a permutation class $\mathcal{C}$ is an ideal of Σ, and hence its complementary $\mathcal{F} = \Sigma \setminus \mathcal{C}$ is a filter. Therefore, denoting by T the set of minimal elements of $\mathcal{F}$, it is immediately proven that $\mathcal{C} = Av(T)$. This allows us to state the following proposition.

Proposition 1.2.4. *Every permutation class* $\mathcal{C}$ *can be represented as*

$$\mathcal{C} = Av(T),$$

where T *is the set of minimal elements of* $\Sigma \setminus \mathcal{C}$. *The set* T *will be called the* basis *of* $\mathcal{C}$.

From now on, when referring to a permutation class $Av(T)$ we always suppose that T is the set of minimal elements of the complementary filter $\mathcal{F} = \Sigma \setminus \mathcal{C}$.

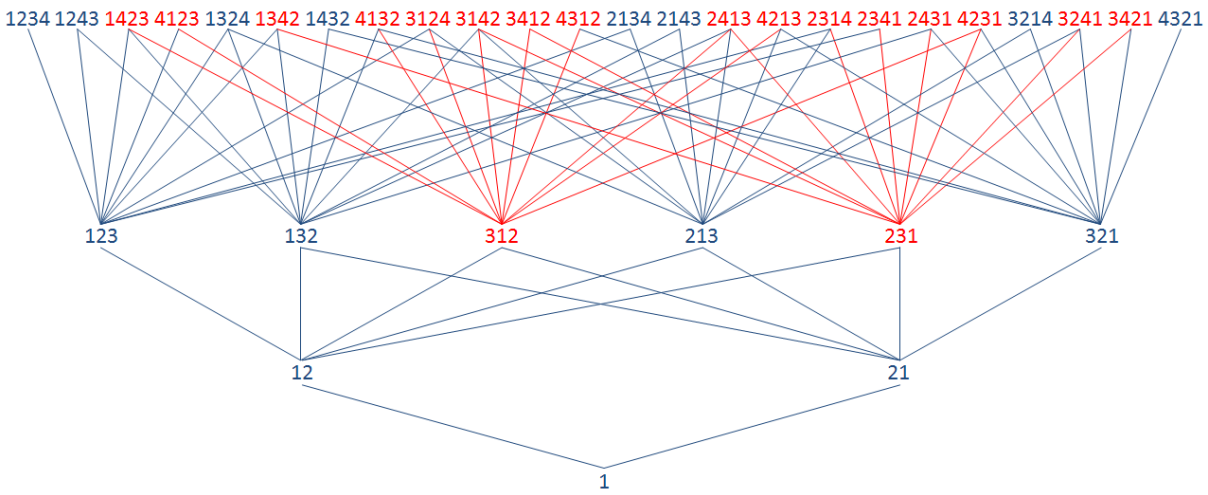


Figure 1.4. *The first elements of the permutation class* $\mathcal{C} = Av(231, 312)$ *(in blue) and the elements of the complementary filter* $\mathcal{F} = \Sigma \setminus \mathcal{C}$ *(in red). All the permutations in* $\mathcal{F}$ *contain at least one between the patterns* 231 *and* 312.

1.3 Symmetries

There are three fundamental symmetries on permutations: *reverse*, *complement* and *inverse*.

Definition 1.3.1. *We call* reverse, complement *and* inverse, *respectively, the transformations*

$$r\colon \begin{array}{ccc} \Sigma_n & \longrightarrow & \Sigma_n \\ \sigma & \longmapsto & \sigma^r \end{array} \qquad c\colon \begin{array}{ccc} \Sigma_n & \longrightarrow & \Sigma_n \\ \sigma & \longmapsto & \sigma^c \end{array} \qquad -1\colon \begin{array}{ccc} \Sigma_n & \longrightarrow & \Sigma_n \\ \sigma & \longmapsto & \sigma^{-1} \end{array},$$

where σ^{-1} *is the inverse of* σ*, while* σ^r *and* σ^c *are the permutations whose* $i-th$ *element is defined as*

$$(\sigma^r)_i = \sigma_{n+1-i}$$

and

$$(\sigma^c)_i = n + 1 - \sigma_i.$$

Observe that, denoting by id^r the reverse of the identity permutation $id^r = (n)\,(n-1)\ldots 21$, the permutations σ^r and σ^c can be defined as

$$\sigma^r = \sigma \circ id^r \qquad \text{and} \qquad \sigma^c = id^r \circ \sigma. \tag{1.1}$$

In the graphical representation, the reverse, complement and inverse act, respectively, as a horizontal, vertical and diagonal symmetry (see figure 1.5). Moreover, these transformations generate the group of symmetries of the square: the *dihedral group*

$$D_4 = \langle r, c, -1 \rangle = \{id, r, c, -1, c \circ r, -1 \circ r, -1 \circ c, -1 \circ c \circ r\}.$$

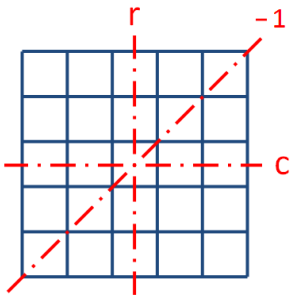


Figure 1.5. *The 3 generators of the group D_4.*

The action of D_4 partitions Σ in many equivalence classes, that we call *symmetry classes*.

Definition 1.3.2. *The* symmetry class *of a permutation σ, denoted by $[\sigma]$, is the set of permutations that can be obtained from σ through the action of D_4:*

$$[\sigma] = \{\sigma^\phi \,|\, \phi \in D_4\}.$$

Definition 1.3.3. *We say that two permutations σ_1 and σ_2 are* symmetry-equivalent *($\sigma_1 \cong \sigma_2$) if they belong to the same symmetry class.*

Definition 1.3.4. *Let T be a set of permutations, and denote by*

$$T^\phi = \{\sigma^\phi \,|\, \sigma \in T\}$$

Symmetry class	*Permutations*
$[123]$	123 321
$[132]$	132 213 231 312

Table 1.1. *The permutations of Σ_3 split into 2 symmetry classes.*

the set of permutations obtained through the action of a fixed element $\phi \in D_4$ on the permutations of T. The symmetry class *of T, denoted by $[T]$, is the collection of sets of permutations that can be obtained from T through the action of D_4:*

$$[T] = \{T^\phi \mid \phi \in D_4\}.$$

Definition 1.3.5. *We say that two sets of permutations T_1 and T_2 are* symmetry-equivalent *($T_1 \cong T_2$) if they belong to the same symmetry class.*

Example 1.3.6. If $T = \{123, 231\}$, then

$$[T] = [123, 231] = \{\{123, 231\}, \{123, 312\}, \{132, 321\}, \{213, 321\}\},$$

where the symmetry-equivalent sets of permutations are $T = ((T^r)^c)^{-1} = \{123, 231\}$, $T^{-1} = (T^r)^c = \{123, 312\}$, $T^r = (T^r)^{-1} = \{132, 321\}$ and $T^c = (T^c)^{-1} = \{213, 321\}$. For the other symmetry classes of pairs of permutations of Σ_3, see table 1.2.

Observe that, if ϕ is any element of D_4, then for all $\sigma, \tau \in \Sigma$

$$\tau \preccurlyeq \sigma \iff \tau^\phi \preccurlyeq \sigma^\phi.$$

Hence, for any τ, $\sigma \in \Sigma$ and for any $\phi \in D_4$,

$$\sigma \in Av(\tau) \iff \sigma^\phi \in Av(\tau^\phi).$$

By this relation, the following proposition is immediately proven.

Symmetry class	*Pairs of permutations*
[123,132]	{123,132} {123,213} {321,231} {321,312}
[123,231]	{123,231} {123,312} {132,321} {213,321}
[123,321]	{123,321}
[132,213]	{132,213} {231,312}
[132,231]	{132,231} {132,312} {213,231} {213,312}

Table 1.2. *There are* 15 *possible pairs of permutations of* Σ_3*, partitioned into* 5 *symmetry classes.*

Proposition 1.3.7. *Let* T *be a set of patterns. Then, for every* $\phi \in D_4$*,*

$$Av(T)^\phi = Av(T^\phi).$$

By the previous proposition we can prove the following one.

Proposition 1.3.8. *Two permutation classes* $Av(T_1)$ *and* $Av(T_2)$ *are symmetry-equivalent if and only if their bases are symmetry-equivalent:*

$$Av(T_1) \cong Av(T_2) \iff T_1 \cong T_2.$$

Proof. By definition, the permutation classes $Av(T_1)$ and $Av(T_2)$ are symmetry-equivalent if and only if there exists $\phi \in D_4$ such that $Av(T_1) = Av(T_2)^\phi$. By proposition 1.3.7 we have that $Av(T_2)^\phi = Av(T_2^\phi)$, and hence $Av(T_1) = Av(T_2^\phi)$. Since the basis of a permutation class is unique, this is sufficient to state that $T_1 = T_2^\phi$, and hence T_1 and T_2 are symmetry-equivalent. The proof of the converse is analogous. □

1.4 Cardinality

Among the research directions related to permutation classes, one of the most investigated in literature is the enumeration problem. For a given set of

patterns T, the goal is to find the number of permutations of $Av(T)$ of length n, namely, the distribution of

$$|Av_n(T)|,$$

for every $n \in \mathbb{N}$.

Actually, proposition 1.4.2 guarantees that, in general, it is not necessary to consider *all* the possible patterns: without loss of generality, we can focus only on a subset of them.

Definition 1.4.1. *We say that two permutation classes $Av(T_1)$ and $Av(T_2)$ are* equidistributed *if*

$$|Av_n(T_1)| = |Av_n(T_2)|, \qquad \forall\, n \in \mathbb{N}.$$

Two equidistributed permutation classes are also called Wilf-equivalent.

Proposition 1.4.2. *If two bases of patterns T_1 and T_2 are symmetry-equivalent, the associated permutation classes are equidistributed:*

$$T_1 \cong T_2 \quad \Longrightarrow \quad |Av_n(T_1)| = |Av_n(T_2)|, \quad \forall\, n \in \mathbb{N}.$$

Proof. By proposition 1.3.8, $T_1 \cong T_2$ implies that $Av(T_1) \cong Av(T_2)$ and hence $Av(T_1) = Av(T_2)^\phi$, for a suitable $\phi \in D_4$. Therefore, the permutations of length n of $Av(T_1)$ are obtained by the permutations of length n of $Av(T_2)$ by the action of one of the symmetries of D_4, and hence they are equinumerous. □

The previous proposition implies that it is sufficient to find the cardinality of a permutation class $Av_n(T)$ to obtain the cardinality of all the other symmetry-equivalent permutation classes.

Example 1.4.3. If we want to study all the possible cardinalities $|Av_n(T)|$, when T is a pair of permutations of Σ_3 (see table 1.2), we can focus on the cardinalities of only 5 permutation classes, chosen so that their bases belong to pairwise distinct symmetry classes.

We remark that, in general, the converse of proposition 1.4.2 does not hold: it is not difficult to show that there exist non-symmetry-equivalent permutation classes which are equidistributed. This is the case, for example, of the

permutation classes $Av(123)$ and $Av(132)$, that we will discuss in the following section.

1.5 Enumerative results

Several methods have been used in literature to enumerate permutation classes (see e.g. [12] for an updated overview).

When looking for an explicit formula for the enumeration of $|Av_n(T)|$, the solution is trivial if T contains at least one pattern of length $t \leqslant 2$.

The first nontrivial result on this topic was found by MacMahon [15], who proved that the distribution of the permutation class $Av(123)$ is the sequence $(C_n)_n$ of *Catalan numbers*

$$C_n = \frac{1}{n+1}\binom{2n}{n}. \tag{1.2}$$

Many years later, Knuth [13] proved that even $Av(312)$ is enumerated by the Catalan numbers. In his proof, Knuth observed that the permutations avoiding 312 are exactly those that can be obtained through a *stack* starting from the identity permutation. This kind of relation can be extended to other permutation classes and data structures: we will discuss them in the next chapters.

The results obtained by MacMahon and Knuth proved that the two permutation classes $Av(123)$ and $Av(312)$ are Wilf-equivalent, although they are not symmetry equivalent:

$$|Av_n(123)| = |Av_n(132)|, \qquad \forall\, n \in \mathbb{N}.$$

Since every other pattern of length 3 is symmetry-equivalent to one between 123 or 312, we can state the following theorem.

Theorem 1.5.1. *For all $\tau \in \Sigma_3$, the number of permutations of length n avoiding τ is the $n-th$ Catalan number:*

$$|Av_n(\tau)| = C_n.$$

The equidistribution of the permutation classes $Av(\tau)$, $\tau \in \Sigma_3$, has been proved in many other different ways. Several authors used bijections between two non-symmetry-equivalent permutation classes. Most of these bijections involve *Dyck paths*, that we will introduce in chapter 5. We refer to [9] for a detailed overview on these bijections.

The cardinality of $Av(T)$, when T is a subset of Σ_3 with two or more patterns, was studied by Simion and Schmidt [20], who completed the enumeration of $Av_n(T)$ for all $T \subset \Sigma_3$.

The situation becomes much more complicated when considering permutations avoiding patterns of length 4, or more. The permutation classes $Av(\tau)$, $\tau \in \Sigma_4$, are partitioned into 7 symmetry classes and 3 Wilf classes: $Av(1234)$, $Av(1324)$ and $Av(1342)$. Gessel [11] and Bóna [4] found the enumeration of the first and the third class, respectively, while the same problem on $Av(1324)$ still remains unsolved.

Many other enumerative results have been found for permutation classes avoiding two patterns τ_1 and τ_2. When $\tau_1 \in \Sigma_3$ and $\tau_2 \in \Sigma_4$, all the Wilf classes that arise have been enumerated. When both $\tau_1, \tau_2 \in \Sigma_4$, many Wilf classes have been enumerated, but several of them are currently under investigation. In the following, we will focus in particular on the classes $Av(3241, 4231)$ and $Av(2431, 4231)$, which are strictly related to the sorting procedures that we will present in chapter 3.

1.6 Asymptotics

It is very difficult, in general, to enumerate the sequence $(Av_n(T))_n$ when the basis T contains patterns of length 5, or more.

Rather than finding the explicit enumeration, it is often convenient to look for some information on the *asymptotic behaviour* of the sequence $(Av_n(T))_n$. A very strong result in this direction was found by Marcus and Tardos in 2004 [16], who proved a famous conjecture due to Stanley and Wilf.

Theorem 1.6.1. (Marcus-Tardos Theorem, Stanley-Wilf Conjecture) *For any permutation class $Av(\tau)$, there exists a constant c_τ such that*

$$|Av_n(\tau)| \leqslant {c_\tau}^n.$$

Observe that the number of permutations avoiding τ is asymptotically irrelevant, if compared to the total number of permutations. In fact, by the Marcus-Tardos Theorem,

$$\frac{|Av_n(\tau)|}{|\Sigma_n|} \leqslant \frac{{c_\tau}^n}{n!} \xrightarrow{n\to+\infty} 0.$$

The constant c_τ found by Marcus and Tardos is

$$c_\tau = 15^{2|\tau|^4\binom{|\tau|^2}{|\tau|}}.$$

In the last years, some sharper bounds have been found by Bóna [7] and Cibulka [8], but further improvements may be possible.

Chapter 2

Data structures and computable permutations

2.1 Data structures

In the first chapter we showed that each permutation class can be represented in terms of pattern avoidance. In this chapter we show that permutation classes are also closely connected with the action of some particular data structures, that we introduce below.

A *deque* ($\mathbb{D}$) is a device which is able to store a set of elements, sequentially arranged. The name, acronym of *Double Ended QUEue*, is due to the fact that elements move in and out through two gates, placed at both ends of $\mathbb{D}$.

Figure 2.1. *A deque.*

At any time, we can decide whether to put an element into the deque, or to take out another one. In the first case, we take the leftmost element of the input sequence and we put it into the deque through one of the gates. If we choose the left gate, the input element becomes the leftmost element of $\mathbb{D}$; if

we choose the right gate, it becomes the rightmost element of $\mathbb{D}$. When we take out an element from $\mathbb{D}$, we can choose either the leftmost or the rightmost one, and then put it in the rightmost position of the output sequence.

We denote by I the insertion of the input element into $\mathbb{D}$ through the left gate, and by O the extraction from $\mathbb{D}$ of the leftmost element. Similarly, we denote by $\bar{I}$ and $\bar{O}$ the same operations on the right gate (see figure 2.2).

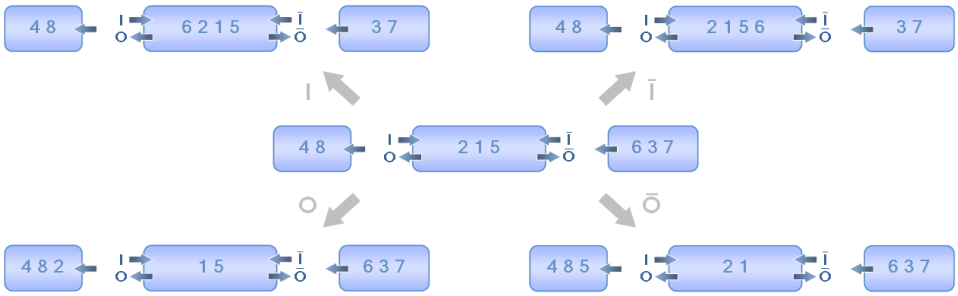


Figure 2.2. *The action of the four possible input/output operations.*

If we close one or both gates in the input or output direction, we obtain four other devices, which are essentially different from the original deque.

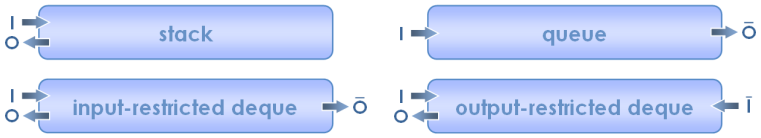


Figure 2.3. *By restricting the input and output operations we obtain four variants of the deque.*

Here is a short description of each of them.

- *Queue* ($\mathbb{Q}$): elements enter through one gate and exit through the opposite one; it is a *First In, First Out* (*FIFO*) data structure.

- *Stack* ($\mathbb{S}$): elements are obliged to enter and exit through the same gate; it is a *Last In, First Out* (*LIFO*) data structure.
- *Input-restricted deque* ($\mathbb{D}^{ir}$): elements enter through only one gate, while the exit is allowed through both ones.
- *Output-restricted deque* ($\mathbb{D}^{or}$): elements can enter through both gates, while the exit is allowed through only one gate.

Device		$\boldsymbol{I}$	$\boldsymbol{O}$	$\boldsymbol{\bar{I}}$	$\boldsymbol{\bar{O}}$
$\mathbb{Q}$	*Queue*	•			•
$\mathbb{S}$	*Stack*	•	•		
$\mathbb{D}^{ir}$	*Input-restricted deque*	•	•		•
$\mathbb{D}^{or}$	*Output-restricted deque*	•	•	•	
$\mathbb{D}$	*Deque*	•	•	•	•

Table 2.1. *Gate activation for each type of device.*

We use the letter $\mathbb{X}$ to generically denote one of the five devices described in table 2.1. Observe that each one of them allows a dual version $\mathbb{X}^*$, obtained by interchanging I and O with $\bar{I}$ and $\bar{O}$ (see table 2.2). Obviously, these dual versions are perfectly analogous to their original ones. Hence, when not mentioned, we will always refer to the first combination of the I/O operations (table 2.1).

2.2 Data structures and $\mathbb{X}-$sequences

The five devices described in the previous section differ for the gate activation. The particular combination of the active input and output directions leads to some very profound differences on the behaviour of the data structures.

When a given sequence of input elements passes through a device $\mathbb{X}$, many output sequences are possible. The set of all possible output sequences depends

Device		***I***	***O***	$\bar{\boldsymbol{I}}$	$\bar{\boldsymbol{O}}$
$\mathbb{Q}^*$	*Queue*		•	•	
$\mathbb{S}^*$	*Stack*			•	•
$\mathbb{D}^{ir*}$	*Input-restricted deque*		•	•	•
$\mathbb{D}^{or*}$	*Output-restricted deque*	•		•	•
$\mathbb{D}^*$	*Deque*	•	•	•	•

Table 2.2. *Dual gate activation*

on which input and output operations are performed, and, therefore, on the particular data structure that is chosen.

Definition 2.2.1. *A sequence S of input and output operations (IO–sequence) is said to be an* $\mathbb{X}$–sequence *if it can be performed by the device* $\mathbb{X}$*:*

(i) S consists of input/output operations allowed by $\mathbb{X}$;

(ii) S contains as many output operations as input ones;

(iii) every prefix of S has more input than output operations, or an equal number of them.

We will denote by $\mathcal{X}$ the set of all $\mathbb{X}$–sequences and by $\mathcal{X}_\ell$ the set of all $\mathbb{X}$–sequences of fixed length ℓ.

Example 2.2.2. The sequence $S_1 = IIOIOIOO$ is an $\mathbb{S}$–sequence (and hence a $\mathbb{D}^{ir}$, $\mathbb{D}^{or}$ and $\mathbb{D}$–sequence), while $S_2 = \bar{I}IOI\bar{O}\bar{I}OO$ is only a $\mathbb{D}$–sequence. Conversely, it is not possible to perform the sequence $S = I\bar{O}O\bar{I}\bar{I}O$ by any device, since condition (iii) does not hold.

Each device $\mathbb{X}$ is able to perform many different $\mathbb{X}$–sequences. We enumerate them in the following proposition.

Proposition 2.2.3. *Let C_n be the n–th Catalan number (see (1.2)). The following relations hold:*

$$\begin{aligned} |\mathcal{Q}_{2n}| = |\mathcal{S}_{2n}| &= C_n \\ |\mathcal{D}^{ir}_{2n}| = |\mathcal{D}^{or}_{2n}| &= 2^n C_n \\ |\mathcal{D}_{2n}| &= 2^{2n} C_n \end{aligned}$$

Proof. There exists a trivial bijection between the $\mathbb{S}$–sequences and the set $\mathscr{D}_{2n}$ of Dyck paths of length $2n$ (see chapter 5). The same holds also for the $\mathbb{Q}$–sequences, and hence $|\mathcal{Q}_{2n}| = |\mathcal{S}_{2n}| = |\mathscr{D}_{2n}| = C_n$. For the other three devices, we just observe that each $\mathbb{S}$–sequence is associated to 2^n $\mathbb{D}^{ir}$–sequences, each one obtained by swapping some of the O symbols for $\bar{O}$ in all the possible 2^n ways. The proof for $\mathbb{D}^{or}$ and $\mathbb{D}$ is analogous. □

2.3 The action of $\mathbb{X}$–sequences

Let Σ_n be the set of permutations of length n, and $\mathcal{X}_{2n}$ the set of all $\mathbb{X}$–sequences of length $2n$. We define the map

$$\begin{array}{rccc} \alpha_n\colon & \mathcal{X}_{2n} \times \Sigma_n & \longrightarrow & \Sigma_n \\ & (S, \sigma) & \longmapsto & S(\sigma) \end{array}$$

where

$$\begin{array}{rccc} S\colon & \Sigma_n & \longrightarrow & \Sigma_n \\ & \sigma & \longmapsto & S(\sigma) \end{array}$$

maps σ onto the permutation $S(\sigma)$ obtained by applying the sequence S on σ. In order to avoid ambiguity, the input permutation σ is always read from left to right, and the output permutation is created by writing the output elements from left to right. See figure 2.4 for a step-by-step description of the action of an $\mathbb{X}$–sequence.

Every $\mathbb{X}$–sequence can be performed with a generic sequence of input symbols: the final output sequence is, indeed, just a rearrangement of them. Hence, in order to describe the action of an $\mathbb{X}$–sequence we do not lose generality if we always take as input the identity permutation.

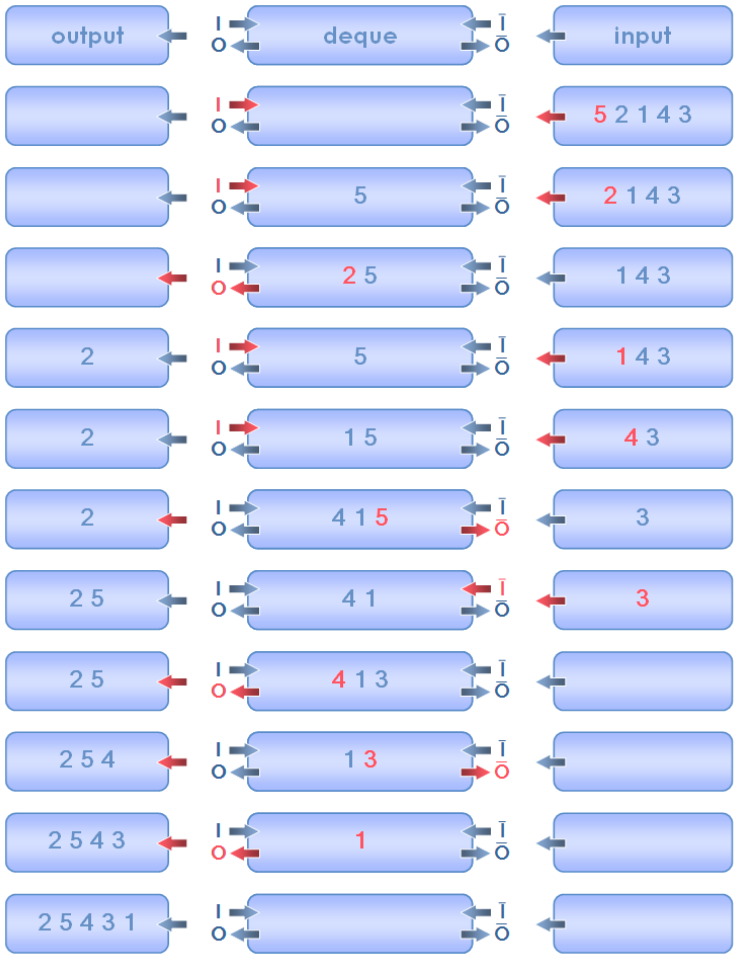


Figure 2.4. *The action of the* $\mathbb{X}$*–sequence* $S = II O I I \bar{O} \bar{I} O \bar{O} O$ *on the input permutation* $\sigma = 52143$*. The final result is the output permutation* $S(\sigma) = 25431$*.*

Proposition 2.3.1. *Let $S \in \mathcal{X}_{2n}$ and $\sigma \in \Sigma_n$, and let $id = 12\dots n$ be the identity permutation. Then, denoting by $\circ$ the usual composition of permutations, we have*

$$S(\sigma) = \sigma \circ S(id).$$

Example 2.3.2. Taking $S = II\bar{O}I\bar{I}O\bar{O}O$ and $\sigma = 52143$, as in figure 2.4, we have $S(id) = 21453$ and $S(\sigma) = \sigma \circ S(id) = 25431$.

2.4 $\mathbb{X}-$ sequences and computable permutations

In the previous section we observed that the action of IO– sequences can be analyzed, without loss of generality, by studying which permutations can be obtained starting from the identity permutation.

We can define the map

$$\begin{array}{rccc} \beta\colon & \mathcal{X} & \longrightarrow & \Sigma \\ & S & \longmapsto & S(id) \end{array},$$

which associates to each $\mathbb{X}$–sequence S the permutation $S(id)$, where the length of id is one half the length of S.

Definition 2.4.1. *Let $S \in \mathcal{X}_{2n}$ and let id be the identity permutation of length n. The permutation $S(id)$ will be called the permutation* computed *by S. Coherently, we will denote by $\mathcal{X}(id)$ the set of* $\mathbb{X}$–computable permutations*:*

$$\mathcal{X}(id) = \{\sigma \in \Sigma \,|\, \exists\, S \in \mathcal{X} \,|\, S(id) = \sigma\}.$$

In particular, we will use the symbol $\mathcal{X}(id)_n$ when we refer to $\mathbb{X}-$computable permutations of length n.

2.5 Equivalent $\mathbb{X}-$ sequences

It is easy to show that a permutation $\sigma \in \mathcal{X}(id)$ can be computed by many different $\mathbb{X}$–sequences.

Example 2.5.1. If $\mathbb{X} = \mathbb{D}$, the permutation $\sigma = 231$ is computed by $S_1 = \bar{I}IOIO\bar{O}$, $S_2 = \bar{I}I\bar{I}O\bar{O}\bar{O}$, $S_3 = I\bar{I}\bar{O}IOO$ and many others (24 in all).

For this reason, it may be convenient to consider the quotient set $\tilde{\mathcal{X}} = \mathcal{X}/\sim$, where $\sim$ is the equivalence relation defined below.

Definition 2.5.2. *Let $S, T \in \mathcal{X}_{2n}$. We say that S and T are* equivalent *($S \sim T$) if, for an input permutation $\sigma \in \Sigma_n$, they compute the same output permutation: $S(\sigma) = T(\sigma)$.*

Observe that the equivalence $\sim$ is well-defined. In fact, if $S(\sigma) = T(\sigma)$ for a fixed $\sigma \in \Sigma_n$, then the same necessarily holds for all $\tau \in \Sigma_n$. We prove this in the following proposition.

Proposition 2.5.3. *If S and T are two equivalent $\mathbb{X}$–sequences of length $2n$, then, for any fixed input permutation $\tau \in \Sigma_n$, they compute the same output:*

$$S \sim T \quad \Longleftrightarrow \quad S(\tau) = T(\tau), \quad \forall\, \tau \in \Sigma_n.$$

Proof. From the definition of equivalent $\mathbb{X}$–sequences, $S \sim T$ implies that $S(\sigma) = T(\sigma)$ for a given $\sigma \in \Sigma_n$. By proposition 2.3.1 we obtain that $S(id) = T(id)$ and hence, trivially, $S(\tau) = T(\tau)$ for every $\tau \in \Sigma_n$. □

In the next chapter, we will describe some procedures that, for each fixed device $\mathbb{X}$, define a "canonical" representative of each equivalence class of $\tilde{\mathcal{X}}$.

2.6 Computable permutations and permutation classes

As we might expect, it is not possible to compute all permutations by making use of one of the devices, even using the deque.

More precisely, the permutations which can be computed by one of the five devices can be described in terms of pattern-avoiding permutations. For a detailed proof of this, see [13], [17] and [24].

Theorem 2.6.1. *For every device $\mathbb{X}$, the set $\mathcal{X}(id)$ is a permutation class.*

In the following subsections, we will describe the basis of each class of $\mathbb{X}$–computable permutations. In the following chapter, we will discuss the same result by making use of the so called $\mathbb{X}$–*sorting procedures*.

2.6.1 Permutations computed by $\mathbb{Q}$

The set $\mathcal{Q}(id)$ consists of the identity permutations of any length:

$$\mathcal{Q}(id) = \{1, 12, 123, 1234, \dots\}.$$

In fact, the queue is the only data structure which preserves the relative order of the elements: this is the main reason why the queue, in the frame of this work, is the less interesting device. Anyhow, we can describe the set of permutations computed by $\mathbb{Q}$ as the permutation class

$$\mathcal{Q}(id) = Av(21).$$

2.6.2 Permutations computed by $\mathbb{S}$

In the first volume of *The Art of Computer Programming* [13], Donald Knuth observed that the smallest permutation which cannot be obtained using a stack is 312. In fact, the element 3 goes in the first position of the output sequence if and only if we already pushed 1 and 2 into the stack; in this situation, we cannot pop the element 1 before 2, and hence it is impossible to gain 312.

More generally, Knuth shows that the set of permutations which can be obtained through $\mathbb{S}$ is the set of 312-avoiding permutations

$$\mathcal{S}(id) = Av(312).$$

2.6.3 Permutations computed by $\mathbb{D}^{ir}$

As observed in [13], the two smallest permutations which cannot be obtained using an input-restricted deque are 4213 and 4231. The presence of 4 in the first position of the output sequence, in fact, implies that the first three elements are still into the deque when 4 is pulled out. Moreover, these elements, read from left to right, must be in decreasing order (since only the left input gate is open), and hence the element 2 cannot be popped out before 1 or 3. For this reason, the permutations 4213 and 4231 cannot be obtained using the input-resctricted deque.

More generally, as it can be deduced from [13] and [24], the set of $\mathbb{D}^{ir}-$computable permutations is the permutation class

$$\mathcal{D}^{ir}(id) = Av(4213, 4231).$$

2.6.4 Permutations computed by $\mathbb{D}^{or}$

The two smallest permutations which cannot be obtained using an output-restricted deque are 4132 and 4231. In fact, since only the left output gate is open, the latter permutations can be obtained if their subsequences 132 and 231 lie into the device in this exact order when the element 4 is pushed. However, this is not possible since the element 3 arrives when 1 and 2 are already into the deque, and we cannot push it between them.

More generally, it is known that the set of $\mathbb{D}^{or}-$computable permutations is the permutation class

$$\mathcal{D}^{or}(id) = Av(4132, 4231).$$

This result was proved by West [24]. In his proof, West makes use of the explicit enumeration of $\mathcal{D}^{or}(id)_n$, which was found by Knuth by using the so called *kernel method* on the associated generating functions.

2.6.5 Permutations computed by $\mathbb{D}$

Neither the deque is able to compute all permutations, and the set of $\mathbb{D}-$computable permutations is once again a pattern class. However, the $\mathbb{D}-$computable permutations must avoid an *infinite* set of patterns, that we denote by $T_{\mathbb{D}}$. This result was proved by Pratt [17], who also gave an explicit description of the set $T_{\mathbb{D}}$, that we show in the next theorem.

Theorem 2.6.2. *The set of permutations which can be computed by the deque is the pattern class*

$$\mathcal{D}(id) = Av(T_{\mathbb{D}}),$$

where the set $T_{\mathbb{D}}$ contains all the patterns of odd length of one of the following forms ($k \geqslant 1$):

(i) the pattern of length $4k+1$

$$5\,2\,7\,4\,\ldots\,(4k+1)\,(4k-2)\,3\,(4k)\,1,$$

which can be obtained from the identity permutation $12\ldots(4k+1)$ *by leaving the even elements fixed, rotating the odd elements cyclically left two places and interchanging* 1 *and* 3*;*

(ii) the pattern of length $4k+3$

$$5\,2\,7\,4\,\ldots\,(4k+3)\,(4k)\,1\,(4k+2)\,3,$$

which can be obtained from the identity permutation $12\ldots(4k+3)$ *by leaving the even elements fixed and rotating the odd elements cyclically left two places;*

(iii) the patterns like (i) *or* (ii)*, with the elements* 1 *and* 2 *interchanged;*

(iv) the patterns like (i) *or* (ii)*, with the last two elements interchanged;*

(v) the patterns like (i) *or* (ii)*, with both the elements* 1 *and* 2 *and the last two elements interchanged.*

Hence, the shortest forbidden patterns have length 5, and for every odd length $\ell \geqslant 5$ there are four patterns to be avoided. The set of forbidden patterns is

$$T_{\mathbb{D}} = \{52341, 51342, 52314, 51324, 5274163, 5174263, 5274136, 5174236, \ldots\}.$$

Chapter 3

Sortable permutations and sorting procedures

3.1 Sortable permutations

In the previous chapter, we considered the five data structures as devices that are able to rearrange a sequence of input elements. In particular, we took the identity permutation as input sequence and, for each type of device, we characterized the set of permutations that can be obtained thereby.

The same problem may be considered from another point of view. Suppose we have a generic permutation σ as input sequence: which one of the data structures is able to *sort* σ? In other words, which IO–sequences are able to rearrange the elements of σ so that the resulting output sequence is the identity permutation?

Definition 3.1.1. *We say that a permutation σ is* sorted *by the data structure $\mathbb{X}$ if there exists an $\mathbb{X}$–*sequence *$S \in \mathcal{X}$ such that $S(\sigma) = id$. In this case, the sequence S will be called an $\mathbb{X}$–*sorting sequence *for σ, and the set of permutation which can be sorted by $\mathbb{X}$ will be denoted by $Sort(\mathbb{X})$. In particular, we will denote by $Sort_n(\mathbb{X})$ the $\mathbb{X}$–sortable permutations of length n.*

In the next proposition, we show the profound connection between the two problems described above.

Proposition 3.1.2. *A permutation σ is sorted by an $\mathbb{X}$–sequence S if and only if its inverse σ^{-1} is computed by S:*

$$S(\sigma) = id \iff \sigma^{-1} = S(id).$$

Proof. The relation $S(\sigma) = id$ implies, by proposition 2.3.1, that $\sigma \circ S(id) = id$, and hence $S(id) = \sigma^{-1}$. □

The previous proposition implies that

$$Sort(\mathbb{X}) = \mathcal{X}(id)^{-1}, \tag{3.1}$$

where $\mathcal{X}(id)^{-1}$, by definition 1.3.4, is the set

$$\mathcal{X}(id)^{-1} = \{\sigma^{-1} \,|\, \sigma \in \mathcal{X}(id)\}.$$

In the previous chapter we proved that

$$\mathcal{X}(id) = Av(T_{\mathbb{X}}),$$

where $T_{\mathbb{X}}$ is the basis of the class of $\mathbb{X}$–computable permutations. Recalling that $Av(T)^{-1} = Av(T^{-1})$ (see proposition 1.3.7), it follows that

$$Sort(\mathbb{X}) = Av({T_{\mathbb{X}}}^{-1}).$$

Hence, for a given device $\mathbb{X}$, the set of $\mathbb{X}$–sortable permutations is a permutation class, and the patterns of the basis of $Sort(\mathbb{X})$ are the inverse patterns of the basis of $\mathcal{X}(id)$. This allows us to state the following theorem.

Theorem 3.1.3. *For every device $\mathbb{X}$, the set of $\mathbb{X}$–sortable permutations is a permutation class:*

$$\begin{aligned} Sort(\mathbb{Q}) &= Av(21), \\ Sort(\mathbb{S}) &= Av(231), \\ Sort(\mathbb{D}^{ir}) &= Av(3241, 4231), \\ Sort(\mathbb{D}^{or}) &= Av(2431, 4231), \\ Sort(\mathbb{D}) &= Av({T_{\mathbb{D}}}^{-1}), \end{aligned}$$

where

$${T_{\mathbb{D}}}^{-1} = \{52341, 25341, 42351, 24351, 5274163, 2574163, 5264173, 2564173, \dots\}$$

is the set of the inverse patterns of the set $T_{\mathbb{D}}$, defined in theorem 2.6.2.

3.2 Equivalent sorting sequences

If σ is an $\mathbb{X}$–sortable permutation, then, in general, more than one $\mathbb{X}$–sequence is able to sort it. The set of $\mathbb{X}$–sorting sequences for σ is an equivalence class under relation $\sim$ (see definition 2.5.2): in fact, they produce the same output permutation id. This class will be denoted by

$$\mathcal{C}_{\mathbb{X}}(\sigma) = \{S \in \mathcal{X} \mid S(\sigma) = id\},$$

and can be seen as the image of σ under the map

$$\begin{array}{rccc} \mathcal{C}_{\mathbb{X}}\colon & Sort(\mathbb{X}) & \longrightarrow & \tilde{\mathcal{X}} \\ & \sigma & \longmapsto & \mathcal{C}_{\mathbb{X}}(\sigma). \end{array}$$

Example 3.2.1. Let $\sigma = 312$. For each device $\mathbb{X}$, we list below the equivalence classes of its $\mathbb{X}$–sorting sequences.

$$\begin{aligned} \mathcal{C}_{\mathbb{Q}}(\sigma) &= \varnothing \\ \mathcal{C}_{\mathbb{S}}(\sigma) &= \{IIOIOO\} \\ \mathcal{C}_{\mathbb{D}^{ir}}(\sigma) &= \{IIOIOO, IIOIO\bar{O}\} \\ \mathcal{C}_{\mathbb{D}^{or}}(\sigma) &= \{IIOIOO, \bar{I}IOIOO\} \\ \mathcal{C}_{\mathbb{D}}(\sigma) &= \{IIOIOO, IIOIO\bar{O}, IIO\bar{I}\bar{O}O, IIO\bar{I}\bar{O}\bar{O}, I\bar{I}\bar{O}IOO, I\bar{I}\bar{O}IO\bar{O}, \\ &\quad I\bar{I}\bar{O}\bar{I}\bar{O}O, II\bar{I}O\bar{O}O, II\bar{I}O\bar{O}\bar{O}, I\bar{I}I\bar{O}OO, I\bar{I}I\bar{O}O\bar{O}, I\bar{I}\bar{O}\bar{I}\bar{O}\bar{O}, \\ &\quad \bar{I}I\bar{I}O\bar{O}O, \bar{I}I\bar{I}O\bar{O}\bar{O}, \bar{I}\bar{I}I\bar{O}OO, \bar{I}\bar{I}I\bar{O}O\bar{O}, \bar{I}IOIOO, \bar{I}IOIO\bar{O}, \\ &\quad \bar{I}IO\bar{I}\bar{O}O, \bar{I}IO\bar{I}\bar{O}\bar{O}, \bar{I}\bar{I}\bar{O}IOO, \bar{I}\bar{I}\bar{O}IO\bar{O}, \bar{I}\bar{I}\bar{O}\bar{I}\bar{O}O, \bar{I}\bar{I}\bar{O}\bar{I}\bar{O}\bar{O}\} \end{aligned}$$

Observe that, for the stack, the following result holds.

Proposition 3.2.2. *Let σ be a stack sortable permutation. Then, there exists only one $\mathbb{S}$–sequence that sorts σ. In other terms:*

$$|\mathcal{C}_{\mathbb{S}}(\sigma)| = 1 \qquad \forall\, \sigma \in Sort(\mathbb{S}).$$

Proof. Since $Sort(\mathbb{S}) = Av(231)$, theorem 1.5.1 and proposition 2.2.3 imply that $|Sort_n(\mathbb{S})| = |\mathcal{S}_{2n}| = C_n$. Moreover, different permutations are sorted by different $\mathbb{S}$–sequences, and this completes the proof.

The proposition can also be proved by using the characterization of the $\mathbb{S}$–sequences given in proposition 5.4.1. This avoids to involve the preceding enumerative results. □

As concerns the queue, we remark that all the possible $\mathbb{Q}$–sequences do not affect the relative order of the input elements, and hence they can only sort *id*. For this reason, from now on, in the discussion of the $\mathbb{X}$–sorting procedures we will skip the case of $\mathbb{Q}$, which is of no interest for our purposes.

3.3 Greedy sorting procedures

When talking about *stack sorting disciplines*, the stack sortable permutations are often defined in literature as permutations that can be sorted by a definite *stack sorting procedure*, that we will show in section 3.9.1. In this procedure, the wanted $\mathbb{S}$–sorting sequence is created by scanning the input permutation σ and simultaneously deciding, through a precise sorting rule, which input/output operation must be performed at each step of the process.

The definition of the $\mathbb{S}$–sortable permutations as the ones that can be sorted by the stack sorting procedure is conceptually different from our definition (3.1.1) of $\mathbb{S}$–sortable permutations: in our definition, we just require that an $\mathbb{S}$–sorting sequence exists, without any sorting procedure involved.

Actually, it is not difficult to show - as we will prove in the next sections - that these two definitions are equivalent for $\mathbb{S}$: in fact, the stack sorting procedure is not restrictive, in the sense that the set of permutations which are sorted by the procedure coincides with the set of $\mathbb{S}$–sortable permutations obtained through our definition.

What is often concealed in literature, and that we want to highlight here, is exactly the difference between these two definitions. This distinction, which does not appear very important for $\mathbb{S}$, becomes substantial for the devices which do not allow a non-restrictive sorting procedure.

For these reasons, what we want to do in the following is to determine for which devices $\mathbb{X}$ there exists, as for $\mathbb{S}$, a non-restrictive *sorting procedure* X which is able to sort, under certain conditions, all the $\mathbb{X}$–sortable permutations. This *greedy* sorting procedure associates to each $\mathbb{X}$–sortable permutation σ one of its possible $\mathbb{X}$–sorting sequences, that we will denote by $S_{\sigma,\mathbb{X}}$.

Hence, the action of the procedure X can be described through the map

$$\begin{array}{rccc} S_{\mathbb{X}}\colon & Sort(\mathbb{X}) & \longrightarrow & \mathcal{X} \\ & \sigma & \longmapsto & S_{\sigma,\mathbb{X}}. \end{array}$$

The $\mathbb{X}$–sequence $S_{\sigma,\mathbb{X}}$ can be taken as the *canonical representative* of the class $\mathcal{C}_{\mathbb{X}}(\sigma)$.

Beyond the theoretical interest on the existence of these non-restrictive sorting procedures, another relevant aspect concerns their possible use in more concrete context. We will discuss some of them in the next section.

3.4 Deterministic sorting procedures

Permutations can be seen as a simple description of a more general class of linear sequences. For instance, the problem of sorting a generic input sequence can be led back to the same problem on permutations. In fact, a permutation can be seen as a *renormalization* of a generic sequence of input symbols, taken from a totally ordered set.

In the frame of this work, our aim is to find a decision algorithm which determines if an input sequence of totally ordered symbols is $\mathbb{X}$–sortable or not. As observed above, it is sufficient to find such a decision algorithm for permutations.

A brute-force approach suggests to try all the possible $\mathbb{X}$–sequences, stopping when one of the desired sorting sequences is found. Needless to say, such a method is devoid of any theoretical interest, and definitely unusable for long input permutations: for instance, for $|\sigma| = 10$, in the worst case we have to try 16796 $\mathbb{S}$–sequences, and over 10^{10} $\mathbb{D}$–sequences (see proposition 2.2.3).

The problem is solved if we are able, for example, to exhibit a *deterministic* sorting procedure X which sorts all the $\mathbb{X}$–sortable permutations. In this case, the computational efficiency of the decision algorithm depends on the efficiency of X. Our goal is to define a sorting procedure which works in time $O(|\sigma|)$: this could be done, for example, by scanning the input permutation through the device and simultaneously deciding, at each state of the process and in time $O(1)$, which is the next input/output operation to be performed. This can be

realized, for example, by requiring that the choice of the operation to perform at each step of X must depend on some of the elements which have already been scanned and some of new input ones. In particular, in the following we will define an *operation choice* sub procedure which depends, for each fixed state of the process, only on the elements which are next to the device gates and on the first input element (see definition 3.6.1).

As far as we know, only a few authors have considered similar procedures (see Bóna [6] and, very recently, Denton [10]). In the following sections, we will show that, under the previous and other minor restrictions, there exist only one possible sorting procedure for the stack and the two restricted deques. Conversely, we will also prove that, under the same restrictions, it is not possible to implement a deque sorting procedure.

Furthermore, the previous sorting procedures can be used to define many new sorting algorithms. In fact, as we will see in chapter 4, a certain number of iterations of these procedures is sufficient to sort all the input permutations of given length.

3.5 $\mathbb{X}$–Procedures

From now on, we will use the following notations for the pseudocode of the procedures. A procedure P will be declared as follows:

Procedure P: *input* $\rightarrow$ *output*.

When we call the procedure P in the statement of an algorithm, we will denote by $P(x)$ the output produced by P when x is the input. In particular, we use the assignment statement

$$y \leftarrow P(x),$$

meaning that the variable y takes the value of the output $P(x)$.

In the code of the procedures, sequences of elements such as permutations, $\mathbb{X}$–sequences, or others will be considered as *arrays* and denoted in bold. Moreover, is $\boldsymbol{v}$ is an array, then:

- $\boldsymbol{v}[k]$ is the k–th element of the array $\boldsymbol{v}$;
- $\boldsymbol{v}[\ell]$ is the last element of the array $\boldsymbol{v}$ (meaning that the symbol ℓ denotes the length of the array);
- $\boldsymbol{v}[a \ldots b]$ is the array consisting of the elements of $\boldsymbol{v}$ from position a to position b;
- $[\boldsymbol{v}, \boldsymbol{w}]$ is the array obtained through the concatenation of $\boldsymbol{v}$ and $\boldsymbol{w}$.

In this section we analyze the procedures which can be executed by a device $\mathbb{X}$. We will use the following notations to represent the main parts of $\mathbb{X}$ involved in the sorting process:

- $\boldsymbol{input}$ is the array of input elements, and $\boldsymbol{input}[1]$ is the first element that will go into the device;
- $\boldsymbol{inside}$ is the array of elements lying inside $\mathbb{X}$; $\boldsymbol{inside}[1]$ is the leftmost element and $\boldsymbol{inside}[\ell]$ is the rightmost element;
- $\boldsymbol{output}$ is the array of output elements, where $\boldsymbol{output}[\ell]$ is the last one that moved to the output.

The following sub procedure describes the action of each input/output operation.

Procedure $Perform$:
$(operation, \boldsymbol{input}, \boldsymbol{inside}, \boldsymbol{output}) \rightarrow (\boldsymbol{input}, \boldsymbol{inside}, \boldsymbol{output})$

switch *operation*
case I:
$\boldsymbol{inside} \leftarrow [\boldsymbol{input}[1], \boldsymbol{inside}]$
$\boldsymbol{input} \leftarrow \boldsymbol{input}[2 \ldots \ell]$
case $\bar{I}$:
$\boldsymbol{inside} \leftarrow [\boldsymbol{inside}, \boldsymbol{input}[1]]$
$\boldsymbol{input} \leftarrow \boldsymbol{input}[2 \ldots \ell]$
case O:
$\boldsymbol{output} \leftarrow [\boldsymbol{output}, \boldsymbol{inside}[1]]$

$\boldsymbol{inside} \leftarrow \boldsymbol{inside}[2 \ldots \ell]$

case $\bar{O}$:

$\boldsymbol{output} \leftarrow [\boldsymbol{output}, \boldsymbol{inside}[\ell]]$

$\boldsymbol{inside} \leftarrow \boldsymbol{inside}[1 \ldots \ell - 1]$

end switch

Example 3.5.1. In the situation depicted in figure 3.1 we have $\boldsymbol{input} = 72$, $\boldsymbol{inside} = 314$ and $\boldsymbol{output} = 65$. Table 3.1 shows the new content of

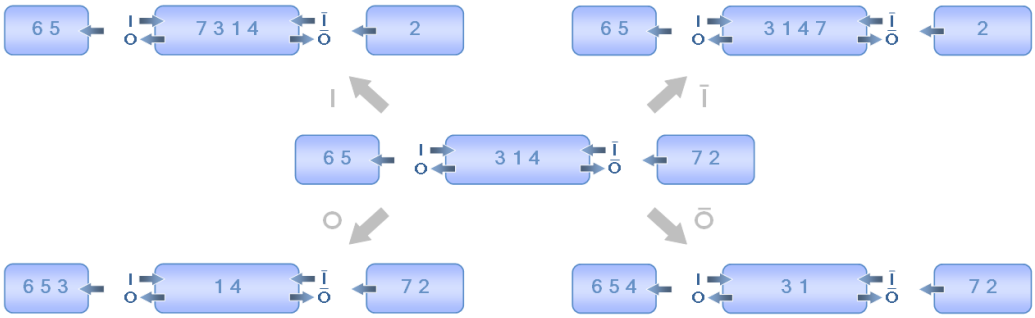


Figure 3.1

$\boldsymbol{input}$, $\boldsymbol{inside}$ and $\boldsymbol{output}$ after performing one of the possible input/output operations.

Definition 3.5.2. *Let P be a procedure which executes an $\mathbb{X}$–sequence $S = S_1 S_2 \ldots S_{2n}$ onto an input permutation $\sigma \in \Sigma_n$. The procedure P consists of* $2n$ steps*: each step coincides with the execution of one of the input/output operations of S. More precisely, we call* step t *the execution of the operation S_t through the procedure Perform; we call* state t *the content of the arrays $\boldsymbol{input}$, $\boldsymbol{inside}$ and $\boldsymbol{output}$ between step t and step $t+1$.*

For simplicity, we denote by

$$\langle State \rangle_{(t)} = (\boldsymbol{input}_{(t)}, \boldsymbol{inside}_{(t)}, \boldsymbol{output}_{(t)})$$

	input	***inside***	***output***
Initial state	72	314	65
After I	2	7314	65
After $\bar{I}$	2	3147	65
After O	72	14	653
After $\bar{O}$	72	31	654

Table 3.1. *The action of the four input/output operations for the situation depicted in figure 3.1.*

the elements involved at a fixed state t of the procedure. According to this convention, the initial state is

$$\langle State \rangle_{(0)} = (\sigma, \varnothing, \varnothing),$$

and the final state is

$$\langle State \rangle_{(2n)} = (\varnothing, \varnothing, S(\sigma)).$$

We also denote by

$$\langle State \rangle_{(t)} \oplus op$$

the state obtained by performing the operation op on $\langle State \rangle_{(t)}$, and we denote by

$$\boldsymbol{input}_{(t)} \oplus op$$

$$\boldsymbol{inside}_{(t)} \oplus op$$

$$\boldsymbol{output}_{(t)} \oplus op$$

the corresponding parts of the device after the same operation. In particular, when an $\mathbb{X}$–sequence $S = S_1 S_2 \dots S_{2n}$ is given, we have

$$\langle State \rangle_{(t+1)} = \langle State \rangle_{(t)} \oplus S_t \qquad .$$

Example 3.5.3. Take $\mathbb{X} = \mathbb{D}$, $S = IIOII\bar{O}\bar{I}O\bar{O}O$ and $\sigma = 52143$, as in figure 2.4. In table 3.2 we list the content of ***input***, ***inside*** and ***output*** at each state of the execution of S. We have, for instance, $\langle State \rangle_{(2)} = (143, 25, \varnothing)$ and $\langle State \rangle_{(3)} = \langle State \rangle_{(2)} \oplus O = (143, 5, 2)$.

	input	***inside***	***output***	*Next step*
State 0	52143			I
State 1	2143	5		I
State 2	143	25		O
State 3	143	5	2	I
State 4	43	15	2	I
State 5	3	415	2	$\bar{O}$
State 6	3	41	25	$\bar{I}$
State 7		413	25	O
State 8		13	254	$\bar{O}$
State 9		1	2543	O
State 10			25	

Table 3.2. *Analysis of* $Execute(IIOII\bar{O}\bar{I}O\bar{O}O, 52143)$.

We distinguish between two types of procedures:

- the procedure *Execute*;
- the procedure $\mathbb{X}$–$CreateExecute$.

In the procedure *Execute*, the IO–sequence is already given as input together with the permutation σ, and hence the procedure does not depend on the device:

Procedure $Execute\colon (\boldsymbol{S}, \boldsymbol{\sigma}) \to \boldsymbol{\tau}$
  $\boldsymbol{input} \leftarrow \boldsymbol{\sigma}$
  $\boldsymbol{inside} \leftarrow \varnothing$
  $\boldsymbol{output} \leftarrow \varnothing$
  $n \leftarrow length(\boldsymbol{\sigma})$
  for *step* **from** 1 **to** $2n$ **do**
    $(\boldsymbol{input}, \boldsymbol{inside}, \boldsymbol{output}) \leftarrow Perform(\boldsymbol{S}[step], \boldsymbol{input}, \boldsymbol{inside}, \boldsymbol{output})$

end for
$\boldsymbol{\tau} \leftarrow \boldsymbol{output}$

In the procedure $\mathbb{X}$–*CreateExecute*, each operation S_t of the $\mathbb{X}$–sequence $S = S_1 S_2 \dots S_{2n}$ is created and immediately executed at the step t of the process. More precisely, as already mentioned in section 3.4, in order to keep the constant execution time we require that the choice of the operation to perform must depend only on the first, second and last elements of $\boldsymbol{inside}_{(t)}$ (or, if possible, only some of them) and on the first input element $\boldsymbol{input}[1]_{(t)}$. This will be done by using the sub procedure

Procedure $\mathbb{X}$–*OperationChoice*:

$$(\boldsymbol{inside}[1], \boldsymbol{inside}[2], \boldsymbol{inside}[\ell], \boldsymbol{input}[1]) \rightarrow operation.$$

at step t of the main procedure $\mathbb{X}$–*CreateExecute*. Actually, as we will see in section 3.9.2, if $\mathbb{X} = \mathbb{D}^{or}$ it is sufficient to know $\boldsymbol{inside}[1]$, $\boldsymbol{inside}[\ell]$ and $\boldsymbol{input}[1]$. Moreover, if $\mathbb{X} = \mathbb{S}$ we just require $\boldsymbol{inside}[1]$ and $\boldsymbol{input}[1]$. In the following, for simplicity, when referring to the operation choice procedure for a generic device $\mathbb{X}$ we will insert all the four parameters, although some of them are not necessary for the stack and the output-restricted deque.

The procedure $\mathbb{X}$–*OperationChoice* will be described in detail, for each device $\mathbb{X}$, in section 3.9. We give below the pseudocode of the main procedure.

Procedure $\mathbb{X}$–*CreateExecute*: $\boldsymbol{\sigma} \rightarrow \boldsymbol{\tau}$
$\boldsymbol{input} \leftarrow \boldsymbol{\sigma}$
$\boldsymbol{inside} \leftarrow \varnothing$
$\boldsymbol{output} \leftarrow \varnothing$
$\boldsymbol{S} \leftarrow \varnothing$
$step \leftarrow 0$
while $\boldsymbol{inside} \neq \varnothing \vee \boldsymbol{input} \neq \varnothing$ **do**
    $step \leftarrow step + 1$
    $operation \leftarrow$
        $\leftarrow \mathbb{X}$–$OperationChoice(\boldsymbol{inside}[1], \boldsymbol{inside}[2], \boldsymbol{inside}[\ell], \boldsymbol{input}[1])$

$\boldsymbol{S}[step] \leftarrow operation$
$(\boldsymbol{input}, \boldsymbol{inside}, \boldsymbol{output}) \leftarrow$
$\leftarrow Perform(\boldsymbol{S}[step], \boldsymbol{input}, \boldsymbol{inside}, \boldsymbol{output})$
end while
$\boldsymbol{\tau} \leftarrow \boldsymbol{output}$

3.6 $\mathbb{X}$– Sorting Procedures

Definition 3.6.1. *A procedure will be called an* $\mathbb{X}$–sorting procedure*, and denoted by* X*, if:*

(i) X *is a procedure of type* $\mathbb{X}$*–CreateExecute: the choice of the operation to perform at step* $t+1$ *depends only on the first element of the input sequence and on the first, second and last element which lie inside* $\mathbb{X}$ *at state* t (local operation choice condition)*;*

(ii) X *sorts all the possible* $\mathbb{X}$*–sortable permutations* (optimality condition)*; more precisely, the set of permutations sorted by* X

$$Sort(\mathsf{X}) = \{\sigma \in \Sigma \,|\, \mathsf{X}(\sigma) = id\}$$

coincides with the set of $\mathbb{X}$*–sortable permutations:*

$$Sort(\mathsf{X}) = Sort(\mathbb{X}).$$

Hence, an $\mathbb{X}$*–sorting procedure* takes a permutation σ, creates and immediately executes an $\mathbb{X}$–sequence $S_{\sigma,\mathbb{X}}$ and produces as output the permutation $\tau = S_{\sigma,\mathbb{X}}(\sigma)$, obtained by applying the sequence of operations $S_{\sigma,\mathbb{X}}$ to the input permutation σ. In particular, by condition (ii), it follows that

$$\sigma \in Sort(\mathbb{X}) \iff S_{\sigma,\mathbb{X}}(\sigma) = id. \tag{3.2}$$

Definition 3.6.2. *The* $\mathbb{X}$*–sequence* $S_{\sigma,\mathbb{X}}$ *will be called the* $\mathbb{X}$–sorting sequence associated to σ by the $\mathbb{X}$–sorting procedure X*.*

$\mathbb{X}$– **Sorting Procedure** X: $\boldsymbol{\sigma} \to \boldsymbol{\tau}$

$\boldsymbol{input} \leftarrow \boldsymbol{\sigma}$
$\boldsymbol{inside} \leftarrow \varnothing$
$\boldsymbol{output} \leftarrow \varnothing$
$\boldsymbol{S}_{\boldsymbol{\sigma},\mathbb{X}} \leftarrow \varnothing$
$step \leftarrow 0$
while $\boldsymbol{inside} \neq \varnothing \vee \boldsymbol{input} \neq \varnothing$ **do**
$step \leftarrow step + 1$
$operation \leftarrow$
$\leftarrow \mathbb{X}\text{–}\, OperationChoice(\boldsymbol{inside}[1], \boldsymbol{inside}[2], \boldsymbol{inside}[\ell], \boldsymbol{input}[1])$
$\boldsymbol{S}_{\boldsymbol{\sigma},\mathbb{X}}[step] \leftarrow operation$
$(\boldsymbol{input}, \boldsymbol{inside}, \boldsymbol{output}) \leftarrow$
$\leftarrow Perform(\boldsymbol{S}_{\boldsymbol{\sigma},\mathbb{X}}[step], \boldsymbol{input}, \boldsymbol{inside}, \boldsymbol{output})$
end while
$\boldsymbol{\tau} \leftarrow \boldsymbol{output}$

Example 3.6.3. We analyze the action of the $\mathbb{S}$–sorting procedure S. If we take $\sigma = 312$ as input permutation, we will see in section 3.9 that the procedure S creates the $\mathbb{S}$–sequence $S_{\sigma,\mathbb{S}} = IIOIOO$ and produces the permutation $\tau = 123$ as output (see table 3.3). In fact, σ is $\mathbb{S}$–sortable since it avoids 231 (see theorem 3.1.3), and hence, as stated in (3.2), $\tau = S_{\sigma,\mathbb{S}}(\sigma) = id$.

	input	***inside***	***output***	*Next step*
State 0	312			I
State 1	12	3		I
State 2	2	13		O
State 3	2	3	1	I
State 4		23	1	O
State 5		3	12	O
State 6			123	

Table 3.3. *Analysis of* $Execute(IIOIOO, 312)$.

Conversely, if we take $\sigma' = 231$ as input permutation, the $\mathbb{S}$–sequence associated to σ' is $S_{\sigma',\mathbb{S}} = IOIIOO$ and produces the permutation $\tau' = 213 = S_{\sigma',\mathbb{S}}(\sigma')$ as output, which is not the identity permutation since τ' is not $\mathbb{S}$–sortable. If our objective is to sort σ', we can iterate S on τ', and, if necessary, on the subsequent output permutations. In this case just one iteration is needed, since $S_{\tau',\mathbb{S}}(\tau') = id$.

Up to now, we have defined the structure and the main features of the sorting procedures, but we have not yet described in detail the operation choice procedures. In order to do this, we will show in the next sections some necessary conditions, that will help us in the definition of the choice procedures.

3.7 Inside $\mathbb{X}$: monotonicity and unimodality

Definition 3.7.1. *A sequence $\tau = \tau_1\tau_2\dots\tau_n$ is* unimodal *if there exists p ($1 \leqslant p \leqslant n-1$) such that the subsequences $\tau_1\dots\tau_p$ and $\tau_{p+1}\dots\tau_n$ are, respectively, increasing and decreasing:*

$$\tau_i \leqslant \tau_j, \quad \forall\, i,j : i \leqslant j \leqslant p \qquad \text{and} \qquad \tau_i \geqslant \tau_j, \quad \forall\, i,j : p \leqslant i \leqslant j.$$

We remark that *increasing* and *decreasing* sequences are always unimodal (by setting, respectively, $p = n-1$ and $p = 1$).

Proposition 3.7.2. *Suppose that a permutation σ is sorted by an IO–sequence S, namely, $S(\sigma) = id$. Hence, at each state t of the sorting process:*

(i) if $S \in \mathcal{D}^{or}$, then $\boldsymbol{inside}_{(t)}$ is increasing;

(ii) if $S \in \mathcal{D}$, then $\boldsymbol{inside}_{(t)}$ is unimodal.

Proof. In order to obtain the increasing output permutation id, when an element leaves the device it must be the minimum among all the elements of $\boldsymbol{inside}$. Hence, at each state of the sorting process, in case (i) the minimum lies always in the leftmost position (which is the closest to the output gate), and this implies that, for every t, the elements of $\boldsymbol{inside}_{(t)}$ must be increasing. In case (ii), the minimum lies at either end of the device, and hence the sequence of elements of $\boldsymbol{inside}_{(t)}$ is always unimodal. □

We recall that $\mathcal{S} \subset \mathcal{D}_{or}$ and $\mathcal{D}_{ir} \subset \mathcal{D}$. Hence, the following corollary is straightforward.

Corollary 3.7.3. *Let $\mathbb{X}$ be the device used to sort an input permutation σ. Hence, at each state t of the sorting process:*

(i) if $\mathbb{X} = \mathbb{S}$ or $\mathbb{X} = \mathbb{D}^{or}$, then $\boldsymbol{inside}_{(t)}$ is increasing;

(ii) if $\mathbb{X} = \mathbb{D}^{ir}$ or $\mathbb{X} = \mathbb{D}$, then $\boldsymbol{inside}_{(t)}$ is unimodal.

3.8 Operation choice rules

Before giving the operation choice rules, we recall that we are dealing with permutations although they should be considered as representations of a more general input sequence of totally ordered symbols (see section 3.4). Hence, if we want the choice rules to be used even on generic sequences of symbols, we cannot use the typical properties of permutations (that a generic sequence does not have) with the intention of simplifying the final algorithm.

For instance, we cannot use the fact that the element 1 is the smallest element in order to put it in the ouput as soon as it arrives. This kind of approach, indeed, implicitly assumes that the operation choices can be made by knowing a priori *all* the elements involved in the process: this conflicts with condition (i) of definition 3.6.1, which states that the choice of the operation to perform is made by knowing at most three elements inside the device (the first, second and last) and only the first input one.

In other terms, we can state that the choice of the operation to perform is made depending only on the *relative order* of the four cited elements.

For the first choice rule, we observe that the output operations are always *dangerous*. In fact, whenever we output an element a we are aware that the output sequence will not be sorted if another element $b < a$ will occur later. For this reason, the first operation choice rule is that we always input elements whenever possible. In other terms, if we can choose between an input or an output operation, we must prefer the first. We formally state this in the following choice rule.

Operation Choice Rule 1. *Let X be an $\mathbb{X}$–sorting procedure and let $S_{\sigma,\mathbb{X}}$ be the $\mathbb{X}$–sorting sequence associated to σ by X. For each state t of X:*

(a) if $\boldsymbol{input}_{(t)} \neq \varnothing$ and at least one between

$$\boldsymbol{inside}_{(t)} \oplus I \qquad \text{and} \qquad \boldsymbol{inside}_{(t)} \oplus \bar{I}$$

is either increasing (for $\mathbb{X} = \mathbb{S}$, $\mathbb{D}^{or}$) or unimodal (for $\mathbb{X} = \mathbb{D}^{ir}$, $\mathbb{D}$) then the $(t+1)$–th operation $(S_{\sigma,\mathbb{X}})_{t+1}$ of $S_{\sigma,\mathbb{X}}$ is either

$$(S_{\sigma,\mathbb{X}})_{t+1} = I \qquad or \qquad (S_{\sigma,\mathbb{X}})_{t+1} = \bar{I};$$

(b) otherwise, the $(t+1)$–th operation of $S_{\sigma,\mathbb{X}}$ is either

$$(S_{\sigma,\mathbb{X}})_{t+1} = O \qquad or \qquad (S_{\sigma,\mathbb{X}})_{t+1} = \bar{O}.$$

Now, we discuss separately the input and output operations. The first, base rule is given below.

Operation Choice Convention. *There are three different situations for which the operation choice is irrelevant:*

- *when $\boldsymbol{inside}_{(t)} = \varnothing$ and $\boldsymbol{input}_{(t)} \neq \varnothing$, then only the input operations I and $\bar{I}$ are allowed and they give the same result: in this situation we will always perform I;*
- *when $\boldsymbol{input}_{(t)} = \varnothing$ and $\boldsymbol{inside}_{(t)}$ contains only one element, then only the output operations O and $\bar{O}$ are allowed and they give the same result: in this situation we will always perform O;*
- *if $\boldsymbol{inside}[1] = \boldsymbol{inside}[\ell]$ (this does not occur when the input sequence is a permutation) and we are obliged, for other reasons, to output one of these elements, we will always perform O.*

This convention guarantees the uniqueness of the choice rules that we are going to describe.

When an input operation is allowed (case (a) of proposition 1), the following operation choice rule holds.

Operation Choice Rule 2. *Let $S_{\sigma,\mathbb{X}}$ be the $\mathbb{X}$–sorting sequence associated to σ by the $\mathbb{X}$–sorting procedure X, and suppose that, for a fixed state t of X, either $(S_{\sigma,\mathbb{X}})_{t+1} = I$ or $(S_{\sigma,\mathbb{X}})_{t+1} = \bar{I}$. Then:*

(i) if $\mathbb{X} = \mathbb{S}$ or $\mathbb{X} = \mathbb{D}^{ir}$ then $(S_{\sigma,\mathbb{X}})_{t+1} = I$;

(ii) if $\mathbb{X} = \mathbb{D}^{or}$ and $\boldsymbol{inside}_{(t)} \neq \varnothing$ then

(a) $(S_{\sigma,\mathbb{X}})_{t+1} = I$ if $\boldsymbol{input}[1]_{(t)} \leqslant \boldsymbol{inside}[1]_{(t)}$;
(b) $(S_{\sigma,\mathbb{X}})_{t+1} = \bar{I}$ if $\boldsymbol{input}[1]_{(t)} \geqslant \boldsymbol{inside}[\ell]_{(t)}$;

Proof. The proof of case (i) is straightforward, since $\bar{I}$ is not allowed for $\mathbb{S}$ and $\mathbb{D}^{ir}$. For case (ii), we first observe that the first input element $\boldsymbol{input}[1]_{(t)}$ must be smaller than $\boldsymbol{inside}[1]_{(t)}$ or greater than $\boldsymbol{inside}[\ell]$, or equal to one of them (if we consider generic sequences of input symbols): otherwise, the insertion of $\boldsymbol{input}[1]_{(t)}$ will break the monotonicity of $\boldsymbol{inside}$, and this (by corollary 3.7.3) leads to a final unsorted output sequence, which conflicts with the definition of $\mathbb{X}$–sorting sequence. In the other cases, in order to preserve the monotonicity of $\boldsymbol{inside}$, we must perform I when $\boldsymbol{input}[1]_{(t)} \leqslant \boldsymbol{inside}[1]_{(t)}$ and $\bar{I}$ otherwise. □

Lemma 3.8.1. *Let X be an $\mathbb{X}$–sorting procedure. Hence, at each state t of X we have*

$$\min\left[\boldsymbol{input}_{(t)} \cup \boldsymbol{inside}_{(t)}\right] \geqslant \max \boldsymbol{output}_{(t)}.$$

Proof. Suppose that there exists a state t and an element $a \in \boldsymbol{input}_{(t)} \cup \boldsymbol{inside}_{(t)}$ such that $a < \max \boldsymbol{output}_{(t)}$. Hence, when we put a into $\boldsymbol{output}$ we break the monotonicity of the final output sequence, and hence we cannot get *id*. □

When the input operations are not allowed (case (b) of the operation choice rule 1), the following rule holds.

Operation Choice Rule 3. *Let $S_{\sigma,\mathbb{X}}$ be the $\mathbb{X}$–sorting sequence associated to σ by the $\mathbb{X}$–sorting procedure X, and suppose that, for a fixed state t of X, we have either $(S_{\sigma,\mathbb{X}})_{t+1} = O$ or $(S_{\sigma,\mathbb{X}})_{t+1} = \bar{O}$. Then:*

(i) *if* $\mathbb{X} = \mathbb{S}$ *or* $\mathbb{X} = \mathbb{D}^{or}$ *then* $(S_{\sigma,\mathbb{X}})_{t+1} = O$;

(ii) *if* $\mathbb{X} = \mathbb{D}^{ir}$ *then*

(a) *if* $\boldsymbol{inside}[1]_{(t)} \leqslant \boldsymbol{inside}[\ell]_{(t)}$ *then* $(S_{\sigma,\mathbb{X}})_{t+1} = O$;

(b) *if* $\boldsymbol{inside}[1]_{(t)} > \boldsymbol{inside}[\ell]_{(t)}$ *then* $(S_{\sigma,\mathbb{X}})_{t+1} = \bar{O}$.

Proof. The proof of case (i) is straightforward, since $\bar{I}$ is not allowed for $\mathbb{S}$ and $\mathbb{D}^{or}$. For case (ii), if we perform O in case (b) we violate the condition of lemma 3.8.1; the same occurs performing $\bar{O}$ in case (a), except when $\boldsymbol{inside}[1]_{(t)} = \boldsymbol{inside}[\ell]_{(t)}$: in this case, we perform O according to the third operation choice convention. □

3.9 $\mathbb{X}$– OperationChoice Procedures

The three operation choice rules described in the previous section, together with the operation choice convention, lead to a *unique* possible procedure $\mathbb{X}$– OperationChoice for $\mathbb{S}$, $\mathbb{D}^{ir}$ and $\mathbb{D}^{or}$.

In the following subsections we show the operation choice procedures for these devices and we will prove that no operation choice procedure is allowed for the deque. In the graphical description of the procedures we will use bullets instead of numbers to represent the elements involved, with the convention that the elements drawn higher up are greater than those depicted lower down. Moreover, we draw the first input element:

- in *green*, if it can be inserted into the device without affecting the monotonicity, or the unimodality, of the inner elements (according to corollary 3.7.3);
- in *yellow*, if it can be inserted only after the extraction of some of the inner elements; as already observed in section 3.8, this operation prevents the final output sequence to be sorted, if later occurs another element which is smaller than the ones which have been extracted;
- in *red*, if its insertion will definitely cause an unsorted final output sequence.

3.9.1 The procedure $\mathbb{S}-$OperationChoice

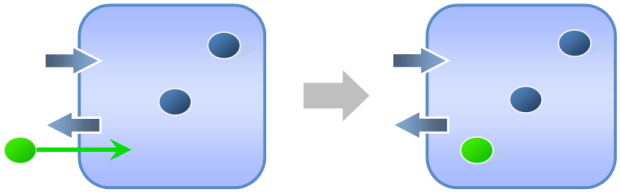
The stack sorting rule is very well known in literature (see e.g. [5]), and it is easy to show that it is the only possible one for $\mathbb{S}$. However, many authors often omit to show that this rule is an optimal sorting rule, in the sense that it is able to sort all the possible $\mathbb{S}$–sortable permutations. Here, the rule is given as the choice rule of the $\mathbb{S}$–sorting procedure, and hence, according to definition 3.6.1, its optimality is guaranteed.

Procedure $\mathbb{S}-OperationChoice$: $(\boldsymbol{inside}[1], \boldsymbol{input}[1]) \to operation$

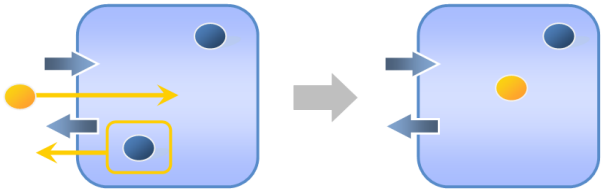
```
if inside[1] = ∅ ∧ input[1] ≠ ∅ then
    operation ← I
else
   if inside[1] ≠ ∅ ∧ input[1] = ∅ then
       operation ← O
   else
      if inside[1] ≠ ∅ ∧ input[1] ≠ ∅ then
          if input[1] ⩽ inside[1] then
             operation ← I
          else
             operation ← O
          end if
       end if
    end if
end if
```

Let us describe the rule for a nontrivial case, namely, when both ***inside*** and ***input*** are nonempty. By corollary 3.7.3, the elements inside $\mathbb{S}$ must be increasing; according to the first operation choice rule, the first input element can be inserted into $\mathbb{S}$ if there are no smaller elements inside $\mathbb{S}$ (figure 3.2a); otherwise, we must take out all the smaller elements before inserting the new one (figure 3.2b).

In these last situations, the first input element is yellow because its inser-



(a) *The input element is smaller than all the elements inside the stack, and its insertion does not affect the monotonicity of the inner sequence. We just perform* I.



(b) *Before inserting the new input element we must extract the smaller elements. In the situation depicted above, the sequence of operations to perform is* OI.

Figure 3.2. *The two possible situations that arise for the stack.*

tion might prevent the final output permutation to be sorted. This happens if and only if, after the insertion of the yellow element, an element smaller than at least one of the elements that we have extracted will occur in input. For example, the situation of figure 3.2b and the consequent arrival of a smaller element correspond to an occurence of a 231 pattern: the element 2 is represented by the one (blue) which is removed when 3 (yellow) arrives; later, when 1 appears, the element 2 has already been removed, and hence 1 will follow 2 in the final output sequence. This is a further proof of

$$Sort(\mathbb{S}) = Av(231).$$

3.9.2 The procedure $\mathbb{D}^{ir}-$OperationChoice

Even for the input-restricted deque, the operation choice rules and conventions lead to a unique possible sorting procedure, that we will discuss in the following.

Procedure $\mathbb{D}^{ir}-OperationChoice$:
$(\boldsymbol{inside}[1], \boldsymbol{inside}[2], \boldsymbol{inside}[\ell], \boldsymbol{input}[1]) \rightarrow operation$

```
if inside[1] = ∅ ∧ input[1] ≠ ∅ then
    operation ← I
else
    if inside[1] ≠ ∅ ∧ input[1] = ∅ then
        if inside[1] ⩽ inside[ℓ] then
            operation ← O
        else
            operation ← Ō
        end if
    else
        if inside[1] ≠ ∅ ∧ input[1] ≠ ∅ then
            if input[1] ⩽ inside[1] ∨
                ∨inside[1] ⩾ inside[2] ∨ inside[2] = ∅ then
                operation ← I
            else
```

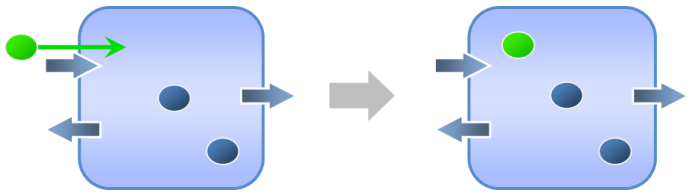
```
                if inside[1] ⩽ inside[ℓ] then
                    operation ← O
                else
                    operation ← Ō
                end if
            end if
        end if
    end if
end if
```

As stated in corollary 3.7.3, the sequence of elements inside $\mathbb{D}^{ir}$ must always be unimodal. Hence, at a fixed (nontrivial) state of the sorting process, one of the following two cases arises:

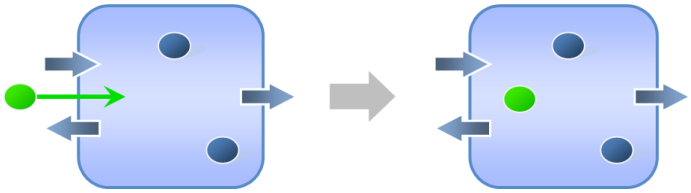
- if ***inside*** is decreasing, the insertion of a new input element is always possible, since it does not affect the monotonicity of ***inside*** (figure 3.3a) or turns its monotonicity into unimodality (figure 3.3b);
- if ***inside*** is unimodal (but not decreasing), we can insert a new element without extractions if and only if it is smaller then the first inner element (figure 3.4a); otherwise, before inserting it is necessary to take out at least one of the inner elements (figures 3.4b and 3.4c).

We observe that, since the inner sequence is always unimodal, in order to verify if it is also decreasing it is sufficient to compare its first two elements (this occurs if and only if $\boldsymbol{inside}[1] \geqslant \boldsymbol{inside}[2]$). Hence, in the operation choice procedure we get this information in costant time, without scanning all the elements of ***inside***.

The situation of the last two figures might lead to a final unsorted sequence. This occurs if an element smaller than the maximum of the ones that we have extracted will occur in input after the insertion of the yellow element. In other terms, the situation of figure 3.4b and the consequent arrival of a smaller element corresponds to an occurence of a 4231 pattern. In fact, the

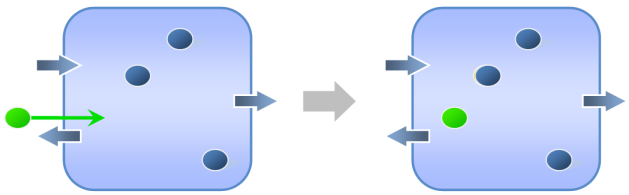


(a) *The maximum of* ***inside*** *is the element closest to the left gate (i.e. the inner sequence is decreasing), and the new input element is greater than all the inner ones. Hence, its insertion (I) does not affect the monotonicity of* ***inside****.*

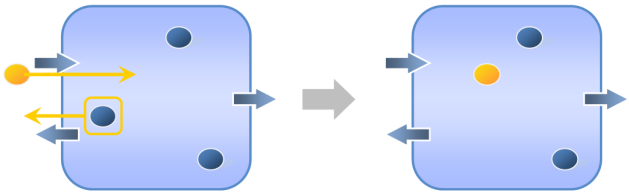


(b) *The inner sequence is decreasing and the new input element is smaller than the maximum of* ***inside****. Hence, after the insertion of the new input element, the inner sequence gets unimodal.*

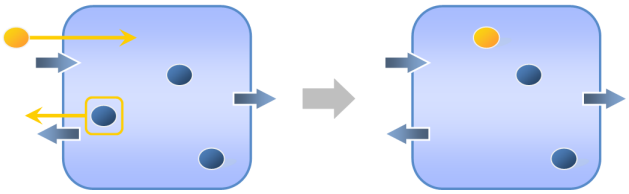
Figure 3.3. *The two possible situations that arise when the sequence of elements inside the input-restricted deque is increasing.*



(a) *The insertion of the input element does not affect the unimodality of the inner sequence. We just perform I.*



(b) *Before inserting the new input element we must extract the elements that are smaller than the input one. In this case, the final inner sequence is still unimodal.*



(c) *As in the previous case, we must extract the smaller inner elements before inserting the new one. The final inner sequence becomes decreasing.*

Figure 3.4. *The three possible situations that arise when the sequence of elements inside the input-restricted deque is unimodal.*

element 4 is the central (blue) inner element; the element 2 is the element that is extracted when 3 (yellow) arrives; later, when 1 appears, the element 2 is already in the output sequence, which will not be sorted. A similar situation occurs in the case of figure 3.4c, where the depicted state and the consequent arrival of a smaller element corresponds to an occurence of a 3241 pattern. In this case, the element 3 is the central inner element; the element 2 is the element that is extracted when 4 (yellow) arrives; later, when 1 appears, the element 2 is already in the output sequence, which will not be sorted.

These two considerations show that

$$Sort(\mathbb{D}^{ir}) = Av(3241, 4231).$$

3.9.3 The procedure $\mathbb{D}^{or}-$OperationChoice

The operation choice rules and conventions lead to the following, unique possible sorting procedure.

Procedure $\mathbb{D}^{or}-OperationChoice$:
$(\boldsymbol{inside}[1], \boldsymbol{inside}[\ell], \boldsymbol{input}[1]) \rightarrow operation$

```
if inside[1] = ∅ ∧ input[1] ≠ ∅ then
    operation ← I
else
    if inside[1] ≠ ∅ ∧ input[1] = ∅ then
        operation ← O
    else
        if inside[1] ≠ ∅ ∧ input[1] ≠ ∅ then
            if input[1] ⩽ inside[1] then
                operation ← I
            else
                if input[1] ⩾ inside[ℓ] then
                    operation ← Ī
                else
                    operation ← O
                end if
```

end if
end if
end if
end if

Recall that, by corollary 3.7.3, the elements inside $\mathbb{D}^{or}$ must be increasing. In the following figures, we consider the possible insertion of the first input element through both the ends of the deque.

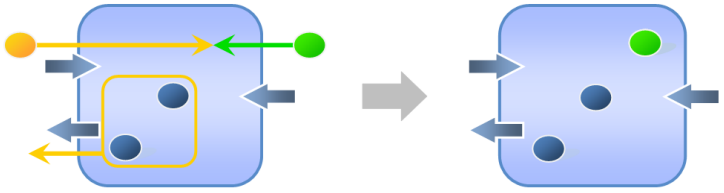
Suppose that the new input element is bigger than all the inner ones (figure 3.5a). In this case, if we insert through the left gate (yellow) we must remove all the inner elements; therefore, it is preferable to insert the input element through the right one (green), since this operation ($\bar{I}$) does not affect the monotonicity of ***inside***.

If the new input element is smaller than all the inner ones (figure 3.5b), if we insert it through the right gate (red) we will not be able to get the final sorted sequence. Hence, we must insert it through the left gate (green), without taking out any of the inner elements.

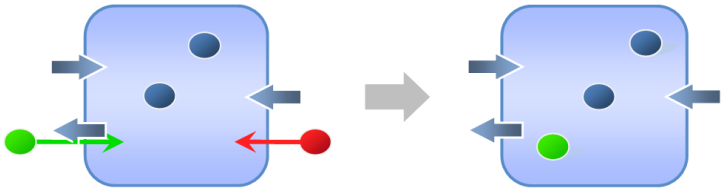
If the new input element is greater than the minimum of ***inside*** but smaller than its maximum (figure 3.5c), inserting through the right gate (red) would definitely cause the output sequence to be unsorted. Hence, we must extract the smaller elements and then insert the input one (yellow) through the left gate.

Observe that the situation of figure 3.5c leads to a final unsorted sequence if and only if either a 2431 or a 4231 pattern occurs. The first elements 2 and 4 correspond to the two elements inside the deque (the minimum and the maximum, respectively) when the 3 arrives. Their relative arrival order (24 or 42) is irrelevant, since they take place into the deque through different gates. Since the element 3 pulls out 2, a future element 1 will prevent the final output permutation to be sorted. This agrees with the permutation class description of the $\mathbb{D}^{or}-$sortable permutations:

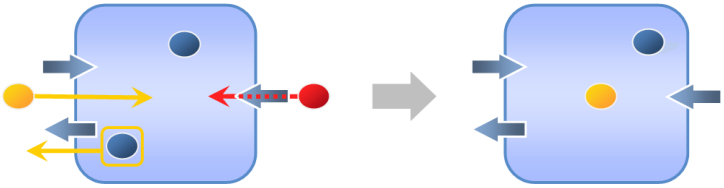
$$Sort(\mathbb{D}^{or}) = Av(2431, 4231).$$



(a) *The new input element is greater than all the inner ones. We insert it through the right gate ($\bar{I}$).*



(b) *The new input element is smaller than all the inner ones. We can perform I.*



(c) *The new input element is greater than the minimum and smaller than the maximum of* ***inside****. Hence, we remove the smaller elements and then we insert the new one through the left gate.*

Figure 3.5. *The three possible situations that arise for the output-restricted deque.*

3.9.4 The case of $\mathbb{D}$

The deque sorting problem is not as famous as the stack sorting one, but it occasionally appeared in literature, especially since Knuth introduced it in his book [13].

The most famous deque sorting algorithm is due to Rosenstiehl and Tarjan [19], who ingeniously solved the problem by using a *deque of twin stacks*. However, the algorithm described in their paper is very different from our notion of $\mathbb{D}$–sorting procedure. In fact, besides using an additional structure which is not allowed in our sorting procedures, the main difference is that in Rosenstiehl and Tarjan's method we always know that we are sorting a permutation, and not a generic sequence of input symbols. In fact, the algorithm provides that, at a generic step, the element $k+1$ will be added to the output as soon as the output sequence is $12\dots k$.

The difference between the above notions of sorting procedures is also analyzed in a very recent paper by Denton [10], who distinguishes the two previous notions by calling them *with complete* and *with incomplete* information. Our notion of $\mathbb{D}$–sorting procedure, which is based on the local operation choice condition (see definition 3.6.1), corresponds to a non-omniscient view of the entire sequence of input elements, and hence to a deque sorting method with incomplete information.

By the operation choice rules and conventions discussed in the previous section, we are now able to show that it is not possible to define a $\mathbb{D}$–sorting procedure. In fact, observe that, in the operation choice rule 2, nothing is said when $\mathbb{X} = \mathbb{D}$: in this case, it is not possible to decide a priori, according to the local operation choice condition, which input operation must be performed. We show this in the proof of the following theorem.

Theorem 3.9.1. *Every possible procedure $\mathbb{D}$–OperationChoice defines a procedure $\mathbb{D}$–CreateExecute which does not sort all the possible $\mathbb{D}$–sortable permutations. Hence, it is not possible to define a $\mathbb{D}$–sorting procedure.*

Proof. Suppose that, at a fixed state of the sorting process, the deque contains two elements. Without loss of generality, we can suppose that these elements are in increasing order. The arrival of an element smaller than the two inner

ones gives rise to a situation for which no insertion choice rule is provided: we can insert the new input element either through the left or the right gate. Both cases are possible according to corollary 3.7.3: in the first case the inner sequence is still increasing, while in the second case it becomes unimodal. We will call these two possible alternatives *option A* and *option B*, respectively (see figure 3.6).

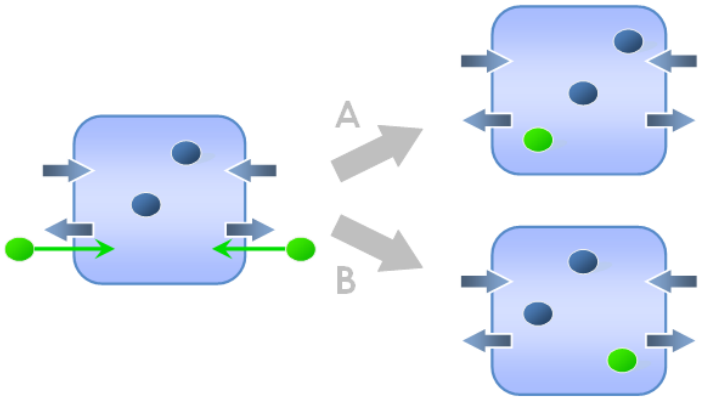


Figure 3.6. *The two equally admissible insertion alternatives.*

Unfortunately, no one of these options guarantees to gain the final sorted sequence, since each one of them fails for particular $\mathbb{D}$–sortable input permutations. Consider, for instance, the permutations $\sigma_1 = 541362$ and $\sigma_2 = 43251$, which can be sorted through $\mathbb{D}$ by the sequences $S_1 = II\bar{I}I\bar{O}\bar{I}IOOOOO$ and $S_2 = III\bar{I}IOOOOO$, respectively. After the insertion of the first two elements, both of them lead to the situation described above. If we choose option A the permutation σ_1 will not be sorted: after the insertion of 5 (I), 4 (I) and 1 (I), the element 3 must be inserted through the right gate ($\bar{I}$) and then the arrival of 6 forces to pull out 1 and 3 before the arrival of 2. On the other hand, option B is not able to sort σ_2: in fact, after the insertion of 4 (I), 3 (I) and 2 ($\bar{I}$), the element 5 forces to pull out 2 before the arrival of 1, and hence, again, the final output sequence will not be sorted. □

Chapter 4

Sorting algorithms

4.1 Sorting procedures and sorting algorithms

In the previous chapter we described the deterministic procedures which are able to sort all the $\mathbb{X}$–sortable permutations for a given device $\mathbb{X} \in \{\mathbb{S}, \mathbb{D}^{ir}, \mathbb{D}^{or}\}$. However, an $\mathbb{X}$–sorting procedure cannot be considered as a *sorting algorithm*, since it is not able to sort the permutations belonging to $\Sigma \setminus Sort(\mathbb{X})$.

In this section, we will show how to use the $\mathbb{X}$–sorting procedures to define some new sorting algorithms. As we will show in the following, if we want to sort any possible input permutation σ we must iterate an $\mathbb{X}$–sorting procedure a certain number of times, depending on the length of σ. In order to prove this, we primarily observe that every $\mathbb{X}$–sorting procedure puts the largest element of the input permutation in the rightmost position of the output. This is stated in the following lemma.

Lemma 4.1.1. *Let σ and $\mathsf{X}(\sigma)$ be, respectively, the input and output permutation of an $\mathbb{X}$–sorting procedure X. Then*

$$\mathsf{X}(\sigma) = \tau\ n,$$

where $\tau \in \Sigma_{n-1}$.

Moreover, it is not difficult to check that if the last $n-k$ elements (from the k+1–th to the n–th) of the input permutation are already in the their

correct position, after performing X even the k–th entry will be in its correct position.

Lemma 4.1.2. *Let $\sigma = \sigma_1 \dots \sigma_n$ and $\mathsf{X}(\sigma)$ be, respectively, the input and output permutation of an $\mathbb{X}$–sorting procedure X. If $\sigma_i = i$ for all i such that $k{+}1 \leqslant i \leqslant n$, then*

$$\mathsf{X}(\sigma_1\, \sigma_2\, \dots\, \sigma_k\; k{+}1\; \dots\; n) = \tau\; k\; k{+}1\; \dots\; n,$$

where $\tau \in \Sigma_{k-1}$.

By induction, using the previous lemmas, it is now clear that any input permutation σ is definitely sorted by $|\sigma| - 1$ iterations of an $\mathbb{X}$–sorting procedure.

Proposition 4.1.3. *Let X be an $\mathbb{X}$–sorting procedure. Then*

$$\mathsf{X}^{n-1}(\sigma) = id, \qquad \forall\, \sigma \in \Sigma_n.$$

This allows us to define the following $\mathbb{X}$–*sorting algorithm* (where, obviously, the final output is $\sigma = id$).

$\mathbb{X}$–**sorting Algorithm** : $\sigma \to \sigma$
  $n \leftarrow length(\sigma)$
  for i **from** 1 **to** $n - 1$
    $\sigma \leftarrow \mathsf{X}(\sigma)$
  end for

We remark that $n-1$ iterations are not necessary for almost all the permutations of Σ_n. For many input permutations, in fact, it is sufficient to iterate the procedure a smaller number of times, which is far from $n - 1$. Hence, it could be convenient to modify the previous algorithm to the following one.

$\mathbb{X}$–**sorting Algorithm** : $\sigma \to \sigma$
  while $\sigma \neq id$
    $\sigma \leftarrow \mathsf{X}(\sigma)$
  end while

Observe that in this second algorithm the while loop cannot fall into an infinite loop, as guaranteed by proposition 4.1.3. Moreover, the computational cost of the while condition can be reduced if we keep trace of the swaps performed: when no swaps occur in a whole iteration of X, the permutation is sorted.

Another different improvement of the latter algorithm is possible if we are able to determine the minimum number of iterations that allows to sort all the permutations of fixed length n.

Definition 4.1.4. *The* sharp iteration number *of a procedure* X *for input permutations of length* n *is the integer* $k(n)$ *such that*

$$\forall\, \sigma \in \Sigma_n \quad \mathsf{X}^{k(n)}(\sigma) = id \quad \wedge \quad \exists\, \bar{\sigma} \in \Sigma_n :\ \mathsf{X}^{k(n)-1}(\bar{\sigma}) \neq id.$$

Obviously, by proposition 4.1.3 it is immediately proven that, for every procedure X, we have $k(n) \leqslant n-1$. For the stack sorting procedure, it is not difficult to show that $k(n) = n-1$.

Proposition 4.1.5. *The sharp iteration number of* S *is* $n-1$.

Proof. Take $\sigma = 234\ldots n1$ as input permutation. It is easy to check that, at each iteration of S, the element 1 swaps with the element to its left, while the others keep their position. Hence, we need exactly $n-1$ iterations of S in order to sort σ. □

As concerns the restricted deques, we do not have a definite answer. However, for small integers n, the values of $k(n)$ obtained through direct computations are much smaller than $n-1$.

4.2 The bubblesort procedure

In the previous section we presented the $\mathbb{X}$–sorting algorithm, which consists of the iteration of the procedure X a suitable number of times.

The same structure can be found in the very well-known *bubblesort* algorithm. In fact, this algorithm presents some features which are very similar to

the $\mathbb{X}$–sorting algorithms: as for these, for example, it consists of the iteration of a procedure, that we will denote by B. It is interesting, as we will do in section 4.6, to analyze the interaction between the procedure B, the stack procedure S and their dual versions.

Bubblesort Algorithm : $\sigma \to \sigma$

$n \leftarrow length(\sigma)$
for i **from** 1 **to** $n-1$
  $\sigma \leftarrow B(\sigma)$
end for

In a single iteration of the procedure B, the permutation is scanned from left to right, and two consecutive elements are swapped if the smaller follows the greater.

Procedure B : $\sigma \to \sigma$

for j **from** 1 **to** $n-1$
  if $\sigma[j] > \sigma[j+1]$ **then**
    $swap\,(\sigma[j], \sigma[j+1])$
  end if
end for

Example 4.2.1. If the input permutation is $\sigma = 51372846$, then three iterations of B are needed in order to sort σ. In fact, we have $B(\sigma) = 13527468$, $B^2(\sigma) = 13254678$ and, finally, $B^3(\sigma) = 12345678$.

Observe that the procedure B acts as the procedure S, with the additional restriction that at most two elements can be contained into the stack at each state of the sorting process.

A variant of the classic bubblesort procedure B is the *dual bubblesort procedure* $\tilde{B}$. As in B, this dual procedure swaps two consecutive elements if the smaller follows the greater. The difference is that $\tilde{B}$ scans the permutation from right to left.

Procedure $\tilde{B}$: $\sigma \to \sigma$

for j **from** $n-1$ **to** 1
  if $\sigma[j] > \sigma[j+1]$ **then**
    $swap\,(\sigma[j], \sigma[j+1])$
  end if
end for

Knuth [14] introduced $\tilde{B}$ in the definition of a slightly more efficient version of the bubblesort: the so called *cocktail shaker sort*. In this algorithm, that we give below, the procedures B and $\tilde{B}$ are alternated, and this reduces the average number of comparisons.

Cocktail Shaker Sort Algorithm : $\sigma \to \sigma$

$n \leftarrow length(\sigma)$
for i **from** 1 **to** $n-1$
  if $i \bmod 2 = 0$ **then**
    $\sigma \leftarrow B(\sigma)$
  else
    $\sigma \leftarrow \tilde{B}(\sigma)$
  end if
end for

4.3 Dual procedures

The action of the dual bubblesort procedure $\tilde{B}$ can be also described by making use of the reverse-complement operator $\rho = r \circ c$ (see definition 1.3.1). In fact, since

$$\rho(\sigma)_i = n + 1 - \sigma_{n+1-i},$$

it is easy to check that

$$\tilde{B} = \rho \circ B \circ \rho.$$

By analogy, for every $\mathbb{X}$–sorting procedure X we can define the *dual procedure*

$\tilde{X}$ as

$$\tilde{X} = \rho \circ X \circ \rho.$$

Hence, by using lemma 4.1.1, we get immediately the following result.

Lemma 4.3.1. *Let σ and $\tilde{X}(\sigma)$ be, respectively, the input and output permutation of a dual $\mathbb{X}$–sorting procedure $\tilde{X}$. Then*

$$\tilde{X}(\sigma) = 1\,\tau,$$

where τ is a word, with no repeated letters, over the alphabet $\{2, 3, \ldots, n\}$.

In this case, if the first k elements of the input permutation are already in their correct position, after performing $\tilde{X}$ even the $(k{+}1)$–th will be in its correct position.

Lemma 4.3.2. *Let $\sigma = \sigma_1 \ldots \sigma_n$ and $\tilde{X}(\sigma)$ be, respectively, the input and output permutation of a dual $\mathbb{X}$–sorting procedure $\tilde{X}$. If $\sigma_i = i$ for all i such that $1 \leqslant i \leqslant k$, then*

$$\tilde{X}(1\,2\,\ldots\,k\,\sigma_{k+1}\,\sigma_{k+2}\,\ldots\,\sigma_n) = 1\,2\,\ldots\,k\;\;k{+}1\;\tau,$$

where τ is a word, with no repeated letters, over the alphabet $\{k{+}2, \ldots, n\}$.

Hence, as for the classical procedures, by induction on the previous lemmas we can state the following proposition.

Proposition 4.3.3. *Let $\tilde{X}$ be a dual $\mathbb{X}$–sorting procedure. Then*

$$\tilde{X}^{n-1}(\sigma) = id, \qquad \forall\, \sigma \in \Sigma_n.$$

From now on, we will denote by $\mathcal{P}$ the set of procedures that we considered so far:

$$\mathcal{P} = \{B, S, D^{ir}, D^{or}, \tilde{B}, \tilde{S}, \tilde{D^{ir}}, \tilde{D^{or}}\}.$$

4.4 Hybrid algorithms

In the previous sections we exhibited some sorting algorithms which are implemented by iterating a given procedure, or alternating a procedure with its dual.

More generally, we can define many new *hybrid algorithms* by blending the previous procedures and their duals, in all possible ways. In particular, as guaranteed by lemmas 4.1.1 and 4.1.2 together with their dual versions 4.3.1 and 4.3.2, $n-1$ iterations of these procedures (in any combination) are able to sort every permutation of length n. In other terms, a hybrid algorithms can be used as a sorting algorithm, if we iterate the procedures a sufficient number of times.

Proposition 4.4.1. *Let P_1, P_2, ... P_{n-1} be a succession of procedures, where $P_i \in \mathcal{P}$. Then*

$$P_1 \circ P_2 \circ \cdots \circ P_{n-1}(\sigma) = id, \qquad \forall\, \sigma \in \Sigma_n.$$

It is very interesting to analyze the joint action of the procedures of a hybrid algorithm even for a small number of iterations, which do not guarantee, in general, to get the final sorted permutation. In the following, we will focus mainly on two problems. In the rest of this section and in the subsequent ones, we will prove some commutation properties among the procedures of $\mathcal{P}$. After this, in section 4.7 we will found a pattern avoidance description of the permutations sorted by a fixed number of iterations of B and $\tilde{B}$.

As one might expect, if we mix together some iterations of a procedure with some iterations of a completely different one, the action of the resulting hybrid algorithm depends, in general, on the order we have used to perform the procedures. Despite this, in the following sections we will prove that - quite surprisingly - the output of some particular combinations of B, S and their duals depends only on the number of iterations of each procedure, and not on their relative order. Direct computations suggest that similar commutation properties may hold even for B combined with $\tilde{D^{ir}}$, and B with $\tilde{D^{or}}$. As far as we know, these commutation properties have not yet been proved, and hence they could be an interesting problem to explore.

In table 4.1 we summarize the results about the commutativity of each pair of procedures. In particular, a counterexample is provided for the non-commutative procedures. Moreover, as can be checked on table 4.1, the following proposition is immediately proven.

Proposition 4.4.2. *Let* $P_1, P_2 \in \mathcal{P}$. *Then*

$$P_1 \circ P_2 = P_2 \circ P_1 \iff \tilde{P}_1 \circ \tilde{P}_2 = \tilde{P}_2 \circ \tilde{P}_1.$$

	B	$\tilde{B}$	S	$\tilde{S}$	D^{ir}	$\tilde{D}^{ir}$	D^{or}
$\tilde{B}$	Yes (Theorem 4.6.6)						
S	No (4231)	Yes (Theorem 4.6.7)					
$\tilde{S}$	Yes (Theorem 4.6.7)	No (4231)	No (4312)				
D^{ir}	No (34251)	? (Open)	No (43251)	No (53421)			
$\tilde{D}^{ir}$	? (Open)	No (51423)	No (54231)	No (51432)	No (5463271)		
D^{or}	No (53241)	? (Open)	No (53241)	No (53421)	No (634251)	No (645132)	
$\tilde{D}^{or}$	? (Open)	No (52431)	No (54231)	No (52431)	No (546231)	No (625341)	No (465231)

Table 4.1. *Commutation and non-commutation properties among the procedures. For the non-commutative pairs of procedures, one of the smallest counterexamples is given.*

4.5 Bubblesort, stacksort and their duals

In the previous sections we defined the procedures B and S by their implementation. It is also possible to give a recursive definition of the previous procedures: if $\sigma = \alpha n \beta$, where n is the greatest element of σ, then

$$B(\alpha\, n\, \beta) = B(\alpha)\, \beta\, n \qquad \text{and} \qquad S(\alpha\, n\, \beta) = S(\alpha)\, S(\beta)\, n,$$

while, for the dual procedures, we have

$$\tilde{B}(\alpha\, 1\, \beta) = 1\, \alpha\, \tilde{B}(\beta) \qquad \text{and} \qquad \tilde{S}(\alpha\, 1\, \beta) = 1\, \tilde{S}(\alpha)\, \tilde{S}(\beta).$$

We can give equivalent definitions of B and S also referring to the local left-to-right maxima. We recall that an element of a permutation σ is a *left-to-right maximum* if it is greater than all the previous elements. Hence, the set of all left-to-right maxima of σ is

$$Max_{lr}(\sigma) = \{\sigma_i : \sigma_i > \sigma_j,\ \forall\, j < i\}.$$

We can write σ highlighting its left-to-right maxima $M_1, M_2, \ldots, M_k$, where $M_k = n$, in the following way:

$$\sigma = M_1\, u_1\, M_2\, u_2 \ldots M_k\, u_k = (M_\alpha\, u_\alpha)_\alpha,$$

where u_α are (possibly empty) words. Hence,

$$B((M_\alpha\, u_\alpha)_\alpha) = (u_\alpha\, M_\alpha)_\alpha \qquad \text{and} \qquad S((M_\alpha\, u_\alpha)_\alpha) = (S(u_\alpha)\, M_\alpha)_\alpha. \tag{4.1}$$

We can give analogous definitions for $\tilde{B}$ and $\tilde{S}$ referring to the local right-to-left minima. An element of σ is a *right-to-left minimum* if it is smaller than all the following elements:

$$Min_{rl}(\sigma) = \{\sigma_i : \sigma_i < \sigma_j,\ \forall\, j > i\}.$$

Observe that

$$\rho(Max_{lr}(\sigma)) = Min_{rl}(\rho(\sigma)) \quad \text{and} \quad \rho(Min_{rl}(\sigma)) = Max_{lr}(\rho(\sigma)). \tag{4.2}$$

Writing σ as

$$\sigma = v_h\, m_h \ldots v_2\, m_2\, v_1\, m_1 = (v_\beta\, m_\beta)_\beta,$$

it follows that

$$\tilde{B}((v_\beta\, m_\beta)_\beta) = (m_\beta\, v_\beta)_\beta \qquad \text{and} \qquad \tilde{S}((v_\beta\, m_\beta)_\beta) = (m_\beta\, \tilde{S}(v_\beta))_\beta. \tag{4.3}$$

4.6 Commutation properties among bubblesort, stacksort and their duals

From now on, all inequalities involving sequences are intended to hold for all the elements belonging to the sequences: for example, $\alpha < \beta$ means that every element of α is smaller than every element of β.

In order to prove the commutation properties between the previous procedures, we first give some simple results.

Remark 4.6.1. *If $\mathcal{M}$ is both a left-to-right maximum and a right-to-left minimum of a permutation $\sigma = \alpha\,\mathcal{M}\,\beta$ (α and β possibly empty), then*

(i) $\alpha < \mathcal{M} < \beta$;

(ii) $\mathcal{M}$ *lies in the* $\mathcal{M}$*-th position of* σ*;*

(iii) $\mathcal{M}$ *immediately precedes a left-to-right maximum* M*;*

(iv) $\mathcal{M}$ *immediately follows a right-to-left minimum* m*;*

(v) the following relations hold:

$$\begin{aligned} B(\alpha\,\mathcal{M}\,\beta) &= B(\alpha)\,\mathcal{M}\,B(\beta), \qquad & \tilde{B}(\alpha\,\mathcal{M}\,\beta) &= \tilde{B}(\alpha)\,\mathcal{M}\,\tilde{B}(\beta) \\ S(\alpha\,\mathcal{M}\,\beta) &= S(\alpha)\,\mathcal{M}\,S(\beta), \qquad & \tilde{S}(\alpha\,\mathcal{M}\,\beta) &= \tilde{S}(\alpha)\,\mathcal{M}\,\tilde{S}(\beta). \end{aligned}$$

Proof. Statements (i), (ii), (iii) and (iv) are straightforward. Concerning (v), observe that σ can be written as $\sigma = \alpha'\,m\,\mathcal{M}\,M\,\beta'$, and hence the application of B, S and their duals - see (4.1) and (4.3) - does not affect the relative order between α, $\mathcal{M}$ and β. □

Lemma 4.6.2. *For every permutation σ the following relations hold:*

(i) $Max_{lr}(\sigma) \subset Max_{lr}(B(\sigma))$ *and* $Min_{rl}(\sigma) \subset Min_{rl}(B(\sigma))$*;*

(ii) $Max_{lr}(\sigma) \subset Max_{lr}(S(\sigma))$ *and* $Min_{rl}(\sigma) \subset Min_{rl}(S(\sigma))$*;*

(iii) $Max_{lr}(\sigma) \subset Max_{lr}(\tilde{B}(\sigma))$ *and* $Min_{rl}(\sigma) \subset Min_{rl}(\tilde{B}(\sigma))$*;*

(iv) $Max_{lr}(\sigma) \subset Max_{lr}(\tilde{S}(\sigma))$ *and* $Min_{rl}(\sigma) \subset Min_{rl}(\tilde{S}(\sigma))$.

Proof. (i) Let $\sigma = (M_\alpha\, u_\alpha)_\alpha$, whence $B(\sigma) = (u_\alpha\, M_\alpha)_\alpha$. Every $M_i \in Max_{lr}(\sigma)$ is preceded, in $B(\sigma)$, by the same elements as in σ and by $u_i < M_i$: this yields $M_i \in Max_{lr}(B(\sigma))$.

Let now $m \in Min_{rl}(\sigma)$. If $m \in u_i$ for some i, in $B(\sigma)$ it is followed by the same elements as in σ and by $M_i > m$; hence, $m \in Min_{rl}(\sigma)$ if and only if $m \in Min_{rl}(B(\sigma))$. Otherwise, if $m = M_i$ the elements following m in $B(\sigma)$ follow m also in σ, and thus $m \in Min_{rl}(B(\sigma))$.

(ii) Let $\sigma = (M_\alpha\, u_\alpha)_\alpha$ and $S(\sigma) = (S(u_\alpha)\, M_\alpha)_\alpha$. Every $M_i \in Max_{lr}(\sigma)$ is preceded, in $S(\sigma)$, by the same elements as in σ and by $S(u_i) < M_i$, and hence $M_i \in Max_{lr}(S(\sigma))$. Now, let $m \in Min_{rl}(\sigma)$. When we push m into the stack, the smaller elements inside are popped out and, immediately after, also m is popped out by the following element, which is greater. After this, the elements in the stack and those that are waiting to enter are all greater than m, and hence $m \in Min_{rl}(S(\sigma))$.

The proofs of (iii) and (iv) are straightforward by using (i), (ii) and relations (4.2).

□

When the bubblesort (or the stacksort) acts on σ, some new right-to-left minima may arise. In the following lemma we prove that each of them lies immediately to the right of a (new or old) right-to-left minimum; of course, a similar (reversed) result holds for the dual procedures $\tilde{B}$ and $\tilde{S}$.

Lemma 4.6.3. *For every permutation σ the following relations hold:*

(i) in $B(\sigma)$, every $m \in Min_{rl}(B(\sigma)) \setminus Min_{rl}(\sigma)$ immediately follows an element $m' \in Min_{rl}(B(\sigma))$;

(ii) in $S(\sigma)$, every $m \in Min_{rl}(S(\sigma)) \setminus Min_{rl}(\sigma)$ immediately follows an element $m' \in Min_{rl}(S(\sigma))$;

(iii) in $\tilde{B}(\sigma)$, every $M \in Max_{lr}(\tilde{B}(\sigma)) \setminus Max_{lr}(\sigma)$ immediately precedes an element $M' \in Max_{lr}(\tilde{B}(\sigma))$;

(iv) in $\tilde{S}(\sigma)$, every $M \in Max_{lr}(\tilde{S}(\sigma)) \setminus Max_{lr}(\sigma)$ immediately precedes an element $M' \in Max_{lr}(\tilde{S}(\sigma))$.

Proof. (i) Let $\sigma = (M_\alpha\, u_\alpha)_\alpha$ and $B(\sigma) = (u_\alpha\, M_\alpha)_\alpha$. The considerations made in the proof of case (i) of 4.6.2 imply that, if $m \in Min_{rl}(B(\sigma)) \setminus Min_{rl}(\sigma)$, then necessarily $m \in Max_{lr}(\sigma)$ and thus, by lemma 4.6.2, $m \in Max_{lr}(B(\sigma))$. Hence, m is both a left-to-right maximum and a right-to-left minimum of $B(\sigma)$ and then (see remark 4.6.1) it is preceded by a right-to-left minimum.

(ii) We focus on two consecutive right-to-left minima m_{i+1} and m_i of σ, where $m_{i+1} < m_i$. Thus, we can write $\sigma = u\, m_{i+1}\, v\, m_i\, w$, where u, v and w are (possibly empty) words, with $v, w > m_i$. After the stacksort, every possible "new" right-to-left minimum between m_{i+1} and m_i, i.e. every possible $m \in Min_{rl}(S(\sigma)) \setminus Min_{rl}(\sigma)$, with $m_{i+1} < m < m_i$, must belong to u and must be popped out of the stack after m_{i+1} and before m_i. This holds if and only if, into the stack, when m_{i+1} arrives there is a nonempty set $u' \subset u$, $m_{i+1} < u' < m_i$. In this case, m_{i+1} and all the elements of u' are popped out in increasing order when the first element of v is pushed (or when m_i is, if $v = \varnothing$), and they become all (consecutive) right-to-left minima of $S(\sigma)$.

The proofs of (iii) and (iv) are straightforward by using (i), (ii) and relations (4.2).

□

We now introduce the notion of *local sorting operator*, which will be useful in the proof of the next theorem. Let $\sigma = (M_\alpha\, u_\alpha)_\alpha = (v_\beta\, m_\beta)_\beta$, where M_i and m_j are the left-to-right maxima and right-to-left minima of σ, respectively, and define

$$B_{M_i}(\sigma) = M_1\, u_1 \dots M_{i-1}\, u_{i-1}\, u_i\, M_i\, M_{i+1}\, u_{i+1} \dots M_k\, u_k$$

and

$$\tilde{B}_{m_j}(\sigma) = v_h\, m_h \dots v_{j+1}\, m_{j+1}\, m_j\, v_j\, v_{j-1}\, m_{j-1} \dots v_1\, m_1.$$

Obviously,

$$B(\sigma) = B_{M_1} \circ B_{M_2} \circ \cdots \circ B_{M_k}(\sigma) \tag{4.4}$$

and

$$\tilde{B}(\sigma) = \tilde{B}_{m_1} \circ \tilde{B}_{m_2} \circ \cdots \circ \tilde{B}_{m_h}(\sigma). \tag{4.5}$$

Remark 4.6.4. The dependence of (4.4) and (4.5) on the local maxima and minima might cause a little trouble if we don't have any information about

them. For example, if we want to rewrite $B \circ \tilde{B}(\sigma)$ with the above notations, we have to know both the right-to-left minima of σ and the left-to-right maxima of $\tilde{B}(\sigma)$. Despite this, every new left-to-right maximum M of $\tilde{B}(\sigma)$ is immediately to the left of another left-to-right maximum (see lemma 4.6.3), and hence its corresponding local bubblesort B_M does not perform any interchange ($B_M = id$). More generally:

- $M \in Max_{lr}(\sigma)$ immediately precedes $M' \in Max_{lr}(\sigma) \Longrightarrow B_M = id$;
- $m \in Min_{rl}(\sigma)$ immediately follows $m' \in Min_{rl}(\sigma) \Longrightarrow \tilde{B}_m = id$.

Therefore, in order to rewrite $B \circ \tilde{B}$ or $\tilde{B} \circ B$ via the local sorting operators, we only need to know $Max_{lr}(\sigma)$ and $Min_{rl}(\sigma)$. This yields the following lemma.

Lemma 4.6.5. *Let* $\sigma = (M_\alpha\, u_\alpha)_\alpha = (v_\beta\, m_\beta)_\beta$. *Then*

$$\begin{array}{rcll} B \circ \tilde{B} = \tilde{B} \circ B & \Longleftrightarrow & B_{M_i} \circ \tilde{B}_{m_j} = \tilde{B}_{m_j} \circ B_{M_i} & \forall\, i,\, j\,; \\ S \circ \tilde{B} = \tilde{B} \circ S & \Longleftrightarrow & S \circ \tilde{B}_{m_j} = \tilde{B}_{m_j} \circ S & \forall\, j. \end{array}$$

Theorem 4.6.6. *The following commutation property holds:*

$$B \circ \tilde{B} = \tilde{B} \circ B.$$

Proof. Let $\sigma = (M_\alpha\, u_\alpha)_\alpha = (v_\beta\, m_\beta)_\beta$, and choose M_i and m_j. By lemma 4.6.5, it is sufficient to prove the commutativity of the local sorting operators B_{M_i} and $\tilde{B}_{m_j}$.

If $M_i = m_j$, $M_i = m_{j+1}$ or $M_{i+1} = m_j$, then at least one of M_i and m_j is simultaneously a left-to-right maximum and a right-to-left minimum, and then (see remarks (4.6.1) and (4.6.4)) either B_{M_i} or $\tilde{B}_{m_j}$ is the identity map.

Now, suppose that $M_i \neq m_j$, $M_i \neq m_{j+1}$ and $M_{i+1} \neq m_j$. If $M_i\, u_i \cap v_j\, m_j = \varnothing$, or $M_i\, u_i \subset v_j$, or even $v_j\, m_j \subset u_i$, the commutative property follows immediately from the fact that the interchanges operated by B_{M_i} and $\tilde{B}_{m_j}$ do not cross each other. In the only two remaining cases, namely, $M_i\, w_1\, m_{j+1}\, w_2\, M_{i+1}\, w_3\, m_j$ and $m_{j+1}\, w_1\, M_i\, w_2\, m_j\, w_3\, M_{i+1}$, where the w_k are (possibly empty) words, the commutativity can be directly checked. □

Theorem 4.6.7. *The following commutation property holds:*

$$S \circ \tilde{B} = \tilde{B} \circ S.$$

Hence, as a direct consequence, $\tilde{S} \circ B = B \circ \tilde{S}$.

Proof. Choosing a right-to-left minimum m_j, by lemma 4.6.5 it is sufficient to prove that S and $\tilde{B}_{m_j}$ commute.

In order to describe the action of S in progress we can use the notation

$$output \,\langle inside] \, input,$$

where $\langle$ is the open gate of the stack. For instance,

$$S(26314875) = \langle 2] \, 6314875 = 2 \langle 6] \, 314875 = 2 \langle 36] \, 14875 = 2 \langle 136] \, 4875 =$$

$$= 213 \langle 46] \, 875 = 21346 \langle 8] \, 75 = 21346 \langle 78] \, 5 = 21346 \langle 578] = 21346578.$$

When we write $x \langle y]$ (with empty input) we refer unambiguously to the last passage of S, just before the final emptying of the stack.

Now, let $\sigma = u \, v_j \, m_j \, w$, where u and w are, respectively, the words $u = v_h \, m_h \ldots v_{j+1} \, m_{j+1}$ and $w = v_{j-1} \, m_{j-1} \ldots v_1 \, m_1$. Obviously, $v_j > m_j$ and $w > m_j$. Let $S(u) = u' \langle u'' \, u''']$, where $u'' < m_j$ and $u''' > m_j$, and let t', t'' and q be the sequences such that $\langle u'''] \, v_j = t' \, \langle t'']$ and $\langle t''] \, w = q$. Now, we have

$$\begin{aligned} S \circ \tilde{B}_{m_j}(\sigma) &= S \circ \tilde{B}_{m_j}(u \, v_j \, m_j \, w) = S(u \, m_j \, v_j \, w) = u' \langle u'' \, u'''] \, m_j \, v_j \, w = \\ &= u'u'' \langle m_j \, u'''] \, v_j \, w = u'u''m_j \, t' \langle t''] \, w = u'u''m_j \, t' \, q \end{aligned}$$

and

$$\begin{aligned} \tilde{B}_{m_j} \circ S(\sigma) &= \tilde{B}_{m_j} \circ S(u \, v_j \, m_j \, w) = \tilde{B}_{m_j}(u' \langle u'' \, u'''] \, v_j \, m_j \, w) = \\ &= \tilde{B}_{m_j}(u'u'' \, t' \langle t''] \, m_j \, w) = \tilde{B}_{m_j}(u'u'' \, t' \langle m_j \, t''] \, w) = \\ &= \tilde{B}_{m_j}(u'u'' \, t' \, m_j \, q) = u'u''m_j \, t' \, q. \end{aligned}$$

Hence, $S \circ \tilde{B} = \tilde{B} \circ S$, and then, by proposition 4.4.2, we get $\tilde{S} \circ B = B \circ \tilde{S}$.

□

Remark 4.6.8. *We can rewrite the commutation properties stated in the previous theorems by using* ρ*:*

(i) $(B \circ \rho)^2 = (\rho \circ B)^2$*;*

(ii) $S \circ \rho \circ B \circ \rho = \rho \circ B \circ \rho \circ S$ *(and hence* $\rho \circ S \circ \rho \circ B = B \circ \rho \circ S \circ \rho$*).*

4.7 Sorting algorithms and permutation classes

If A is any algorithm, we denote by

$$Sort(A) = \{\sigma \mid A(\sigma) = 12\dots n\}$$

the set of permutations sorted by A. Obviously, when A consists of only one iteration of an $\mathbb{X}$–sorting procedure X, it immediately follows that

$$Sort(\mathbb{X}) = Sort(X).$$

In section 3.1 we showed that, for every procedure X, the set $Sort(X)$ is a permutation class. This property holds even for the set of permutations which can be sorted by one iteration of the bubblesort: this was proved recently by Albert et al. [1], who showed that

$$Sort(B) = Av(231, 321). \tag{4.6}$$

Moreover, the same authors proved that

$$Sort(S \circ B) = Av(2341, 2431, 3241, 4231),$$

and, as a generalization of (4.6), that

$$Sort(B^h) = Av(\Gamma_{h+2}), \tag{4.7}$$

where

$$\Gamma_{h+2} = \{\tau \in \Sigma_{h+2} \mid \tau_{h+2} = 1\}.$$

In other terms, the permutations sorted by h iterations of B are exactly those that avoid all the patterns of length $h + 2$ whose final term is 1.

Very recently, Barnabei et al. [3] showed that

$$Sort(\tilde{B} \circ B) = Av(3412, 3421, 4312, 4321).$$

The problem of determining whether a set of sortable permutations is a permutation class is nowadays widely investigated (see e.g. [2]). One of the most well-known results that involves an $\mathbb{X}$–sorting procedure was found by

West [23] for S^2; more recently, Úlfarsson [22] analyzed the S^3-sortable permutations. However, the problem remains open for the other stack cases (S^h, $h \geqslant 4$), for the restricted deques procedures and for all the possible combinations of different procedures of $\mathcal{P}$.

Regarding this, we remark that it is not true, in general, that the permutations sorted by an algorithm A are exactly those that avoid all the smallest $A-$unsortable permutations. More precisely, if

$$\ell_A = \min\{|\tau| : A(\tau) \neq id\} \quad \text{and} \quad \Pi_A = \{\pi : A(\pi) \neq id \wedge |\pi| = \ell_A\},$$

it is not true, in general, that

$$Sort(A) = Av(\Pi_A).$$

West's results [23] are an evident counterexample to the previous relation. In fact, West proves that the permutations sorted by two iterations of stacksort are $Av(2341, 3\bar{5}241)$, where $3\bar{5}241$ denotes the patterns 3241 which are not part of a pattern 35241. Since $\Pi_{S^2} = \{2341, 3241\}$, this implies that $Sort(S^2) \neq Av(2341, 3241)$.

In the following, we solve the previous problem for the hybrid algorithm $B^h \circ \tilde{B}^k$. More precisely, by making use of the commutation properties proved in theorem 4.6.6, we prove that the set of permutations sorted by some iterations of B and some of its dual $\tilde{B}$ can be expressed in terms of pattern avoidance. In order to describe the patterns involved we need the following definition.

Definition 4.7.1. *Given a permutation $\sigma = \sigma_1\sigma_2\ldots\sigma_n$ and n permutations $\alpha_1, \alpha_2, \ldots, \alpha_n$, the* inflation *of σ by $\alpha_1, \alpha_2, \ldots, \alpha_n$ (denoted by $\sigma[\alpha_1, \alpha_2, \ldots, \alpha_n]$) is the permutation of length $\ell = \sum_{i=1}^{n} |\alpha_i|$ which is obtained by replacing each element σ_i with a permutation τ_i such that:*

- *τ_i is order isomorphic to α_i;*
- *$\tau_i < \tau_j \iff \sigma_i < \sigma_j, \quad \forall\, i, j = 1, \ldots, n.$*

Example 4.7.2. The inflation of $\sigma = 312$ by $\alpha_1 = 21$, $\alpha_2 = 213$ and $\alpha_3 = 132$ is $312[21, 213, 132] = 87213465$.

We denote by

$$\sigma\,[\![\ell_1,\ldots,\ell_n]\!] = \{\sigma\,[\alpha_1,\ldots,\alpha_n] : |\alpha_i| = \ell_i,\ \forall\, i = 1,\ldots,n\}$$

the set of all possible inflations of σ by n permutations $\alpha_1, \alpha_2, \ldots, \alpha_n$ of fixed lengths $\ell_1, \ell_2, \ldots, \ell_n$. For instance, $231\,[\![2,1,2]\!] = \{34512, 34521, 43512, 43521\}$.

Observe that (4.7) can be described in terms of inflations as follows:

$$Sort(B^h) = Av(21\,[\![h+1,1]\!]).$$

Lemma 4.7.3. *For every $h, k \geqslant 1$ the following equivalences hold:*

$$\begin{array}{ccc} & \sigma \in Av(21\,[\![h+1,k+1]\!]) & \\ & \overset{(1)}{\Rightleftharpoons} \quad \overset{(2)}{\Rightleftharpoons} & \\ B(\sigma) \in Av(21\,[\![h,k+1]\!]) & \underset{(3)}{\Longleftrightarrow} & \tilde{B}(\sigma) \in Av(21\,[\![h+1,k]\!]) \end{array}$$

Proof. We prove only equivalence (1), since (2) can be proved analogously and (3) follows from (1) and (2). For convenience, we will equivalently show that σ contains a pattern $\tau \in 21\,[\![h+1,k+1]\!]$ if and only if $B(\sigma)$ contains $\tau' \in 21\,[\![h,k+1]\!]$.

Let $\sigma = (M_\alpha\, u_\alpha)_\alpha$ contain $\tau \in 21\,[\![h+1,k+1]\!]$. Observe that the last $k+1$ elements of τ are not left-to-right maxima of σ, and thus they belong to some of the u_α. Hence, in $B(\sigma) = (u_\alpha\, M_\alpha)_\alpha$ their relative order is preserved, and the first of them (i.e. $\tau_{h+2} \in u_i$) follows the same elements than in σ, except for M_i. This implies that $B(\sigma)$ contains a subsequence $\tau' \in 21\,[\![h,k+1]\!]$. The proof of the converse is analogous.

□

Theorem 4.7.4. *The set of permutations sorted by h iterations of B and k iterations of $\tilde{B}$ ($h, k \geqslant 0$), performed in any order, is the set*

$$Sort(B^h \circ \tilde{B}^k) = Av(21\,[\![h+1,k+1]\!]).$$

Proof. Applying lemma 4.7.3 h times for B and k for $\tilde{B}$, we obtain that

$$\sigma \in Av(21\,[\![h+1,k+1]\!]) \iff B^h \circ \tilde{B}^k(\sigma) \in Av(21) = \{id\}.$$

□

Chapter 5

Lattice paths

5.1 Enumeration of $\mathbb{X}-$sortable permutations

The first enumeration of $\mathbb{X}-$computable permutations was given by Knuth [13], who showed that the stack computable permutations are counted by the Catalan numbers (see section 1.5)

$$C_n = \frac{1}{n+1}\binom{2n}{n}.$$

Moreover, Knuth showed that the permutations computed by the restricted deques are counted by the *Schröder numbers* $(S_n)_n$ (sometimes called *large Schröder numbers*), which are defined by the following recurrence relation:

$$\begin{cases} S_n = S_{n-1} + \sum_{i=0}^{n-1} S_i \, S_{n-1-i} \\ S_0 = 1 \end{cases}$$

To get this result, Knuth enumerated some particular sequences of $\mathcal{D}^{or}$ (he called them *admissible sequences*) and defined a bijection between these sequences and the permutations that can be computed through a restricted deque. We remark that the set of Knuth's admissible sequences coincides (up to slight modifications) with the set of representatives given by our $\mathbb{X}-$sorting procedures, that we describe in section 5.4.

Obviously, the enumeration of $\mathbb{X}$–computable permutations immediately gives the number of $\mathbb{X}$–sortable ones. In fact, by relation (3.1), the sets $\mathcal{X}(id)_n$ and $Sort_n(\mathbb{X})$ are equipotent, and hence

$$|Sort_n(\mathbb{S})| = C_n, \qquad |Sort_n(\mathbb{D}^{ir})| = |Sort_n(\mathbb{D}^{or})| = S_n.$$

n	0	1	2	3	4	5	6	7	8
C_n	1	1	2	5	14	42	132	429	1430
S_n	1	2	6	22	90	394	1806	8558	41586

Table 5.1. *The first Catalan and Schröder numbers.*

Up to now, nothing is known on the enumeration of the set $Sort_n(\mathbb{D})$ of deque sortable permutations.

In the following sections we propose a bijective enumeration of the permutations sorted by the stack and the restricted deques. In particular, we will present a bijection that associates a lattice path to each permutation sorted by a given device.

5.2 Sortable permutations and lattice paths

In literature, many combinatorial objects have been enumerated by using bijections with lattice paths.

Definition 5.2.1. *A* lattice path *is any succession of consecutive steps in the $\mathbb{Z}^2$ lattice, where a step $V = (v_x, v_y)$ connects a point $A = (a_x, a_y)$ with the point $B = (a_x + v_x, a_y + v_y)$.*

The first lattice paths that we consider, which are strictly related to the stack sortable permutations, are *Dyck paths*.

Definition 5.2.2. *A* Dyck path *of length $2n$ is a lattice path from $(0, 0)$ to $(2n, 0)$ that consists of steps $U = (1, 1)$ and $D = (1, -1)$ and never goes below the $x-$axis.*

It is not difficult to show that the Catalan numbers enumerate the set $\mathscr{D}_{2n}$ of Dyck paths of length $2n$:

$$C_n = |\mathscr{D}_{2n}|.$$

Rogers and Shapiro [18] showed that the Schröder numbers count another class of lattice path, called *Schröder paths*.

Definition 5.2.3. *A* Schröder path *of length* $2n$ *is a lattice path from* $(0,0)$ *to* $(2n,0)$ *that consists of steps* $U=(1,1)$, $D=(1,-1)$, *and* $HH=(2,0)$ *and never goes below the* $x-$*axis.*

Denoting by $\mathscr{S}_{2n}$ the set of Schröder paths of length $2n$, we have

$$S_n = |\mathscr{S}_{2n}|.$$

In their work, Rogers and Shapiro also give a bijection between Knuth's admissible sequences and Schröder paths. In the following, we will use this bijection and the $\mathbb{X}$–sorting procedures to define a bijection between the restricted deque sortable permutations and the Schröder paths.

5.3 A bijection between $\mathbb{X}-$sortable permutations and lattice paths

The bijections we want to describe associate each $\mathbb{X}$–sortable permutation with a lattice path:

$$Sort(\mathbb{X}) \quad \longleftrightarrow \quad \mathscr{L}.$$

In particular, we will show that it is possible to define a bijection between stack sortable permutations of length n and Dyck paths of length $2n$

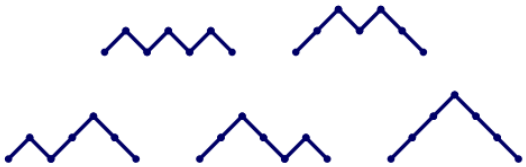
$$Sort_n(\mathbb{S}) \quad \longleftrightarrow \quad \mathscr{D}_{2n},$$

and two bijections between the restricted deque sortable permutations of length n and the Schröder paths of length $2(n-1)$:

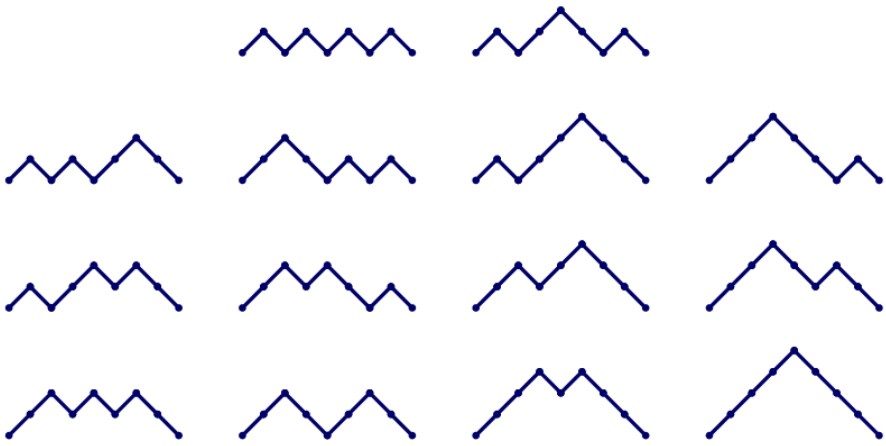
$$Sort_n(\mathbb{D}^{ir}) \quad \longleftrightarrow \quad \mathscr{S}_{2(n-1)} \qquad \text{and} \qquad Sort_n(\mathbb{D}^{or}) \quad \longleftrightarrow \quad \mathscr{S}_{2(n-1)}.$$

(a) $n = 1$

(b) $n = 2$



(c) $n = 3$

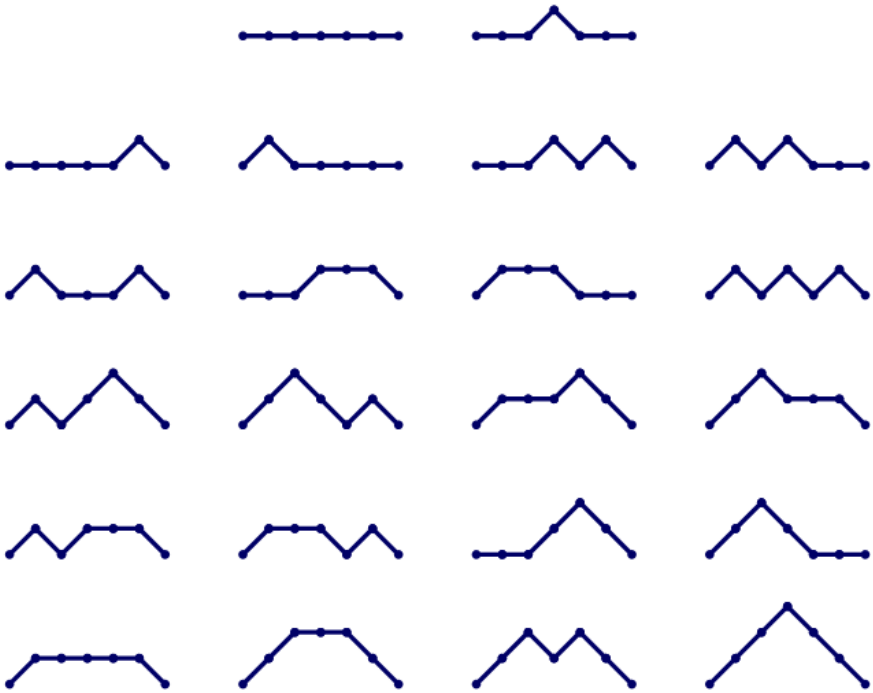


(d) $n = 4$

Figure 5.1. *Dyck paths* $\mathscr{D}_{2n}$.

(a) $n = 1$

(b) $n = 2$



(c) $n = 3$

Figure 5.2. *Schröder paths* $\mathscr{S}_{2n}$.

Recall that an $\mathbb{X}$–sorting procedure associates to a given $\mathbb{X}$–sortable permutation σ one of its possible $\mathbb{X}$–sorting sequences. In section 3.3 we described this action through the map $S_{\mathbb{X}}$, and we denoted by $S_{\sigma,\mathbb{X}}$ the representative chosen by the $\mathbb{X}$–sorting procedure.

Definition 5.3.1. *We denote by $\bar{\mathcal{X}}$ the set of representatives of the $\mathbb{X}$–sequences of any length, which are chosen by the $\mathbb{X}$–sorting procedures. In particular, we denote by $\bar{\mathcal{X}}_{2n}$ the set of representatives of length $2n$.*

It is convenient to describe the bijection between $Sort(\mathbb{X})$ and $\mathscr{L}$ by splitting it into two bijections $\varphi_{\mathbb{X}}$ and $\psi_{\mathbb{X}}$, one between $Sort(\mathbb{X})$ and $\bar{\mathcal{X}}$, the other one between $\bar{\mathcal{X}}$ and $\mathscr{L}$:

$$Sort(\mathbb{X}) \underset{\varphi_{\mathbb{X}}^{-1}}{\overset{\varphi_{\mathbb{X}}}{\rightleftarrows}} \bar{\mathcal{X}} \underset{\psi_{\mathbb{X}}^{-1}}{\overset{\psi_{\mathbb{X}}}{\rightleftarrows}} \mathscr{L}$$

5.4 A characterization of $\bar{\mathcal{X}}$

In this section, we give a characterization of the set of representatives $\bar{\mathcal{S}}$, $\bar{\mathcal{D}}^{ir}$ and $\bar{\mathcal{D}}^{or}$.

Proposition 5.4.1. *The set $\bar{\mathcal{S}}$ coincides with the set $\mathcal{S}$.*

Proof. To prove the proposition it is sufficient to show that distinct $\mathbb{S}$–sequences compute distinct permutations. Let $S, T \in \mathcal{S}$, $S \neq T$, and let k be the first integer such that $S_k \neq T_k$: hence, $S_k = I$ and $T_k = O$, or conversely. Obviously, before performing the k–th operation the output sequence is the same in both cases. Now, if we perform O the next element to be added to the output is the leftmost element i_1 of the inner sequence $i_1 \dots i_\ell$; conversely, if we perform I we insert a new element in the inner sequence which will precede i_1, and then the next element to be added to the output cannot be i_1. □

Proposition 5.4.2. *If $S \in \bar{\mathcal{D}}^{ir} \cup \bar{\mathcal{D}}^{or}$, then every prefix of S has more input operations than output ones (except when it coincides with the whole sequence S). In other terms, executing S never empties the device during the sorting process.*

Proof. As stated in the first operation choice rule (see section 3.8), the $\mathbb{X}$–sorting procedures always perform the input operations whenever this does not affect the monotonicity, or the unimodality, of the inner sequence (see corollary 3.7.3). Now, suppose that a device $\mathbb{X}$ contains only one element at a fixed state of the sorting process. If $\mathbb{X} = \mathbb{D}^{ir}$ we can always perform I since this does not affect the unimodality. Otherwise, if $\mathbb{X} = \mathbb{D}^{or}$ we can always perform either I or $\bar{I}$, depending on the new input element, without affecting the monotonicity of the inner sequence. □

Proposition 5.4.3. *The set $\bar{\mathcal{D}}^{ir}$ is the set of $\mathbb{D}^{ir}$–sequences that satisfy the following conditions:*

(i) the last operation is O;

(ii) no $\bar{O}$ is followed by I;

(iii) every prefix has strictly more input than output operations, except for the whole sequence.

Proof. Let $\hat{\mathcal{D}}^{ir}$ be the set of $\mathbb{D}^{ir}$–sequences that satisfy conditions (i), (ii) and (iii). If $S \in \bar{\mathcal{D}}^{ir}$, then (i) follows from the operation choice convention, (ii) from the first operation choice rule ($\bar{O}I$ has the same effect than $I\bar{O}$) and (iii) from proposition 5.4.2. Hence, $\bar{\mathcal{D}}^{ir} \subset \hat{\mathcal{D}}^{ir}$.

To prove the converse, we show that distinct sequences of $\hat{\mathcal{D}}^{ir}$ compute distinct permutations. Let $S, T \in \hat{\mathcal{D}}^{ir}$, $S \neq T$, and let k be the first integer such that $S_k \neq T_k$ (by condition (i), S_k and T_k cannot be the final operations of the sequences). Then, up to symmetries, three cases arise:

(a) $S_k = I$ and $T_k = O$;
(b) $S_k = I$ and $T_k = \bar{O}$;
(c) $S_k = O$ and $T_k = \bar{O}$.

Observe that, in all cases, at least one of the sequences has an output operation; hence, by condition (iii), before performing the k–th operation the device contains at least two elements. Case (a) coincides with the situation described for the stack in the proof of proposition 5.4.1, and hence the two sequences compute different permutations. In case (b), by condition (ii) the sequence

T cannot have $T_{k+1} = I$. This implies that, after $T_k = \bar{O}$, the sequence T may have a (possible empty) sequence of $\bar{O}$, which must be followed by an operation O. Hence, the permutation computed by S cannot coincide with the one computed by T, since S inserts a new element into the device. Finally, in case (c) the computed permutations may coincide if and only if there is only one element inside the device: but, as observed before, this is not the case. □

Proposition 5.4.4. *The set $\bar{\mathcal{D}}^{or}$ is the set of $\mathbb{D}^{or}$–sequences that satisfy the following conditions:*

(i) the first operation is I;

(ii) no O is followed by $\bar{I}$;

(iii) every prefix has strictly more input than output operations, except for the whole sequence.

Proof. Let $\hat{\mathcal{D}}^{or}$ be the set of $\mathbb{D}^{or}$–sequences that satisfy conditions (i), (ii) and (iii). If $S \in \bar{\mathcal{D}}^{or}$, then (i) follows from the operation choice convention, (ii) from the first operation choice rule ($O\bar{I}$ has the same effect than $\bar{I}O$) and (iii) from proposition 5.4.2. Hence, $\bar{\mathcal{D}}^{or} \subset \hat{\mathcal{D}}^{or}$.

To prove the converse, we show that distinct sequences of $\hat{\mathcal{D}}^{or}$ compute distinct permutations. Let $S, T \in \hat{\mathcal{D}}^{or}$, $S \neq T$, and let k be the first integer such that $S_k \neq T_k$. Observe that, by condition (i), S_k and T_k cannot be the first operations of the sequences, and hence, by condition (iii), the device contains at least one element when the k–th operation is performed. Then, up to symmetries, three cases arise:

(a) $S_k = I$ and $T_k = O$;
(b) $S_k = \bar{I}$ and $T_k = O$;
(c) $S_k = I$ and $T_k = \bar{I}$.

Case (a) coincides with the situation described for the stack in the proof of proposition 5.4.1, and hence the two sequences compute different permutations. In case (b) observe that, after $T_k = O$, the sequence T may have a (possible empty) sequence of O, which must be followed (by condition (ii)) by an operation I. Hence, the sequences S and T cannot compute the same

permutation, because S inserts the new input element through the right gate, while T inserts the same element through the left one. Since the deque has only one output gate, if we perform I the new inserted element will be added to the output before all the elements inside the device; conversely, if we perform $\bar{I}$ the new inserted element will be added to the output after the inner ones. The same holds for case (c), where the choice between I or $\bar{I}$ affects the computed permutation. □

5.5 The bijection φ

The map $\varphi_{\mathbb{X}}$

The map

$$\begin{array}{rccc} \varphi_{\mathbb{X}}\colon & Sort_n(\mathbb{X}) & \longrightarrow & \bar{\mathcal{X}}_{2n} \\ & \sigma & \longmapsto & S_{\sigma,\,\mathbb{X}} \end{array}$$

is completely described by the action of the $\mathbb{X}$–sorting procedures. In fact, it simply associates each $\mathbb{X}$–sortable permutation to the $\mathbb{X}$–sorting sequence given by the $\mathbb{X}$–sorting procedure X.

The map $\varphi_{\mathbb{X}}^{-1}$

Observe that each $\mathbb{X}$–sorting procedure can also be used as $\mathbb{X}$–*computing procedure*. In fact, as showed in proposition 3.1.2, the $\mathbb{X}$–sortable permutations are strictly related to the $\mathbb{X}$–computable ones: a permutation σ is sorted by an $\mathbb{X}$–sequence S if and only if its inverse σ^{-1} is computed by the same $\mathbb{X}$–sequence:

$$S(\sigma) = id \iff \sigma^{-1} = S(id).$$

Hence, in order to describe the map

$$\begin{array}{rccc} \varphi_{\mathbb{X}}^{-1}\colon & \bar{\mathcal{X}}_{2n} & \longrightarrow & Sort_n(\mathbb{X}) \\ & S & \longmapsto & \sigma_{\,S,\,\mathbb{X}} \end{array},$$

it is sufficient to perform S on the identity permutation, and then apply the inverse operator to get $\sigma_{\,S,\,\mathbb{X}}$.

5.6 The bijection ψ

In this section, for each device $\mathbb{X}$ we describe the bijection $\psi_{\mathbb{X}}$ and its inverse $\psi_{\mathbb{X}}^{-1}$. An example of the whole bijection $\psi_{\mathbb{X}} \circ \varphi_{\mathbb{X}}$ is given in figures 5.3, 5.4 and 5.5.

5.6.1 Stack

The maps $\psi_{\mathbb{S}}$ and $\psi_{\mathbb{S}}^{-1}$

$$\begin{array}{rccc} \psi_{\mathbb{S}}: & \bar{\mathcal{S}}_{2n} & \longrightarrow & \mathscr{D}_{2n} \\ & S & \longmapsto & \Delta_{S,\mathbb{S}}. \end{array} \qquad \begin{array}{rccc} \psi_{\mathbb{S}}^{-1}: & \mathscr{D}_{2n} & \longrightarrow & \bar{\mathcal{S}}_{2n} \\ & \Delta & \longmapsto & S_{\Delta,\mathbb{S}}. \end{array}$$

It is sufficient to replace each input/output operation with a path step (for $\psi_{\mathbb{S}}$), or each path step with an input/output operation (for $\psi_{\mathbb{S}}^{-1}$), through the following rule:

$$I \leftrightsquigarrow U, \qquad O \leftrightsquigarrow D.$$

5.6.2 Input-restricted deque

The map $\psi_{\mathbb{D}^{ir}}$

$$\begin{array}{rccc} \psi_{\mathbb{D}^{ir}}: & \bar{\mathcal{D}}_{2n}^{ir} & \longrightarrow & \mathscr{S}_{2(n-1)} \\ & S & \longmapsto & \Delta_{S,\mathbb{D}^{ir}}. \end{array}$$

The path $\Delta_{S,\mathbb{D}^{ir}}$ is obtained through the following steps.

① Delete the first (I) and last (O) operations of S.

② Link each input operation I with an output operation O or $\bar{O}$ (placed to the right of I), so that the resulting matching is a noncrossing partition of S.

③ Rename by I_1 each input operation matched with O, and with I_2 each input operation matched with $\bar{O}$.

④ Replace each entry of the new sequence according to the following rule:

$$I_1 \rightsquigarrow U, \qquad I_2 \rightsquigarrow HH, \qquad O \rightsquigarrow D, \qquad \bar{O} \rightsquigarrow \varnothing.$$

⑤ The resulting sequence gives the path $\Delta_{S,\,\mathbb{D}^{ir}}$.

The map $\psi^{-1}_{\mathbb{D}^{ir}}$

$$\begin{array}{rccc} \psi^{-1}_{\mathbb{D}^{ir}}: & \mathscr{S}_{2(n-1)} & \longrightarrow & \bar{\mathcal{D}}^{ir}_{2n} \\ & \Delta & \longmapsto & S_{\Delta,\,\mathbb{D}^{ir}} \end{array}$$

The following steps describe how to obtain the sequence $S_{\Delta,\,\mathbb{D}^{ir}}$.

① Replace each step of the path Δ according to the following rule:

$$U \rightsquigarrow I_1, \qquad HH \rightsquigarrow I_2, \qquad D \rightsquigarrow O.$$

② Cover the I_2 entries, and match each I_1 with an O (placed to the right of I_1) in order to obtain a noncrossing partition.

③ Unveil the I_2 entries, and add to the string an equal number of $\bar{O}$ (never before I_1 or I_2) so that the noncrossing matching can be completed.

④ Rename I_1 and I_2 by I, and add one I and one O, respectively, at the left and right end of the sequence.

⑤ The resulting sequence is $S_{\Delta,\,\mathbb{D}^{ir}}$.

5.6.3 Output-restricted deque

The map $\psi_{\mathbb{D}^{or}}$

$$\begin{array}{rccc} \psi_{\mathbb{D}^{or}}: & \bar{\mathcal{D}}^{or}_{2n} & \longrightarrow & \mathscr{S}_{2(n-1)} \\ & S & \longmapsto & \Delta_{S,\,\mathbb{D}^{or}}. \end{array}$$

The path $\Delta_{S,\,\mathbb{D}^{or}}$ is obtained through the following steps.

① Delete the first (I) and last (O) operations of S.

② Link each input operation I or $\bar{I}$ with an output operation O (placed to the right of I or $\bar{I}$), so that the resulting matching is a noncrossing partition of S.

③ Rename by O_1 each output operation matched with I, and with O_2 each output operation matched with $\bar{I}$.

④ Replace each entry of the new sequence through the following rule:

$$I \rightsquigarrow U, \qquad \bar{I} \rightsquigarrow \varnothing, \qquad O_1 \rightsquigarrow D, \qquad O_2 \rightsquigarrow HH.$$

⑤ The resulting sequence gives the path $\Delta_{S,\,\mathbb{D}^{or}}$.

The map $\psi^{-1}_{\mathbb{D}^{or}}$

$$\begin{array}{rccc} \psi^{-1}_{\mathbb{D}^{or}}: & \mathscr{S}_{2(n-1)} & \longrightarrow & \bar{\mathcal{D}}^{or}_{2n} \\ & \Delta & \longmapsto & S_{\Delta,\,\mathbb{D}^{or}} \end{array}$$

The following steps describe how to obtain the sequence $S_{\Delta,\,\mathbb{D}^{or}}$.

① Replace each step of the path Δ through the following rule:

$$U \rightsquigarrow I, \qquad D \rightsquigarrow O_1, \qquad HH \rightsquigarrow O_2.$$

② Cover the O_2 entries, and match each I with an O_1 (placed to the right of I) in order to obtain a noncrossing partition.

③ Unveil the O_2 entries, and add to the string an equal number of $\bar{I}$ (never after O_1 or O_2) so that the noncrossing matching can be completed.

④ Rename O_1 and O_2 by O, and add one I and one O, respectively, at the left and right end of the sequence.

⑤ The resulting sequence is $S_{\Delta,\,\mathbb{D}^{or}}$.

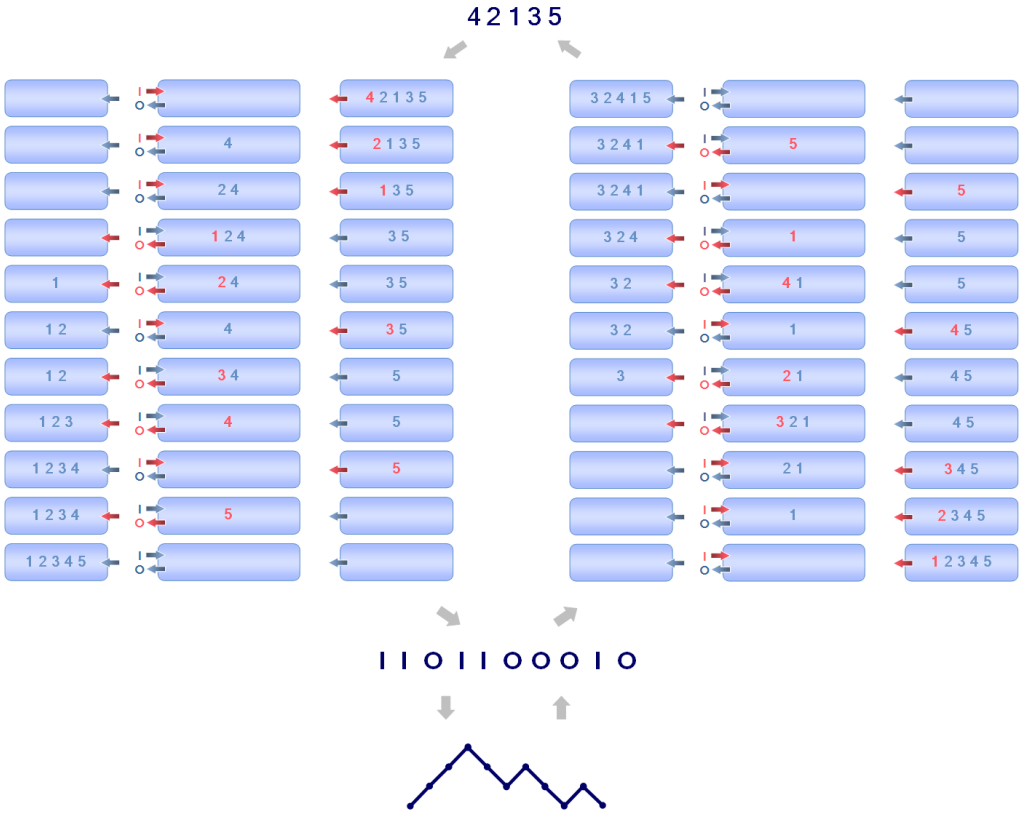


Figure 5.3. *The bijection for the stack.*

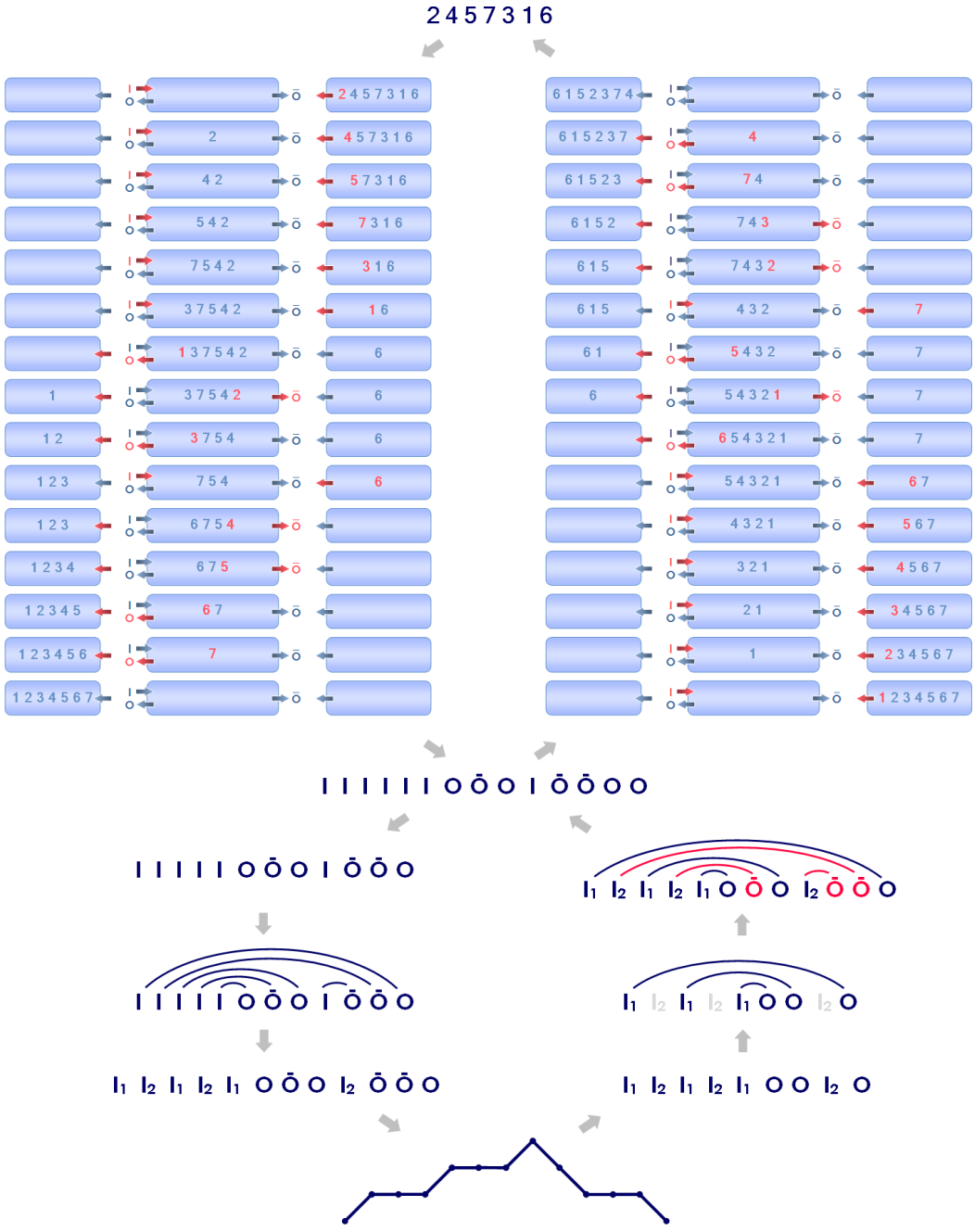


Figure 5.4. *The bijection for the input-restricted deque.*

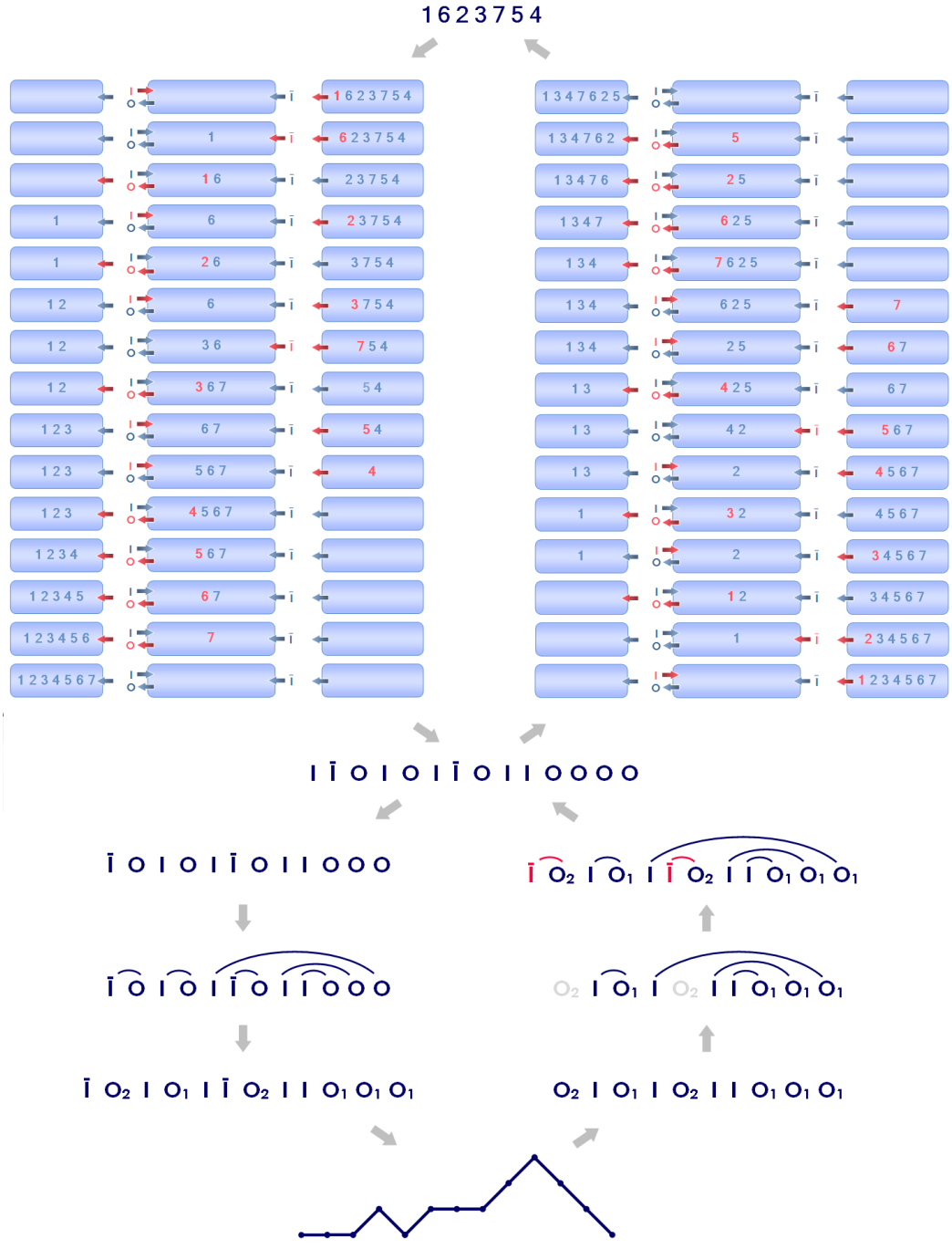


Figure 5.5. *The bijection for the output-restricted deque.*

5.7 A link between the bijections

In this last section, we describe the duality between the two bijections given for the restricted deques.

Definition 5.7.1. *Let $S \in \mathcal{X}$. We denote by $S^\star$ the sequence obtained by reversing S and swapping I for O and $\bar{I}$ for $\bar{O}$.*

Example 5.7.2. If $S = II\bar{O}I\bar{O}I\bar{O}IO\bar{O}IO$, then $S^\star = IO\bar{I}IO\bar{I}IO\bar{I}OIO$.

Observe that $S^\star$ corresponds to the sequence that operates backwards. In other terms, if we obtain τ by applying S on σ, then we can reobtain σ if we apply $S^\star$ on the reverse of τ, and then we reverse again.

Proposition 5.7.3. *Let $\sigma, \tau \in \Sigma$ and let $S \in \mathcal{X}$. Then*

$$S(\sigma) = \tau \qquad \Longleftrightarrow \qquad S^\star(\tau^r) = \sigma^r.$$

Suppose that a sequence S sorts a permutation σ. Then, from the previous proposition it is not difficult to show that $S^\star$ sorts the permutation $\left((\sigma^r)^{-1}\right)^r$.

Proposition 5.7.4. *Let $\sigma \in \Sigma$ and let $S \in \mathcal{X}$. Then*

$$S(\sigma) = id \qquad \Longleftrightarrow \qquad S^\star\Big(\left((\sigma^r)^{-1}\right)^r\Big) = id.$$

Proof. By proposition 5.7.3, $S(\sigma) = id$ if and only if $S^\star(id^r) = \sigma^r$, which yields $(\sigma^r)^{-1} \circ S^\star(id^r) = id$. Hence, proposition 2.3.1 implies that $S^\star\left((\sigma^r)^{-1} \circ id^r\right) = id$, and then, applying relation (1.1), we get $S^\star\Big(\left((\sigma^r)^{-1}\right)^r\Big) = id$. □

If we denote by

$$\mathcal{X}^\star = \{S^\star \mid S \in \mathcal{X}\}$$

the set of all sequences obtained from the set $\mathcal{X}$ through the operator $\star$, then obviously $\mathcal{S}^\star = \mathcal{S}$ and $\mathcal{D}^\star = \mathcal{D}$, while $\mathcal{D}^{ir\star} = \mathcal{D}^{or}$ and $\mathcal{D}^{or\star} = \mathcal{D}^{ir}$. In fact, when we operate backward, an input-restricted deque turns into an output-restricted deque, and conversely. Observe that the same holds for the associated sets of representatives: $\bar{\mathcal{D}}^{ir\star} = \bar{\mathcal{D}}^{or}$ and $\bar{\mathcal{D}}^{or\star} = \bar{\mathcal{D}}^{ir}$. This can be easily proven through the characterization given in section 5.4.

Hence, as observed by Knuth [13], every $\mathbb{D}^{ir}-$computable permutation is the reverse of the inverse of the reverse of a $\mathbb{D}^{or}-$computable one, and conversely. The same result can be given in terms of sortable permutations instead of computable ones: every $\mathbb{D}^{ir}-$sortable permutation is the reverse of the inverse of the reverse of a $\mathbb{D}^{or}-$sortable one.

The sorting sequences S and $S^\star$ associated to σ and $\left((\sigma^r)^{-1}\right)^r$ through $\varphi_{_{\mathbb{D}^{ir}}}$ and $\varphi_{_{\mathbb{D}^{or}}}$ are transformed into each other by the operator $\star$, as stated in proposition 5.7.4. Moreover, it is immediately checked that the Schröder path associated to S by $\psi_{_{\mathbb{D}^{ir}}}$ is the reversal of the path associated to $S^\star$ by $\psi_{_{\mathbb{D}^{or}}}$.

$$\begin{array}{ccccc}
Sort_n(\mathbb{D}^{ir}) & \underset{\varphi^{-1}_{\mathbb{D}^{ir}}}{\overset{\varphi_{\mathbb{D}^{ir}}}{\rightleftarrows}} & \bar{\mathcal{D}}_n^{ir} & \underset{\psi^{-1}_{\mathbb{D}^{ir}}}{\overset{\psi_{\mathbb{D}^{ir}}}{\rightleftarrows}} & \mathscr{S}_{2(n-1)} \\
\Big\updownarrow{\scriptstyle r\circ -1\circ r} & & \Big\updownarrow{\scriptstyle \star} & & \Big\updownarrow{\scriptstyle r} \\
Sort_n(\mathbb{D}^{or}) & \underset{\varphi^{-1}_{\mathbb{D}^{or}}}{\overset{\varphi_{\mathbb{D}^{or}}}{\rightleftarrows}} & \bar{\mathcal{D}}_n^{or} & \underset{\psi^{-1}_{\mathbb{D}^{or}}}{\overset{\psi_{\mathbb{D}^{or}}}{\rightleftarrows}} & \mathscr{S}_{2(n-1)}
\end{array}$$

Bibliography

[1] M. H. Albert, M. D. Atkinson, M. Bouvel, A. Claesson and M. Dukes, *On the inverse image of pattern classes under bubble sort*, Journal of Combinatorics, 2:231-243, 2011.

[2] M. Atkinson, *Permuting machines and permutation patterns*, Proceedings of Permutation Patterns 2007, in London Mathematical Society Lecture Note Series, 376:67-88, 2010.

[3] M. Barnabei, F. Bonetti and M. Silimbani, *Two permutation classes related to the Bubble Sort operator*, The Electronic Journal of Combinatorics, 19(3):P25, 2012.

[4] M. Bóna, *Exact enumeration of* 1342*-avoiding permutaions: a close link with labeled trees and planar maps*, Journal of Combinatorial Theory Series A, 80(2):257-272, 1997.

[5] M. Bóna, *A survey of stack-sorting disciplines*, The Electronic Journal of Combinatorics, 9(2):A1, 2003.

[6] M. Bóna, *Combinatorics of permutations*, Chapman & Hall/CRC, 2004.

[7] M. Bóna, *Sharper estimates for the number of permutations avoiding a layered or decomposable pattern*, Proceedings of Formal Power Series and Algebraic Combinatorics, 2004.

[8] J. Cibulka, *On constants in the Furedi-Hajnal and the Stanley-Wilf conjecture*, Journal of Combinatorial Theory Series A, 116(2):290-302, 2009.

[9] A. Claesson and S. Kitaev, *Classification of bijections between* 321− *and* 132−*avoiding permutations*, Séminaire Lotharingien de Combinatoire, 60:B60d, 2008.

[10] D. Denton, *Methods of computing deque sortable permutations given complete and incomplete information*, arXiv:1208.1532v1 [math.CO], 2012.

[11] I. M. Gessel, *Symmetric functions and P-recursiveness*, Journal of Combinatorial Theory Series A, 53(2):257-285, 1990.

[12] S. Kitaev, *Patterns in permutations and words*, Springer, 2011.

[13] D. E. Knuth, *The Art of Computer Programming – Volume 1: Fundamental Algorithms*, Addison-Wesley, 1968.

[14] D. E. Knuth, *The Art of Computer Programming - Volume 3: Sorting and Searching*, Addison-Wesley, 1973.

[15] P. A. MacMahon, *Combinatory Analysis*, Cambridge University Press, 1915-1916.

[16] A. Marcus and G. Tardos, *Excluded permutation matrices and the Stanley-Wilf conjecture*, Journal of Combinatorial Theory Series A, 107(1):153-160, 2004.

[17] V. R. Pratt, *Computing permutations with double-ended queues, parallel stacks and parallel queues*, Fifth Annual ACM Symposium on Theory of Computing, 268-277, 1973.

[18] D. G. Rogers and L. W. Shapiro, *Deques, trees and lattice paths*, Proceedings of the Eighth Australian Conference on Combinatorial Mathematics, in Combinatorial Mathematics VIII, 293-303, Springer-Verlag, 1981.

[19] P. Rosenstiehl and R. Tarjan, *Gauss codes, planar hamiltonian graphs, and stack-sortable permutations*, Journal of Algorithms, 5:375-390, 1984.

[20] R. Simion and F.W. Schmidt, *Restricted permutations*, European Journal of Combinatorics, 6:383-406, 1985.

[21] R. Tarjan, *Sorting using networks of queues and stacks*, Journal of the ACM, 19:341-346, 1972.

[22] H. Úlfarsson, *Describing West-3-stack-sortable permutations with permutation patterns*, arXiv:1110.1219v1.

[23] J. West, *Sorting twice through a stack*, Theoretical Computer Science, 117:303-313, 1993.

[24] J. West, *Generating trees and the Catalan and Schröder numbers*, Discrete Mathematics, 146(1-3):247-262, 1995.