Stack Sorting with Increasing and Decreasing Stacks

Giulio Cerbai* Lapo Cioni Luca Ferrari*

Dipartimento di Matematica e Informatica "U. Dini"
University of Firenze
Firenze, Italy

{giulio.cerbai,lapo.cioni,luca.ferrari}@unifi.it

Submitted: Nov 20, 2019; Accepted: Dec 17, 2019; Published: Jan 10, 2020

Abstract

We introduce a sorting machine consisting of $k+1$ stacks in series: the first k stacks can only contain elements in decreasing order from top to bottom, while the last one has the opposite restriction. This device generalizes the $\mathfrak{DI}$ machine introduced by Rebecca Smith, which studies the case $k = 1$. Here we show that, for $k = 2$, the set of sortable permutations is a class with infinite basis, by explicitly finding an antichain of minimal nonsortable permutations. This construction can easily be adapted to each $k \geqslant 3$. Next we describe an optimal sorting algorithm, again for the case $k = 2$. We then analyze two types of left-greedy sorting procedures, obtaining complete results in one case and only some partial results in the other one. We close the paper by discussing a few open questions.

Mathematics Subject Classifications: 05A05, 68R05, 68P10

1 Introduction

The problem of sorting a permutation using a stack was first introduced by Knuth [12] in the 1960s; in its classical formulation, the aim is to sort a permutation using a first-in/last-out device. As it is well known, in this case a permutation $\pi = \pi_1 \cdots \pi_n$ is sortable if and only if there do not exist three indices $i < j < k$ such that $\pi_k < \pi_i < \pi_j$. In the language of permutation patterns, we say that the set of sortable permutations is a *class* with basis $\{231\}$, meaning that each of these permutations cannot contain the pattern 231 as a subpermutation; a class is a downset in the permutation pattern poset and each

*G. C. and L. F. are members of the INdAM Research group GNCS; they are partially supported by INdAM - GNCS 2019 project "Studio di proprietá combinatoriche di linguaggi formali ispirate dalla biologia e da strutture bidimensionali" and by a grant of the "Fondazione della Cassa di Risparmio di Firenze" for the project "Rilevamento di pattern: applicazioni a memorizzazione basata sul DNA, evoluzione del genoma, scelta sociale".

class is determined by the minimal elements in its complement, which form its *basis*. Recall that the set of permutations can be partially ordered by means of the relation of "being a pattern", and we write $\sigma \leqslant \pi$ to mean that σ is a pattern of π. The resulting poset is called the *permutation pattern poset*, and a downset (i.e., a subset closed by going downwards) of the permutation pattern poset is usually called a class. For the basics on permutation patterns in combinatorics and computer science, we refer to [4].

More generally (see [19]), one can consider a network of sorting devices, each of which is represented as a node in a directed graph; when there is an arc from node S to node T, the machine is allowed to pop an element from S and push it into T; if we mark two distinct vertices as the input and the output, then the sorting problem consists of looking for a sequence of operations that allows us to move a permutation from the input to the output, finally obtaining the identity permutation.

In this framework, some of the typical problems are the following:

- characterize the permutations that can be sorted by a given network;
- enumerate sortable permutations with respect to their length;
- if the network is too complex, find a specific algorithm that sorts "many" input permutations and characterize such permutations.

Concerning the last problem, note that, for a given network of devices, although the set of sortable permutations forms a class in general, this is not true anymore if one chooses a specific sorting strategy; this approach leads in general to more complicated characterizations which involve other kinds of patterns (as it happens, for instance, for West 2-stack-sortable permutations [20]).

Although it is very hard to obtain interesting results for large networks, a lot of work has been done for some particular, small networks (see [3] for a dated survey, or [11] for a more recent one). In particular, the cases of two stacks in series or in parallel have been intensively studied over the last decades. Unfortunately, even for these apparently simple cases we know very little. Concerning two stacks in parallel, enumeration has been partly solved in [2], by giving a pair of functional equations that characterise the generating function of sortable permutations. Almost nothing is instead known for two stacks in series (basically, the only things we know are that sortable permutations constitute a class having infinite basis [14] and that the problem of deciding if a permutations is sortable has polynomial time complexity [15]). It is therefore natural to impose some constraints, in order to have more manageable problems whose possible solutions could shed some light on the original ones. In the present work we restrict our attention to the case of stacks connected in series, with the restriction that the elements are maintained inside each stack either in increasing or in decreasing order. Our starting point is [18], where Rebecca Smith proved that the permutations sorted by a decreasing stack followed by an increasing one form a class with basis $\{3241, 3142\}$. In the present paper, we try to find some information on what happens when we add more decreasing stacks in front. Our first result is that the device having two decreasing stacks followed by an

increasing one does not have a finite basis. Our proof can be easily adapted to show the same property for any number of decreasing stacks in front. Next, we provide an optimal algorithm to sort permutations, again in the case of two decreasing stacks followed by an increasing one. Our algorithm is optimal in the sense that it is able to sort all sortable permutations. Finally, we select a couple of (greedy) strategies and we prove that one of them can be studied in a very neat way, whereas the other one seems to be too difficult to allow a simple description of sortable permutations in terms of patterns, even including generalized versions of them.

2 Many decreasing stacks followed by an increasing one.

Generalizing the approach of [18], here we will consider a sorting device made by k decreasing stacks in series, denoted by $D_1, \ldots, D_k$, followed by an increasing stack I. Recall that "decreasing" (resp., "increasing") stack means that the elements inside the stack have to be in decreasing (resp., increasing) order from top to bottom. When $k = 0$, we just have a single increasing stack, so we obtain the usual Stacksort procedure. When $k = 1$, we obtain exactly the $\mathfrak{D}\mathfrak{I}$ machine described in [18]. In the sequel we denote our machine with $\mathfrak{D}^k\mathfrak{I}$.

The $\mathfrak{D}^k\mathfrak{I}$ machine can perform the following operations:

- d_0: push the next element of the input permutation into the first decreasing stack D_1;
- d_i, for $i = 1, \ldots, k-1$: pop an element from D_i and push it into the next decreasing stack D_{i+1};
- d_k: pop an element from the last decreasing stack D_k and push it into the increasing stack I;
- d_{k+1}: pop an element from the increasing stack I and output it (by placing it on the right of the list of elements that have already been output).

Notice that each operation can be performed only if it does not violate the restrictions of the stacks; in this case, we call it a *legal* operation. For the special case of the operation d_{k+1}, we will assume that d_{k+1} is legal both if we are pushing into the output the smallest among the elements not already in the output and if all the other operations are not legal.

Remark 1. If a pattern 231 is pushed into the last stack I, then the input permutation cannot be sorted. Moreover, this is the only situation that corresponds to a failure in the sorting procedure. This is a consequence of the classical result of Knuth [12], where in fact the only stack is used exactly as if it were increasing.

For any given k, we are now interested in characterizing the set

$$Sort_k = \{\pi \in S \mid \text{ there is a sequence of legal operations of } \mathfrak{D}^k\mathfrak{I} \text{ that sorts } \pi\}.$$

If $\pi \in Sort_k$, we say that π is k-sortable. Notice that we are using the sorting machine in the most general setting, so using a standard argument it is easy to show that $Sort_k$ is a class for every k. The natural way to describe $Sort_k$ is therefore to understand its basis. Here we show that, even when $k = 2$, the basis of $Sort_k$ is infinite, by explicitly finding an infinite antichain of permutations which are not 2-sortable and are minimal with respect to the pattern ordering. The construction of the infinite antichain described in the next theorem can be easily adapted to every $k \geqslant 2$. The software *PermLab* [1], developed by Michael Albert, has been an extremely useful tool to find such an antichain. This result is in sharp contrast with what happens when $k = 1$, which is the case considered in [18], where it is shown that the basis is finite (of cardinality 2). We start by stating some useful lemmas, whose proofs are straightforward.

Lemma 2. *Let π be an input permutation for the $\mathfrak{D}^k\mathfrak{I}$ machine; if $i < j$ and $\pi_i > \pi_j$, then π_i is necessarily pushed into I before π_j. In other words, the decreasing stacks $D_1, \ldots, D_k$ cannot repair inversions.*

Lemma 3. *Let π be an input permutation for the $\mathfrak{D}^k\mathfrak{I}$ machine and let $a < b < c$ be elements of π. Focus on the instant when, during the sorting process, b is pushed into the increasing stack. Then, if any of the following conditions holds, π cannot be sorted anymore:*

1. *c is in D_j and a is in D_k, with $k \leqslant j$;*
2. *c is in D_j, for some j, and a is still in the input;*
3. *c and a are still in the input, with a following c.*

Proof. The previous lemma implies that, if any of the above conditions is satisfied, a pattern 231 is pushed into the increasing stack, so π cannot be sorted anymore due to Remark 1. □

Rephrasing the last lemma, if we try to sort π and, when b is pushed into the increasing stack, one of the listed conditions holds, then there is no hope to complete the procedure to obtain a sorted output.

Theorem 4. *For $j \geqslant 0$, define the permutation:*

$$\alpha^{(j)} = 2j + 4, 3, \omega^{(j)}, 1, 5, 2,$$

where $\omega^{(j)} = 2j{+}2, 2j{+}5, 2j, 2j{+}3, 2j{-}2, 2j{+}1, \ldots, 6, 9, 4, 7$. Then the set of permutations $\{\alpha^{(j)}\}_{j\geqslant 0}$ constitutes an infinite antichain in the permutation pattern poset, each of whose element is not 2-sortable. Moreover, $\alpha^{(j)}$ is minimal with respect to such a property, i.e. if we remove any element of $\alpha^{(j)}$ we obtain a 2-sortable permutation.

Proof. We start by proving (using induction) that $\alpha^{(j)}$ is not 2-sortable, for every j. If $j = 0$, it is easy to check that $\alpha^{(0)} = 43152$ cannot be sorted using the $\mathfrak{D}^2\mathfrak{I}$ machine. Let $j \geqslant 1$ and $\alpha^{(j)} = \alpha_1 \cdots \alpha_{2j+5}$. Since $\alpha_1 = 2j + 4 > \alpha_2 = 3$, α_1 has to be pushed into

D_2 before α_2 enters D_1. Notice that the maximum of $\alpha^{(j)}$ is $\alpha_4 = 2j + 5$ and there are elements following it in $\alpha^{(j)}$ which are smaller than both α_1 and α_4, so we cannot push α_1 into I due to the previous lemma. Thus the only option we are left with is to push $\alpha_3 = 2j + 2$ into D_1 immediately above α_2. Now, the next element of the input is the maximum α_4, and of course we can push it through the decreasing stacks and finally into I. Observe that pushing the maximum available element in I is always convenient. So the second maximum $\alpha_1 = 2j + 4$, which is currently contained in D_2, can be pushed into I similarly, leaving us with just the elements α_3 and α_2 in D_1, with α_3 on top. The next element of the input is $\alpha_5 = 2j < \alpha_3$, so pushing α_3 into D_2 is forced. Now, getting rid of the two maximal elements of $\alpha^{(j)}$ already pushed into I, notice that we are in the same configuration that arises when processing $\alpha^{(j-1)}$ after considering the first two elements, so we can conclude that $\alpha^{(j)}$ is not 2-sortable by inductive hypothesis. An example of the above argument for $j = 2$ is shown in Figure 1. In passing, we observe that the optimal sorting strategy here would be, at each step, to push the maximum and second maximum element still available into I; in the general case, this strategy fails since 3 remains stuck in D_1, blocked by a larger element in D_2, until we reach the final portion of $\alpha^{(j)}$. This crucial remark will be useful in the last part of this proof.

We now prove that $\alpha^{(j)}$ is minimal not 2-sortable. This can be proved with a case by case analysis, depending on the element we choose to remove. We show in detail just some of these cases, leaving the remaining ones to the reader.

- If we remove the first element $\alpha_1 = 2j+4$, we can push the new first element $\alpha_2 = 3$ directly into D_2; from now on, we can follow the sorting procedure outlined above, pushing at each step the maximum and second maximum available elements into I. However in this case, before processing the three last elements $1, 5, 2$, we have that both 3 and 4 are in D_2, whereas in processing $\alpha^{(j)}$ we have 3 inside D_1 and 4 inside D_2. Therefore we can now push 1 into D_1 and 5 into I and finally $4, 3, 2, 1$ in the correct order, as desired.

- If we remove $\alpha_2 = 3$, we can sort the resulting permutation using the same procedure, this time obtaining a configuration with just 4 in D_2 and $1, 5, 2$ in the input.

- Consider the removal of an element $x = \alpha_i$, for some $i = 3, \ldots, 2j + 2$. In the first part of the sorting procedure, the element 3 is stuck into D_1, similarly to what happens when processing $\alpha^{(j)}$. However, as soon as we scan the element that follows x in $\alpha^{(j)}$, when we push maximum and second maximum in I we are left for a moment with the stack D_2 empty (and just 3 in D_1), because we removed the element x that had to occupy D_2. So we can take advantage of this fact and move 3 into D_2, concluding the sorting procedure as in the previous cases.

- The removal of the elements $1, 5, 2$ can be dealt with in a similar way.

Thus we have seen that, in any case, removing any element of $\alpha^{(j)}$ results in a 2-sortable permutation, so $\alpha^{(j)}$ is minimal not 2-sortable. □

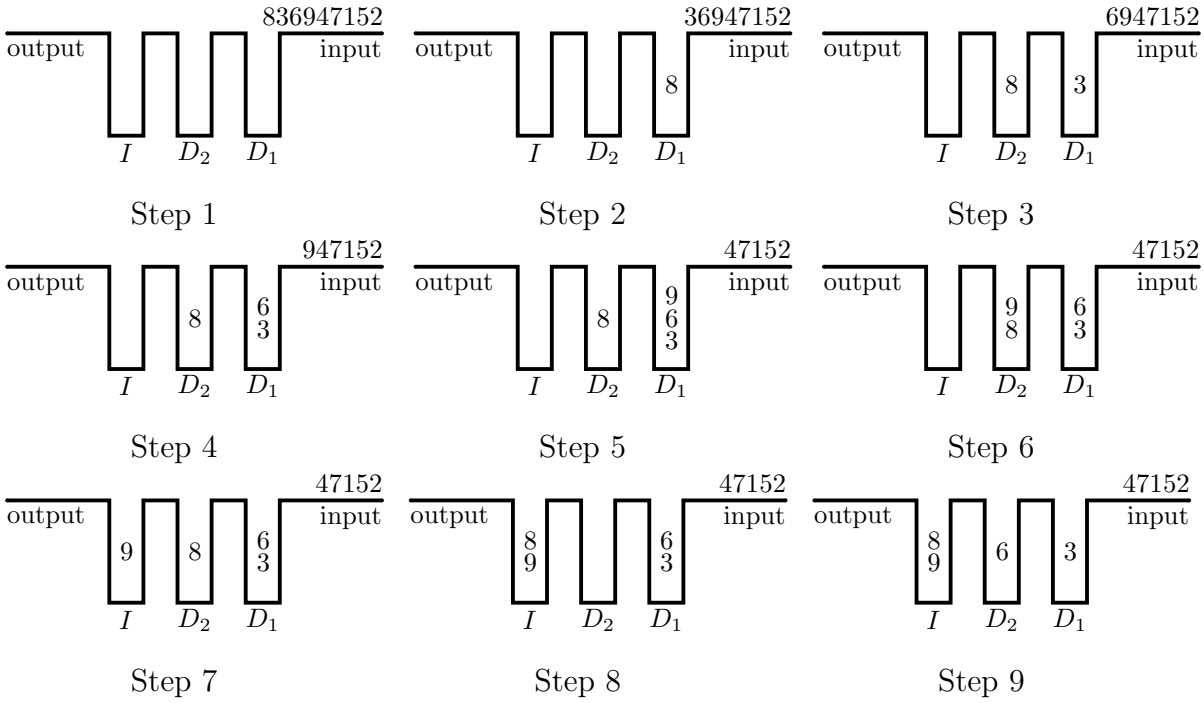

Figure 1: The recursive construction described in Theorem 4 with input $\alpha^{(2)} = 836947152$ (on the right). The last step corresponds to input $\alpha^{(1)} = 6347152$ after having pushed the first two elements into the machine.

Corollary 5. *The basis of $Sort_2$ is infinite, since it contains the infinite antichain $\{\alpha^{(j)}\}_{j\geqslant 0}$ defined in the previous theorem.*

Remark 6. Theorem 4 remains true if we permute the elements 1,2,3 of $\alpha^{(j)}$, for every j.

3 An optimal algorithm for the $\mathfrak{D}^2\mathfrak{I}$-machine

The results of the previous section suggest that it may be very hard to enumerate k-sortable permutations when $k \geqslant 2$. In the present section, we show that, when $k = 2$, we are at least able to design an optimal algorithm, called $\mathcal{D}^2\mathcal{I}$, which sorts all 2-sortable permutations.

Algorithm $\mathcal{D}^2\mathcal{I}$ can be explicitly described as follows:

1. If $Top(I)$ is the next element to be output, then perform d_3.
2. If all the elements contained in D_1 and D_2 are the next elements to be output, then move them to the output.
3. If each of the previous instructions cannot be executed, perform d_1, provided that condition (β) holds.

4. If each of the previous instructions cannot be executed, perform d_0, provided that condition (γ) holds.

5. If each of the previous instructions cannot be executed, perform d_2, provided that condition (α) holds.

6. Otherwise, perform d_3.

Conditions (α), (β) and (γ) are the following (we remark that, when a stack is empty, any statement about it is considered to be true):

(α) $Top(D_2) < Top(I)$.

(β) $Top(D_2) < Top(D_1)$ and $Top(D_1) < Top(I)$.

(γ) $Top(D_1) < Input$, $Input < Top(I)$ and the sequence of elements from $Input$ to the first element larger than $Top(D_2)$ is increasing.

In the sequel, each of the d_i's, for $i = 0, 1, 2, 3$, will be called an *operation*, exactly as we did until now. Instead, each of the six items in the above description of algorithm $\mathcal{D}^2\mathcal{I}$ will be called an *instruction*. Therefore, an instruction of $\mathcal{D}^2\mathcal{I}$ consists of performing a (legal) operation, provided that some constraints are satisfied.

It is not difficult to realize that instruction 2 of the above algorithm is not essential for its correctness, so in principle we could remove it. However, in some cases (and in particular in the proof of the optimality) it is convenient to have it.

Algorithm $\mathcal{D}^2\mathcal{I}$ sets certain priorities between operations, provided that certain conditions are fulfilled. In general, given any two operations $\tilde{d}$ and $\bar{d}$, we will use the notation $\tilde{d} \rhd \bar{d}$ to mean that $\tilde{d}$ has higher priority than $\bar{d}$ (and so, if both $\tilde{d}$ and $\bar{d}$ are legal, $\tilde{d}$ is performed). Moreover, we denote with ${}^{(\omega)}d$ any operation d which, in order to be performed, has to be legal and also to satisfy an additional constraint ω.

Using these notations, we can illustrate algorithm $\mathcal{D}^2\mathcal{I}$ (in which instruction 2 has been removed) with the following chain of priorities:

$$d_3 \rhd {}^{(\beta)}d_1 \rhd {}^{(\gamma)}d_0 \rhd {}^{(\alpha)}d_2.$$

Notice that condition (α) is equivalent to saying that operation d_2 is legal; however, for homogeneity's sake, we have preferred to state it explicitly in the description of our algorithm.

Remarks.

1. If, at some point, algorithm $\mathcal{D}^2\mathcal{I}$ performs instruction 6, then the input permutation is not sorted at the end of the process, and this is the only obstruction to the sorting process. In other words, $\mathcal{D}^2\mathcal{I}$ sorts a permutations if and only if it never executes instruction 6.

2. To some extent, algorithm $\mathcal{D}^2\mathcal{I}$ generalizes Smith's algorithm for a decreasing stack and an increasing stack in series. More specifically, interpreting the first stack of our device as the input container (and so removing the decreasing constraint) and operation d_1 as the input operation, which insert the current element of the input permutation into the (new) first decreasing stack, we obtain precisely Smith's algorithm.

The proof of the optimality of our algorithm is not trivial, and requires several steps. Our first goal is to prove some properties of algorithm $\mathcal{D}^2\mathcal{I}$.

Lemma 7. *At every step, we have $Top(D_2) < Top(I)$.*

Proof. The proof is by induction on the step number. At the beginning of the sorting process, the statement in the lemma is true since all the stacks are empty. Now suppose that the statement holds at step n, and consider all possible instructions that can be performed: a simple case-by-case analysis shows that the same inequality is true also at step $n+1$. □

Corollary 8. *The last instruction of $\mathcal{D}^2\mathcal{I}$ can be executed only if D_2 is empty.*

Proof. The previous lemma tells that condition (α) is always true, so instruction 5 of $\mathcal{D}^2\mathcal{I}$ can always be executed provided that D_2 is not empty. □

Lemma 9. *At every step, we have $Top(D_1) < Top(I)$.*

Proof. The proof works by induction, exactly in the same way as Lemma 7. However, it is worth giving the details in at least one case. Suppose that, at step n of the algorithm, we have $Top(D_1) < Top(I)$ and we perform instruction 5, that is we move $Top(D_2)$ into I. Notice that, at step n, we must have $Top(D_1) < Top(D_2)$, otherwise condition (β) would hold, and so instruction 3 would be performed by $\mathcal{D}^2\mathcal{I}$ instead of instruction 5. Therefore, at step $n+1$, we have $Top(D_1) < Top(I)$, because $Top(I)$ at step $n+1$ is exactly $Top(D_2)$ at step n. □

Corollary 10. *The last instruction of $\mathcal{D}^2\mathcal{I}$ can be executed only if D_1 is empty.*

Proof. We know from Corollary 8 that D_2 must be empty in order to execute instruction 6. If D_1 were not empty, then condition (β) would be satisfied, thanks to the previous lemma, and so instruction 3 would be performed. □

From now on, we aim at showing that, if π is a 2-sortable permutation, then there exists a sorting algorithm for π which has many properties that $\mathcal{D}^2\mathcal{I}$ also has. In the end, we will prove that such properties do characterize algorithm $\mathcal{D}^2\mathcal{I}$.

Proposition 11. *Let π be a 2-sortable permutation. There exists a sorting algorithm for π which performs operation d_0 (resp., d_1, d_2) only if condition (γ) (resp., (β), (α)) holds.*

Proof. Condition (α) is obviously necessary in order to perform d_2, since I is an increasing stack.

Consider now condition (β). Again, in order to perform d_1 we must have $Top(D_2) < Top(D_1)$, since D_2 is a decreasing stack. Moreover, we will show that it is necessary to have $Top(D_1) < Top(I)$ if we want to perform d_1 and eventually sort the input. Indeed, suppose that $Top(I) < Top(D_1)$ and set $Top(I) = b$, $Top(D_2) = a$ and $Top(D_1) = c$. There are two cases to analyze. If $a < b$, then performing d_1 would force b to reach the output before a, which would cause the sorting process to fail. On the other hand, if $b < a$, we must have that b is the next element to be output. Therefore we can perform d_3 until $Top(I)$ is not the next element to be output. But in this case necessarily $a < Top(I)$, and we are thus led to the previous case.

Finally, we analyze condition (γ). The inequality $Top(D_1) < Input$ is necessary in order to perform d_0, since D_1 is decreasing; the inequality $Input < Top(I)$ is necessary as well, by an argument similar to that employed for condition (β). We will now show that requiring the third constraint of (γ) to perform d_0 does not prevent the procedure to sort the input. Suppose that the third constraint of (γ) is not satisfied and set $x = Top(D_2)$. This means that currently the input consists of a (nonempty) increasing sequence of elements smaller than x whose last term (call it b) is bigger than the next one (call it a). Of course, $a \leqslant x$ as well. First of all, if it were possible to perform d_1, then necessarily $Top(D_2) < Top(D_1)$. Since we are supposing to be able to perform d_0, we already know that $Top(D_1) < Input$, thus we would have $Top(D_2) < Input$. Therefore the third constraint of (γ) would be satisfied, which it is not. If we decide to perform d_0, we still cannot perform d_1 of course, so we can continue to perform d_0 until we reach a. At that point, the only possible operation to perform would be d_2. However, the same configuration could have been reached by performing d_2 before starting executing d_0. This essentially means that the set of configurations that are reachable by performing d_2 whenever the third constraint of (γ) is not satisfied is a superset of the set of configurations that are reachable by performing d_0 in the same situation. Thus, if the input is 2-sortable, then it is 2-sortable also by an algorithm which executes d_0 only if (γ) is satisfied. □

At this point, it is convenient to make a brief recap. What we have shown until now is that, if π is a 2-sortable permutation, then there exists a sorting algorithm for π having the following features:

- if $Top(I)$ is the next element to be output, it performs d_3;
- it executes instruction 2 of $\mathcal{D}^2\mathcal{I}$ whenever it is possible to execute it;
- it performs operation d_0 (resp., d_1, d_2) only if condition (γ) (resp., (β), (α)) hold;
- if no other operation is allowed, it performs d_3.

In order to conclude our proof, we now need to show that, if π is 2-sortable, then there exists a sorting algorithm $\mathcal{ALG}$ for π which satisfies the above listed properties and, in addition, performs operations d_0, d_1, d_2 in exactly the same order as algorithm $\mathcal{D}^2\mathcal{I}$ does. This would mean precisely that $\mathcal{ALG}$ coincides with $\mathcal{D}^2\mathcal{I}$, as desired.

We start by comparing operations d_1 and d_2. From now on, any sorting algorithm having the properties listed above will be called *special*, and we will denote a generic special algorithm with $\mathcal{ALG}$.

Proposition 12. *Let π be a 2-sortable permutation. There exists a special sorting algorithm $\mathcal{ALG}$ for π for which ${}^{(\beta)}d_1 \rhd {}^{(\alpha)}d_2$.*

Proof. Suppose that, at a certain point of the execution of $\mathcal{ALG}$ on π, it is possible to perform both d_1 and d_2. Clearly, we can suppose that both instruction 1 and 2 of $\mathcal{D}^2\mathcal{I}$ cannot be executed by $\mathcal{ALG}$. This implies that there must exist an element a of π still in the input, which is smaller than $Top(D_2)$. Set $x = Top(D_2)$ and $y = Top(D_1)$. If we perform d_2, then we would have $x = Top(I) < Top(D_1) = y$ (since we are supposing that it was possible to perform also d_1). This means that a could overcome y only when y is already inside I, and this can happen only if x has already been output. This however would cause the output to be unsorted, since in the output x would come before a, and $x > a$. We can thus conclude that, in the hypothesis of the proposition, performing d_2 would make the sorting process fail, and so ${}^{(\beta)}d_1 \rhd {}^{(\alpha)}d_2$, as desired. □

We can now observe that, if π is a 2-sortable permutation, then there exists a special sorting algorithm $\mathcal{ALG}$ for π such that Lemmas 7 and 9 and Corollaries 8 and 10 hold. In fact, all the proofs of the above mentioned results do not depend on the specific algorithm $\mathcal{D}^2\mathcal{I}$, except for Lemma 9, where it is explicitly used that ${}^{(\beta)}d_1 \rhd {}^{(\alpha)}d_2$. However, in view of the previous proposition, without loss of generality we can assume that there is a special sorting algorithm for π which satisfies such a condition. In what follows, a special sorting algorithm with this additional property will be called *extraspecial* (and still denoted $\mathcal{ALG}$).

Before concluding our *tour de force*, we still need a final preparatory result.

Proposition 13. *A permutation π is 2-sortable if and only if it does not contain any occurrence bca of the pattern 231 such that, at some step of any extraspecial sorting algorithm for π, we have $b = Top(I)$ and c and a are still in the input.*

Proof. Suppose that π is 2-sortable and that bca is an occurrence of 231 in π. Moreover, suppose that, at some point of the extraspecial sorting algorithm $\mathcal{ALG}$, we have $b = Top(I)$ and c and a are still in the input. Then, if we continue the execution of $\mathcal{ALG}$, since the first two stacks are decreasing, a can overcome c only inside the increasing stack; but c can enter the increasing stack only if b is in the output. This will cause b to be output before a, and so the input permutation would eventually not be sorted, which is a contradiction.

On the other hand, suppose that π is not 2-sortable and let $\mathcal{ALG}$ be any extraspecial algorithm. Since π is not 2-sortable, at some point $\mathcal{ALG}$ output an element y which is not the correct one; in other words, there exists $x < y$ which is still inside one of the decreasing stacks or in the input. However, the decreasing stacks must be empty, as a consequence of Corollaries 8 and 10, hence x must be in the input. Moreover, if the z is

the first element of the input when y goes to the output, then necessarily $z > y$, since otherwise condition (γ) would be satisfied (which is not possible, since $\mathcal{ALG}$ executes instruction 6). Thus, in particular, $z \neq x$, and the elements yzx constitute an occurrence of 231 in π which violates the required condition. □

We are finally ready to conclude our proof of the optimality of $\mathcal{D}^2\mathcal{I}$.

Theorem 14. *The sorting algorithm $\mathcal{D}^2\mathcal{I}$ is optimal, i.e. it sorts all 2-sortable permutations.*

Proof. Let π be a sortable permutation. Then there exists an extraspecial algorithm $\mathcal{ALG}$ which sorts π. The only possibility for $\mathcal{ALG}$ to be different from $\mathcal{D}^2\mathcal{I}$ is that the order in which $\mathcal{ALG}$ performs operations d_0, d_1 and d_2 may be different. However, we already know that, for an extraspecial algorithm, ${}^{(\beta)}d_1 \rhd {}^{(\alpha)}d_2$. What remains to do is to compare d_0 with d_2 and d_0 with d_1.

First, suppose that $\mathcal{ALG}$ is in a certain configuration, in which both d_0 and d_2 can be performed. We can further assume that condition (β) is not satisfied, otherwise d_2 would certainly not be performed, as a consequence of Proposition 12. Set $c = Top(I), b = Top(D_2)$ and $a = Top(D_1)$, and call y the first element of the current input which is greater than b (if it exists). Since we are supposing that condition (γ) is satisfied, the sequence from the beginning of the current input to y is increasing. If there were an element $x < b$ following y, then performing d_2 would prevent us from successfully sort the permutation, as a consequence of Proposition 13 (the three elements b, y and x would constitute the "bad" occurrence of 231). Therefore, also keeping in mind that $b > a$ (since $a < c$ as a consequence of condition (γ) and we are supposing that condition (β) is not satisfied), we can assert that the set of all numbers contained in D_1, D_2 and in the input before y (if such an element exists) is precisely the set of all numbers $\leqslant b$ which are not already in the output. It is now possible to show that, using algorithm $\mathcal{D}^2\mathcal{I}$, such numbers reach the output before any other number makes any move. Indeed, $\mathcal{D}^2\mathcal{I}$ performs d_0 and pushes the first number of the current input inside D_1 (above a). Then the algorithm keeps performing d_0 until y is reached (in fact condition (β) keeps failing to be satisfied, since all numbers before y in the input are $< b$); at this point, D_1 and D_2 contain precisely the next elements to be output, so $\mathcal{D}^2\mathcal{I}$ performs instruction 2. We can thus conclude that, in the considered configuration, using algorithm $\mathcal{D}^2\mathcal{I}$ does not prevent the permutation from being sorted, hence performing d_0 instead of d_2 is irrelevant (if not necessary).

Now suppose that $\mathcal{ALG}$ is in a certain configuration, in which both d_0 and d_1 can be performed. Letting $Top(I) = d$, $Top(D_2) = a$, $Top(D_1) = b$ and $Input = c$, we then know that $a < b < c < d$. If $\mathcal{ALG}$ chose to perform d_0, then c would be pushed into D_2, with b still in the same stack. Clearly, sooner or later, there would be a step of $\mathcal{ALG}$ moving b from D_2 to D_1. Let us now focus on this exact moment (when b is pushed into D_1) and call the resulting configuration $\aleph$: we claim that, if we modify $\mathcal{ALG}$ by just performing d_1 instead of d_0 in the configuration described at the beginning of the present paragraph, we can reach the same configuration $\aleph$ mentioned above. So suppose that, after having

performed d_0 and before moving b to D_2, the elements that $\mathcal{ALG}$ has pushed into D_1 are $c, c_1, \ldots, c_k$. Clearly, when b is moved into D_2, such elements must all be inside I, since they are all greater than b and D_2 is decreasing. If, in the meanwhile, a has not been pushed into I, then we can reach the same configuration by first moving b into D_2 (thus performing d_1) and then moving all the elements $c, c_1, \ldots, c_k$ into I by performing the same sequence of operations. Otherwise, if a would have been moved into I before all elements $c, c_1, \ldots, c_k$ reach I (possibly together with some further elements from D_2), this should have been done in a configuration in which both d_0 and d_1 were not legal (since we have already shown that both $d_0 \rhd d_2$ and $d_1 \rhd d_2$). This is however impossible, since we will now see that d_1 is certainly legal. Indeed, consider the instant immediately before a is pushed into I. Since b is in D_1, $Top(D_1) > b$ (because D_2 is decreasing) and we know that $b > a$. As a consequence, $Top(D_2) < Top(D_1)$. Moreover, since $\mathcal{ALG}$ is extraspecial, we also know from Lemma 9 that $Top(D_1) < Top(I)$. Therefore condition β is satisfied, hence d_1 is legal. Summing up, we have shown that, if both d_0 and d_1 are legal, then performing d_0 leads to a configuration which can be reached also performing d_1 instead. As a consequence, performing d_1 instead of d_0 preserves sortability. □

The sequence counting permutations of length n that are sortable using the $\mathcal{D}^2\mathcal{I}$ machine starts 1,1,2,6,24,117,651,3961,25661,174062,1222784, and appears to be new to [16].

4 Some further algorithms

As we have seen in the previous section, there exists an optimal algorithm for the $\mathfrak{D}^2\mathfrak{I}$ machine which is able to sort all sortable permutations. However, it is not a very easy one: in order to understand which operation should be performed at each step, one needs to check certain conditions, which in some cases are rather weird. Another approach could be to consider some much easier algorithms, which of course fail to be optimal, but have the nice feature of being more intuitive.

In the present section we briefly sketch two very natural algorithms, one of which turns out to be "too easy" whereas the other one reveals to be "too hard".

4.1 A left-greedy algorithm

Our first proposal is a left-greedy procedure for the $\mathfrak{D}^k\mathfrak{I}$ machine: at each step, we perform the operation d_j having maximum index j among the legal available operations. In other words, such a left-greedy procedure is characterized by the following chain of priorities:

$$d_{k+1} \rhd d_k \rhd d_{k-1} \rhd \cdots d_1 \rhd d_0.$$

Setting $Sort_k^{(lg)} = \{\pi : \pi \text{ is sorted by the left-greedy procedure}\}$, it turns out that $Sort_k^{(lg)}$ is in fact a class which we are able to characterize completely. The choice of a left-greedy strategy, instead of a right-greedy one, is suggested by the results contained in [17].

Proposition 15. *For every $k \geqslant 0$, $Sort_k^{(lg)}$ is a class with basis $\{231\}$.*

Proof. We start by proving that if π contains 231, then $\pi \notin Sort_k^{(lg)}$. Let bca be an occurrence of the pattern 231 in π. If b is pushed into I before c, then π cannot be sorted, as a consequence of Lemma 3. Then suppose that b is stuck into a decreasing stack D_j, for some j. In particular, since the algorithm is left-greedy, this implies that D_k is not empty (more precisely, each stack D_i, with $i \geqslant j$, has to contain at least one element). Let z be the first element that reaches D_k without going directly into I and consider the step in which z is pushed into D_k; again because we are using a left greedy strategy, the next stack I cannot be empty at that moment. Let $y = Top(I)$. Note that $y < z$, otherwise z would be pushed into I. Moreover, since y is not pushed into the output, there must still be an element $t < y$ that is not in the output (and neither in I, of course). In particular, t follows z, because z is the top of D_k. We are thus in a position to apply Lemma 3 with the three elements $t < y < z$, which is enough to conclude that $\pi \notin Sort_k^{(lg)}$.

Conversely, we have to show that, if $\pi \notin Sort_k^{(lg)}$, then π contains the pattern 231. Factorize π as $\pi = \alpha_1\alpha_2\cdots\alpha_r$, where each α_i is a maximal decreasing sequence. Without loss of generality, we can suppose that, if α_1 contains i elements, then $\alpha_1 \neq i(i-1)\cdots 21$; otherwise, in fact, we could simply remove α_1 and consider the remaining permutation: since by hypothesis π is not sortable, there must be an index h such that α_h is not the set of the next elements to be output. So suppose that $\alpha_1 = \pi_1\pi_2\cdots\pi_i \neq i(i-1)\cdots 21$, hence $\pi_i < \pi_{i+1}$. All the elements of α_1 are pushed into the increasing stack, whereas π_{i+1} remains stuck into D_k. Notice that the hypothesis on α_1 implies that not all elements inside the increasing stack can be output, since there is at least one element x following π_{i+1} in π which is smaller than all elements of α_1. Such an element x is still in the input when π_{i+1} reaches D_k (since all the remaining decreasing stacks are clearly empty). Call y the top of the increasing stack when π_{i+1} reaches D_k: then the three elements y, π_{i+1} and x are an occurrence of the pattern 231 in π. □

As a consequence of the previous proposition, our left greedy procedures sort precisely the same permutations as Stacksort does. Thus, in a sense, adding any number of decreasing stacks before an increasing one does not improve the sorting power of the machine, provided that we always perform the leftmost legal operation. This does not mean, however, that the left-greedy algorithms are equivalent to Stacksort. Indeed, taking for instance $k = 1$ and the input permutation 2341, the left-greedy $\mathfrak{D}^k\mathfrak{I}$ machine returns 2134 as output (see Figure 2), whereas Stacksort returns 2314. In other words, while the preimage of the identity permutation is the same for Stacksort and for every left-greedy $\mathfrak{D}^k\mathfrak{I}$ machine, the preimages of other permutations are in general different. It would be certainly interesting to investigate more deeply the preimage of a generic permutation for the left-greedy $\mathfrak{D}^k\mathfrak{I}$ machine. We remark that in [8, 9] the author gives methods for studying preimages under the Stacksort map.

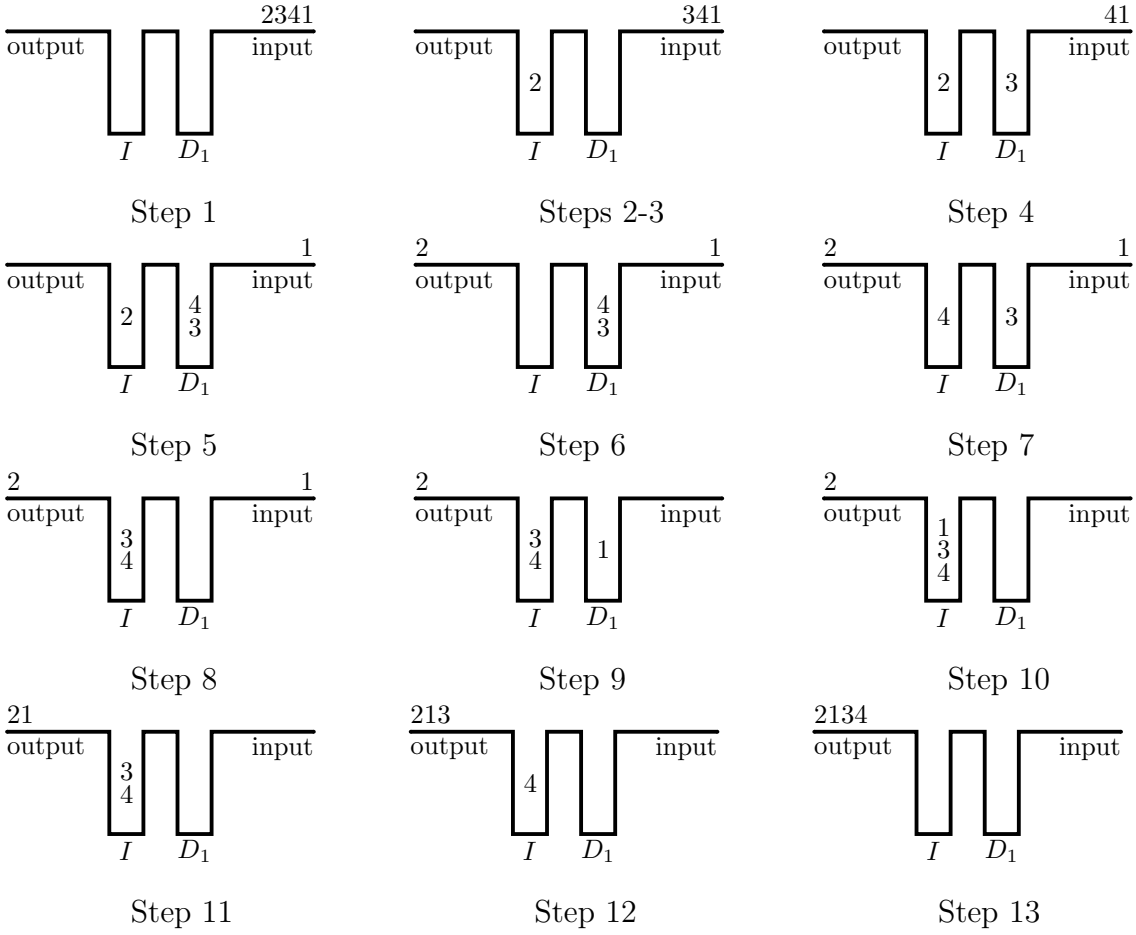

Figure 2: The steps that transform 2341 into 2134 via the left-greedy $\mathfrak{D}^1\mathfrak{I}$ machine.

4.2 A quasi left-greedy algorithm.

There is a better way to design an algorithm which is quasi left-greedy and is able to sort more permutations than the previous one. The idea is to give the increasing stack a privileged role, using it only when no other operation is possible. Formally, at each step we choose to perform the first legal operation according to the following priority rule:

$$d_{k+1} \rhd d_{k-1} \rhd d_{k-2} \rhd \cdots \rhd d_1 \rhd d_0 \rhd d_k.$$

This quasi left-greedy procedure is similar to the optimal algorithm for the $\mathfrak{D}^2\mathfrak{I}$ machine described in Section 3, the only difference being that no additional conditions are required in order to perform operations (other than the fact that each operation can be performed only if it is legal, of course).

In analogy with the previous case, define $Sort_k^{(qlg)}$ to be the set of permutations sorted by the quasi left-greedy algorithm with k decreasing stacks; such permutations will be called *qlg-k-sortable permutations*. We observe immediately that the permutation 231 is qlg-2-sortable. Unfortunately, $Sort_k^{(qlg)}$ is not in general a permutation class, except for the case $k = 1$, for which we have the following result, whose proof can be found in [7].

Lemma 16. *$Sort_1^{(qlg)}$ is a class with basis $\{213\}$.*

When $k > 1$ things become much more involved. As an example, for $k = 2$, the permutation 631425 is qlg-2-sortable, whereas its subpermutation 52314 is not. In fact, a complete characterization of $Sort_2^{(qlg)}$ appears to be quite hard. In the rest of the section we will present some partial results that we have obtained, that should make abundantly clear that understanding the set of qlg-k-sortable permutations is a very hard task. The proofs of all the results can be found in the arxiv version of the present paper [6].

Proposition 17. *Let π be a qlg-2-sortable permutation. Then*

- *π avoids 3214;*
- *if π contains the pattern 52314, then each occurrence of 52314 can be extended to one of the following patterns, where the additional elements are marked with a dot:*
  - $63\dot{1}425$;
  - $7\dot{2}\dot{1}4536$, $7\dot{3}\dot{1}4526$;
  - $\dot{7}\dot{2}8\dot{1}4536$, $\dot{7}\dot{3}8\dot{1}4526$;
  - $\dot{8}\dot{2}7\dot{1}4536$, $\dot{8}\dot{3}7\dot{1}4526$.

The above proposition cannot be inverted, since there exist permutations that are not qlg-2-sortable, yet satisfy the two conditions listed above. An example is given by 11 2 10 1 4 9 3 6 7 5 8; notice, in particular, that it contains three occurrences of 52314 and each of them can be extended to one of the above dotted patterns (more specifically, two of the occurrences can be extended to $\dot{8}\dot{2}7\dot{1}4536$, whereas the remaining one can be extended to $7\dot{2}\dot{1}4536$).

In fact, starting from the permutation 52314, it is possible to construct a sequence of permutations of increasing lengths whose sortability depends on the parity of the length. To be more precise, for $m \geqslant 1$, define the permutation $\gamma_m \in S_{3m+2}$ as follows:

$$\gamma_m = 3m+2,\ \underbrace{2,\ 3m+1,\ 1,}_{P_1}\ \underbrace{4,\ 3m,\ 3,}_{P_2}\ldots,\ \underbrace{2m,\ 2m+1,\ 2m-1,}_{P_m}2m+2.$$

In other words, starting from $\gamma_1 = 52314$, γ_m is obtained by inserting a new occurrence $P_1 = 2, 3m+1, 1$ of the pattern 231 between the first and the second element of γ_{m-1}, then suitably rescaling the remaining elements. We have the following result:

Proposition 18.

1. *γ_i is a pattern of γ_{i+1}, for each $i \geqslant 1$.*
2. *$\gamma_i \in Sort_2^{(qlg)}$ if and only if i is even.*

The existence of an infinite chain of permutations which are alternately sortable and nonsortable suggests that it should be quite difficult to obtain a simple characterization of $Sort_2^{(qlg)}$; it is also conceivable that it should be possible to adapt the above proposition to larger values of k, thus obtaining similar (negative) results.

5 Final remarks

In the present work we started the analysis of a sorting device consisting of k decreasing stacks followed by an increasing one, generalizing the case $k = 1$ addressed in [18]. In general, the problem of characterizing sortable permutations in terms of forbidden patterns seems quite hard, due to the fact that the basis is infinite, as shown in Theorem 4. We have however been able to describe an optimal algorithm in the case $k = 2$ which can sort every sortable permutation. Such an algorithm employs a strategy which is surely nontrivial. Thus we have also briefly discussed some simpler algorithms, which are not able to sort all sortable permutations but are certainly simpler to describe.

There are of course several items that remain to be investigated. In his dissertation [20], West found some examples of permutations that have the same number of preimages under the usual Stacksort algorithm. Defant, Engen and Miller consider permutations with exactly one preimage under Stacksort [10]. Investigating preimages would be interesting for our machines as well, in particular for the left-greedy $\mathfrak{D}^k\mathfrak{I}$ machine. It could also be interesting to study the image of the left-greedy $\mathfrak{D}^k\mathfrak{I}$ machine. Bousquet-Mélou did this for Stacksort [5].

Some further suggestions are the following:

- determine the complexity of the optimal algorithm for the $\mathfrak{D}^2\mathfrak{I}$ machine;
- enumerate sortable permutations, both in the general case and in the restricted (left greedy and quasi left greedy) cases;
- study the machine consisting of two passes through the $\mathfrak{D}\mathfrak{I}$ machine described in [18]: are there analogies with West 2-stack-sortable permutations?

Acknowledgements

We are very grateful to the referee for a careful reading and many useful suggestions, which have improved the overall presentation.

References

[1] M. Albert, *PermLab: Software for Permutation Patterns*, at http://www.cs.otago.ac.nz/staffpriv/malbert/permlab.php

[2] M. Albert, M. Bousquet-Mélou, *Permutations sortable by two stacks in parallel and quarter plane walks*, European J. Combin., 43 (2015) 131–164.

[3] M. Bona, *A survey of stack sorting disciplines*, Electron. J. Combin., 9(2) (2002-2003) #A1.

[4] M. Bona, *Combinatorics of Permutations*, Discrete Mathematics and Its Applications, CRC Press, 2004.

[5] M. Bousquet-Mélou, *Sorted and/or sortable permutations*, Discrete Math., 225 (2000) 25–50.

[6] G. Cerbai, L. Cioni, L. Ferrari, *Stack sorting with increasing and decreasing stacks*, arXiv:1910.03578.

[7] G. Cerbai, A. Claesson, L. Ferrari, *Stack sorting with restricted stacks*, arXiv:1907.08142.

[8] C. Defant, *Postorder preimages*, Discrete Math. Theor. Comput. Sci., 19 (2017) #3, 15 pp.

[9] C. Defant, *Counting 3-stack-sortable permutations*, arXiv:1903.09138.

[10] C. Defant, M. Engen, J. A. Miller, *Stack-sorting, set partitions, and Lassalle's sequence*, arXiv:1809.01340.

[11] S. Kitaev, *Patterns in permutations and words*, Monographs in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg, 2011.

[12] D. E. Knuth, *The art of computer programming, vol. 1, Fundamental Algorithms*, Addison-Wesley, Reading, Massachusetts, 1973.

[13] D. Kremer, *Permutations with forbidden subsequences and a generalized Schröder number*, Discrete Math., 218 (2000) 121–130.

[14] M. Murphy, *Restricted permutations, antichains, atomic classes, and stack sorting*, Doctoral Thesis, University of St Andrews, 2002.

[15] A. Pierrot, D. Rossin, *2-stack sorting is polynomial*, Theory Comput. Syst., 60 (2017) 552–579.

[16] N. J. A. Sloane, *The On-Line Encyclopedia of Integer Sequences*, at oeis.org.

[17] R. Smith, *Comparing algorithms for sorting with t stacks in series*, Ann. Comb., 8 (2004) 113–121.

[18] R. Smith, *Two stacks in series: a decreasing stack followed by an increasing stack*, Ann. Comb., 18 (2014) 359–363.

[19] R. E. Tarjan, *Sorting using networks of queues and stacks*, Journal of the ACM, 19 (1972) 341–346.

[20] J. West, *Permutations with forbidden subsequences and stack-sortable permutations*, PhD thesis, Massachusetts Institute of Technology, 1990.

[21] J. West, *Sorting twice through a stack*, Theoret. Comput. Sci., 117 (1993) 303–313.