2-Stack Sorting is polynomial *

Adeline Pierrot[1] and Dominique Rossin[2]

1 **Adeline Pierrot**
Institute of Discrete Mathematics and Geometry, TU Wien, Austria
2 **Dominique Rossin**
LIX UMR 7161, École Polytechnique and CNRS, 91128 Palaiseau, France

Abstract

This article deals with deciding whether a permutation is sortable with two stacks in series. Whether this decision problem lies in P or is NP-complete is a longstanding open problem since the introduction of serial compositions of stacks by Knuth in *The Art of Computer Programming* [6] in 1973. We hereby prove that this decision problem lies in P by giving a polynomial algorithm to solve it. This algorithm uses the concept of pushall sorting, which was previously defined and studied by the authors in [8, 9].

1998 ACM Subject Classification G.2.1 Permutations, F.2.2 Sorting, E.1 Lists, stacks, and queues

Keywords and phrases permutation, stack, sort, NP-complete

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Stack sorting has been studied first by Donald Knuth in the sixties (see volume 1 of *The Art of Computer Programming* [5]). One of its theoretical interests is that it is a linear time sort. Its drawback is that it cannot sort all permutations.

Characterizing the stack-sortable permutations is a historical problem, which led to define permutation *patterns* and permutations *classes* closed by excluded pattern, an active research domain in combinatorics (see the book [4]). Stack-sorting was then generalized by Tarjan, who introduced sorting networks [10] allowing to sort more permutations, and many variations of this problem have been studied afterwards (see [3] for a summary).

Here we study the decision problem "Is a given permutation σ sortable by two stacks connected in series?". It is cited many times in the literature: in [3], Bóna gives a summary of advances on stack-sorting and mentions this problem as possibly NP-complete; more recently, it is also cited as possibly NP-complete in [1]. Surprisingly, both conjectures exist: in [2], the authors conjecture it is NP-complete, while Murphy in his PhD thesis [7] conjectures it is polynomial.

In this article, we solve this problem that stayed open for several decades by giving a polynomial decision algorithm. Details of the proofs can be found in [8].

The difficulty of this problem, whose statement is however very simple, lies in the fact that both stacks are considered at once, which gives a great liberty on which operation to apply on the permutation at each step, and yields an exponential naive algorithm.

There are two key ideas in this article: first, limit the number of sortings to consider by proving that if a permutation σ is sortable, then there exists a sorting process of σ respecting some condition denoted $\mathcal{P}$. Second, encode a possibly exponential number of sortings by a sequence of graphs called sorting graphs, using pushall stack configurations introduced in [9, 8].

The article is organized as follows: Section 2 studies general properties of two-stack sorting thanks to stack words and stack configurations and limits the number of sortings to consider by introducing Property $\mathcal{P}$. Section 3 introduces the sorting graph $\mathcal{G}^{(i)}$ which encodes possible stack configurations at a given time t_i and gives an algorithm to compute this graph iteratively for all i from 1 to the number of right-to-left minima, leading to an algorithm deciding whether a permutation is 2-stack sortable. Then Section 4 proves that the resulting algorithm is polynomial.

* This work was completed with the support of the ANR project ANR BLAN-0204_07 MAGNUM and of the SFB project SFB F50 Algorithmic and Enumerative Combinatorics.

2 Study of two-stack sorting processes

2.1 Definitions and general problem statement

Let us recall the problem of sorting a permutation with two stacks in series. A permutation of size n is a word of n letters $\sigma = \sigma_1\sigma_2 \dots \sigma_n$ on the alphabet $[1..n]$ containing each letter from 1 to n exactly once. Given two stacks H and V in series (see Figure 1) and a permutation σ, we want to sort the elements of σ using the stacks. We take σ as input: the elements σ_i will be read one by one, beginning with σ_1 and ending with σ_n. We have three different operations (see Figure 1):

ρ: Take the next element of σ still in the input and push it on top of the first stack H.
λ: Pop the topmost element of stack H and push it on top of the second stack V.
μ: Pop the topmost element of stack V and write it to the output.

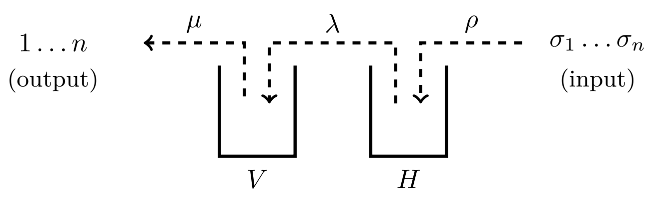
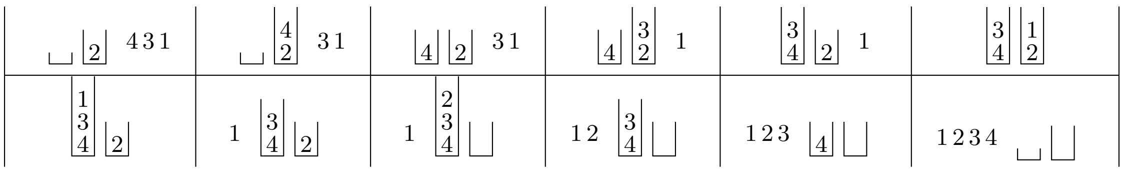


Figure 1 Sorting with two stacks in series.

If there is a sequence $w = w_1 \dots w_k$ of operations ρ, λ, μ leading to the identity $1 \dots n$ as output, the permutation σ is said 2-stack sortable. In that case, we define the sorting word associated to this sorting process as the word w on the alphabet $\{\rho, \lambda, \mu\}$. Note that w must have n times each letter ρ, λ and μ and thus $k = 3n$. For example, 2431 is sortable using the following process:



This sorting process is encoded by the word $w = \rho\rho\lambda\rho\lambda\rho\lambda\mu\lambda\mu\mu\mu$. We can also decorate the word to specify the element on which each operation is performed. The *decorated word* for w and 2431 is $\hat{w} = \rho_2\rho_4\lambda_4\rho_3\lambda_3\rho_1\lambda_1\mu_1\lambda_2\mu_2\mu_3\mu_4$. Note that we have the same information in (σ, w) and in $\hat{w}$. Nevertheless, in a decorated word each letter ρ_i, λ_i or μ_i appears only once. The decorated word associated to (σ, w) is denoted $\hat{w}^\sigma$.

Not all permutations are 2-stack sortable. The smallest non-sortable ones are of size 7, for instance $\sigma = 2435761$. The question of interest here is to decide whether a permutation is sortable.

There is a naive algorithm to solve this question: given a permutation σ of size n, a sorting process corresponds to a word on the alphabet $\{\rho, \lambda, \mu\}$ of size $3n$. It is thus enough to test all words of size $3n$ and check if one of them yields the identity permutation on the output when taking σ as input. But this decision algorithm is exponential since there are 3^{3n} words to test.

The number of words to test can be reduced by noting that not all words correspond to a sorting process: a necessary condition is to contains n times each letter. But some permutation have an exponential number of sorting processes. For instance, it is easy to see that the decreasing permutation $n(n-1) \dots 1$ admits 2^{n-1} sorting processes.

A natural solution would be to define a canonical sorting process among all possible sorting processes of a permutation, to be able to test only this one, but researches in this direction have been unsuccessful. Several greedy algorithms for 2-stack sorting have been defined, (cf. [11] and [2]) but none is able to sort all 2-stack sortable permutations. A key idea of our polynomial algorithm is to limit the number of sortings to consider by studying stack words and stack configurations.

2.2 Stack words and stack configurations

We saw that a sorting process can be described as a word on the alphabet $\{\rho, \lambda, \mu\}$. However not all words on the alphabet $\{\rho, \lambda, \mu\}$ describe sorting processes.

▶ **Definition 1** (stack word and sorting word)**.** Let w be a word on the alphabet $\{\rho, \lambda, \mu\}$ and $\alpha \in \{\rho, \lambda, \mu\}$. Then $|w|_\alpha$ denotes the number of occurrences of α in w.

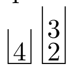
A *stack word* is a word $w \in \{\rho, \lambda, \mu\}^*$ such that for any prefix v of w, $|v|_\rho \geq |v|_\lambda \geq |v|_\mu$.

A *sorting word* is a stack word w such that $|w|_\rho = |w|_\lambda = |w|_\mu$.

For any permutation σ, a sorting word *for* σ is a sorting word encoding a sorting process with σ as input (leading to the identity of size $|\sigma|$ as output).

Intuitively, stack words describe a sequence of operations ρ, λ, μ that can be carried out starting with empty stacks (and an arbitrarily long input), whereas sorting words are words encoding a complete sorting process (stacks are empty at the beginning and at the end of the process).

Another way of describing sorting processes is, instead of focusing on the operations made, to focus on the description of which element lies in each stack (and their order in the stacks) at each step of the process. Such a description for one step is called a *stack configuration.* For example



is a stack configuration which is a part of the sorting process $\rho\rho\lambda\rho\lambda\rho\lambda\mu\lambda\mu\mu\mu$ of 2431.

Stack configurations and stack words describing a sorting process are linked:

▶ **Definition 2.** Let w be a stack word. Starting with a permutation σ as input, the stack configuration reached after performing operations described by the word w is denoted $c_\sigma(w)$.
A stack configuration c is *reachable* for σ if there exists a stack word w such that $c = c_\sigma(w)$. In other words, a stack configuration is reachable for σ if there exists a sequence of operations ρ, λ, μ leading to this configuration with σ as input.
A stack configuration is *poppable* if elements in stacks H and V can be output in increasing order using operations λ and μ.

Any stack configuration which is a part of a sorting process of a permutation σ has to be reachable for σ and poppable. We now describe necessary or sufficient conditions for a stack configuration to be reachable or poppable.

▶ **Lemma 3.** *Let c be a stack configuration. If c is poppable, then the elements of V are in decreasing order (concerning their values) from bottom to top. If c is reachable for a permutation σ, then the elements of H have increasing indices (as letters of σ) from bottom to top.*

Poppable stack configurations have been characterized in [9] by the following Lemma. Recall first that a permutation $\pi = \pi_1\pi_2 \ldots \pi_k$ is a *pattern* of $\sigma = \sigma_1\sigma_2 \ldots \sigma_n$ if there exists indices $1 \leq i_1 < i_2 < \ldots < i_k$ such that $\sigma_{i_1}\sigma_{i_2}\sigma_{i_3} \ldots \sigma_{i_k}$ is order-isomorphic to π.

▶ **Lemma 4.** *A stack configuration c is poppable if and only if :*
- *Stack V does not contain the pattern 12 (seen from bottom to top).*
- *Stack H does not contain the pattern 132 (seen from bottom to top).*
- *Stacks (V, H) do not contain the pattern $|2|13|$.*

Plus, there is a unique way to pop the elements out in increasing order in terms of stack operations.

The first two conditions are usual pattern relations (note that the first one corresponds to the first part of Lemma 3). The third one means that there are no elements i, j, k with i in V and j, k in H (k above j) such that $j < i < k$.

Most of the time, a stack configuration is associated to a permutation implying that the elements in the stacks are a subset of those of the permutation. In particular a *total stack configuration of σ* is a stack configuration in which the elements of the stacks are exactly all those of σ.

▶ **Definition 5** (pushall configuration)**.** A stack configuration is a *pushall* stack configuration of σ if it is poppable, total and reachable for σ.

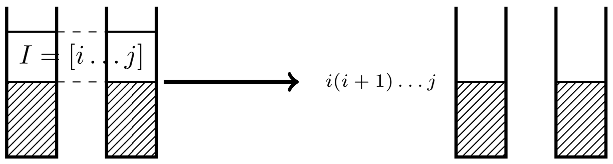
Pushall stack configurations, which were defined and studied in [8] and [9], play a key role in our polynomial algorithm. Indeed, a permutation which ends with its smallest element is 2-stack sortable if and only if it admits a pushall stack configuration. Moreover we have:

▶ **Theorem 6** ([8, 9])**.** *One can compute in time $O(n^2)$ the set of pushall stack configurations of any permutation of size n.*

2.3 Restrict the number of sortings to focus on: Property (P)

Some permutations have an exponential number of sorting processes. To obtain a polynomial algorithm, we restrict the number of sortings to focus on. The following lemma shows that we can focus on sorting processes such that the smallest elements are popped out "as soon as possible".

▶ **Lemma 7.** *Let σ be a 2-stack sortable permutation and $w = uv$ be a sorting word for σ. Assume that after performing the operations of u, the elements $1 \dots i-1$ have been output and the elements $i \dots j$ are at the top of the stacks. Then there exists a sorting word $w' = uu'u''$ for σ such that u' consists only of moving the elements $i \dots j$ from the stacks to the output in increasing order without moving any other elements.*



Now we add some other constraints on the sortings, using the block-decomposition of permutations. A block B of a permutation $\sigma = \sigma_1\sigma_2 \dots \sigma_n$ is a factor $\sigma_i\sigma_{i+1} \dots \sigma_j$ of σ such that the set of values $\{\sigma_i, \dots, \sigma_j\}$ is an interval. Note that by definition of a factor, the set of indices $\{i, \dots, j\}$ is also an interval. Given two blocks B and B' of σ, we say that $B < B'$ if and only if $\sigma_i < \sigma_j$ for all $\sigma_i \in B$, $\sigma_j \in B'$. A permutation σ is $\ominus$-decomposable if it can be written as $\sigma = B_1 \dots B_k$ such that $k \geq 2$ and for all i, $B_i > B_{i+1}$ in terms of blocks. Otherwise we say that σ is $\ominus$-indecomposable. When each B_i is $\ominus$-indecomposable, we write $\sigma = \ominus[B_1, \dots, B_k]$ and call it the $\ominus$-decomposition of σ. Note that we do not renormalize the elements of B_i, thus, except B_k, the B_i are not permutations. Nevertheless, B_i can be seen as a permutation by subtracting $|B_{i+1}| + \cdots + |B_k|$ to all its elements.

The RTL (right-to-left) minima of a permutation are the elements σ_k such that there is no j with $j > k$ and $\sigma_j < \sigma_k$. We denote by σ_{k_i} the ith RTL minimum of σ. If σ has r RTL minima, then $\sigma = \dots \sigma_{k_1} \dots \sigma_{k_2} \dots \sigma_{k_r}$ with $\sigma_{k_1} = 1$ and $k_r = n$.

Take for example the permutation $\sigma = 6\,5\,8\,7\,4\,1\,3\,2$. The $\ominus$-decomposition of σ is $\sigma = \ominus[6\,5\,8\,7, 4, 1\,3\,2]$. Furthermore, σ has 2 RTL-minima which are $\sigma_6 = 1$ and $\sigma_8 = 2$.

We denote $\sigma^{(i)} = \{\sigma_j \mid j < k_i \text{ and } \sigma_j > \sigma_{k_i}\}$ the restriction of σ to elements in the upper left quadrant of the ith RTL minimum σ_{k_i}. The $\ominus_i$-decomposition of σ is the $\ominus$-decomposition of $\sigma^{(i)} = \ominus[B_1^{(i)}, \dots, B_{s_i}^{(i)}]$. In the following, s_i always denotes the number of blocks of $\sigma^{(i)}$ and $B_j^{(i)}$ the jth block in the $\ominus_i$-decomposition.

We note $A^{(i)}$ the common part of the permutations $\sigma^{(i)}$ and $\sigma^{(i+1)}$, i.e., $A^{(i)} = \sigma^{(i)} \bigcap \sigma^{(i+1)} = \{\sigma_j \mid j < k_i \text{ and } \sigma_j > \sigma_{k_{i+1}}\}$. This sub-permutation $A^{(i)}$ intersects $\ominus$-indecomposable blocks of $\sigma^{(i)}$ and $\sigma^{(i+1)}$. Let $p^{(i)}$ (resp. $q^{(i+1)}$) be the index such that $B_{p^{(i)}}^{(i)}$ (resp. $B_{q^{(i+1)}}^{(i+1)}$) contains the smallest value of $A^{(i)}$. Let $D^{(i)} = \left(B_{p^{(i)}}^{(i)} \bigcup B_{q^{(i+1)}}^{(i+1)}\right) \bigcap A^{(i)}$ (see Figure 2).

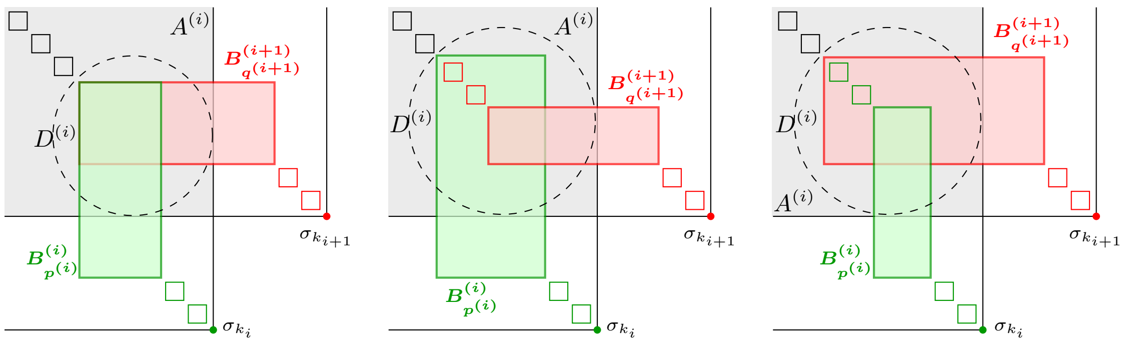


Figure 2 The $\ominus$-decomposition of $\sigma^{(i)}$ and of $\sigma^{(i+1)}$ visualized in the diagram of σ (set of the points at coordinates (i, σ_i)) resp. when $p^{(i)} = q^{(i+1)}$, $p^{(i)} < q^{(i+1)}$ and $p^{(i)} > q^{(i+1)}$).

▶ **Definition 8** (Properties (P_i) and (P))**.** Let σ be a permutation and w a sorting word for σ. We say that w verifies (P_i) if and only if the corresponding decorated word $\hat{w}$ satisfies:

(i) μ_{σ_j} appears before $\rho_{\sigma_{k_i}}$ for all $\sigma_j < \sigma_{k_i}$,
(ii) $\rho_{\sigma_{k_i}}\lambda_{\sigma_{k_i}}\mu_{\sigma_{k_i}}$ is a factor of $\hat{w}$,
(iii) All operations μ_{σ_ℓ} with $\sigma_\ell \in B_j^{(i)}$ and $j \in [p^{(i)}+1..s_i]$ appear before $\rho_{\sigma_{(k_i)+1}}$ in $\hat{w}$.

If a word w verifies Property (P_i) for all i then we say that w verifies Property (P).
We call t_i the time just before σ_{k_i} enters stack H.

▶ **Theorem 9.** *If σ is 2-stack sortable then there is a sorting word of σ satisfying Property (P). In particular, in the sorting process encoded by this word, the elements in the stacks at time t_i are exactly those of $\sigma^{(i)}$.*

Theorem 9 is proved recursively using the following lemmas:

▶ **Lemma 10** (easy)**.** *If the sorting word encoding a sorting process of σ verifies Property (P_i), then the elements in the stacks at time t_i are exactly those of $\sigma^{(i)}$.*

▶ **Lemma 11** (Follows from Lemma 3)**.** *If $\sigma = \ominus[B_1,\dots B_k]$ then in any poppable stack configuration reachable for σ, for all $i<j$, the elements of B_i are, in the stacks, below the elements of B_j.*

▶ **Lemma 12.** *Let w be a sorting word for a permutation σ, r be the number of RTL-minima of σ and $\ell \in [1..r]$. If w verifies (P_i) for $i \in [1..\ell-1]$ then there exists a sorting word w' for σ that verifies (P_i) for $i \in [1..\ell]$.*

The proof of this last lemma involves Lemma 7 and Lemma 11.

Theorem 9 ensures that if a permutation σ is sortable then there exists a sorting in which at each time step t_i, the elements in the stacks are exactly those of $\sigma^{(i)}$. Such a stack configuration is then a pushall stack configuration of $\sigma^{(i)}$. Thus if σ is sortable, then for all i, $\sigma^{(i)}$ has to admit a pushall stack configuration. This necessary condition is not sufficient: the pushall stack configuration for $\sigma^{(i)}$ has to be *accessible* from the one of $\sigma^{(i-1)}$. This is formalized below.

2.4 Stack configurations and accessibility

The stack configurations for a sorting process encode the elements that are currently in the stacks. But some elements are still waiting in the input and some elements have been output. To fully characterize a configuration, we define an *extended* stack configuration of a permutation σ of size n to be a pair (c,i) where $i \in \{1,\dots,n+1\}$ and c is a poppable stack configuration made of all elements within $\sigma_1,\sigma_2,\dots,\sigma_{i-1}$ that are greater than a value p. The elements $\sigma_i,\dots,\sigma_n$ are still in the input and the elements $\sigma_j < p, j < i$ have already been output. Note that we don't ask the configuration to be reachable.

▶ **Definition 13.** Let σ be a permutation and (c,i) be an extended stack configuration of σ. Then an extended stack configuration (c',j) of σ is *accessible* from (c,i) if the stack configuration (c',j) can be reached starting from (c,i) and performing operations ρ,λ and μ such that the elements of $c \cup \{\sigma_i \dots \sigma_n\}$ that are output by the operations μ performed are output in increasing order.

For example, for $\sigma = 2\,3\,1\,6\,5\,8\,4\,7$, the sequence of operations $\mu_2\mu_3\rho_6\rho_5\rho_8\lambda_8$ proves that $(\begin{array}{|c|}8\\\hline\end{array}\ \begin{array}{|c|}5\\6\\\hline\end{array},7)$ is accessible from $(\begin{array}{|c|}2\\3\\\hline\end{array}\ \begin{array}{|c|}\ \\\hline\end{array},4)$. But $(\begin{array}{|c|}2\\3\\\hline\end{array}\ \begin{array}{|c|}6\\\hline\end{array},5)$ is not accessible from $(\begin{array}{|c|}1\\2\\\hline\end{array}\ \begin{array}{|c|}3\\\hline\end{array},4)$.

In the following, given two total pushall stack configurations c and c', corresponding to stack configuration of $\sigma^{(i)}$ and $\sigma^{(i+1)}$, we study conditions for c' to be accessible from c, (i.e. conditions for being able to move elements, starting from c at time t_i, to obtain c' at time t_{i+1}).

▶ **Lemma 14.** *Let (c,k_i), resp. (c',k_{i+1}), be a pushall stack configuration of $\sigma^{(i)}$, resp. $\sigma^{(i+1)}$. Let $\pi = \sigma_{|B^{(i)}_{p^{(i)}} \bigcup B^{(i+1)}_{q^{(i+1)}}}$. Then (c',k_{i+1}) is accessible from (c,k_i) for σ if and only if:*

1. *$(c'_{|\pi}, |\pi|+1)$ is accessible from $(c_{|\pi}, \#(D^{(i)} \bigcup B^{(i)}_{p^{(i)}})+1)$ for π (see Figure 2).*

2. $\forall j < \min(p^{(i)}, q^{(i+1)})$, $c_{|B_j^{(i)}} = c'_{|B_j^{(i)}}$.

3. $\forall j > q^{(i+1)}$, $c'_{|B_j^{(i+1)}}$ *is a pushall configuration of* $\sigma_{|B_j^{(i+1)}}$.

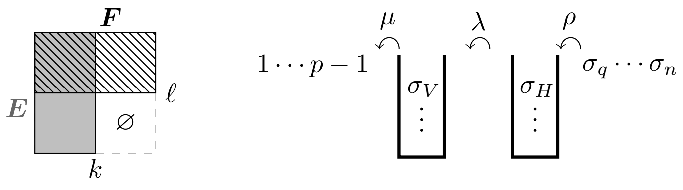
Informally, it is possible to efficiently decide whether a configuration at time t_i can evolve into a given configuration at time t_{i+1}. Moreover, during this transition, only a few operations are undetermined: the largest elements don't move, the smallest ones are output in increasing order, and the remaining ones form a $\ominus$-indecomposable permutation. This will allow us to exhibit a polynomial algorithm checking accessibility. The proof of Lemma 14 relies on Lemma 7, Lemma 11 and the following lemma:

▶ **Lemma 15.** *Let $\sigma_\ell \in A^{(i)}$. During a sorting process of σ, the elements σ_m such that $\sigma_m > \sigma_\ell$ and $m < \ell$ do not move between t_i and t_{i+1}.*
Idea of the proof : σ_ℓ prevent those elements from moving.

Thanks to Lemma 14, if c and c' are two total pushall stack configurations corresponding to stack configurations of $\sigma^{(i)}$ and $\sigma^{(i+1)}$, to decide whether c' is accessible from c it is enough to check three kind of conditions. The last two ones are easy to check, and the first one can be checked using the following lemma:

▶ **Lemma 16.** *Let σ be a permutation of size n and (c, i), (c', j) two extended stack configurations of σ with $i < j$. Let E (resp. F) be the set of elements of c (resp. c').*
- *If there exists $k, \ell \in \{1 \dots n\}$ such that $E = \{\sigma_m \mid m \leq k\}$ and $F = \{\sigma_m \mid \sigma_m \geq \ell\}$,*
- *if moreover $E \cup F = \sigma$,*

then we can decide in linear time whether (c', j) is accessible from (c, i) using Algorithm 1.



Algorithm 1: isAccessible$\big((c,i),(c',j),\sigma\big)$

```
Data: $\sigma$ a permutation and $(c,i),(c',j)$ two stack configurations of $\sigma$ satisfying conditions of
      Lemma 16
Result: true or false depending on whether the configuration $c'$ is accessible from $c$
Put configuration $c$ in the stacks $H$ and $V$
$p \leftarrow$ the smallest element of $c \cup \{\sigma_i \dots \sigma_n\}$ (next element to be output)
$q \leftarrow i$ (next index of $\sigma$ that must enter the stacks)
We denote by $V(c')$ the set of elements of $V$ in configuration $c'$ and by $\sigma_V$ the top of $V$ in
the current configuration (the same goes for $H$).
while $q < j$ or $p < \ell$ or $\sigma_H \in V(c')$ do
    if $\sigma_V = p$ then Perform $\mu$; $p \leftarrow p + 1$ else
        if $\sigma_H < \ell$ then Perform $\lambda$ else
            if $H = \varnothing$ or $\sigma_H \in H(c')$ then Perform $\rho$; $q \leftarrow q + 1$ else
                if $\sigma_q \in H(c')$ or $\sigma_H > \sigma_q$ then Perform $\lambda$ else Perform $\rho$; $q \leftarrow q + 1$
Return $(H, V) == c'$
```

The proof of Lemma 16 relies on Lemma 4 and Lemma 7. The idea is that Algorithm 1 perform only operations that we have to do to obtain (c', j) starting from (c, i). Thus (c', j) is accessible from (c, i) if and only if the configuration obtained at the end of the algorithm is c'.

3 An iterative algorithm

3.1 A first naive algorithm

From Theorem 9, a permutation σ is 2-stack sortable if and only if it admits a sorting process satisfying Property (P). The main idea is to compute the set of sorting processes of σ satisfying

Property (P) and decide whether σ is 2-stack sortable by testing the emptiness of this set.

Verifying (P) means verifying (P_j) for all j from 1 to r, r being the number of right-to-left minima (whose indices are denoted k_j). The algorithm proceeds in r steps: for all i from 1 to r we iteratively compute the sorting processes of $\sigma_{\leq k_i}$ verifying (P_ℓ) for all ℓ from 1 to i (with $\sigma_{\leq k_i} = \sigma_1 \ldots \sigma_{k_i}$). As $\sigma_{\leq k_r} = \sigma$, the last step gives sorting processes of σ satisfying Property (P).

By "compute the sorting processes of $\sigma_{\leq k_i}$" we mean "compute the stack configuration just before σ_{k_i} enters the stacks in such a sorting process":

▶ **Definition 17.** We call P_i-stack configuration of σ a stack configuration $c_\sigma(w)$ for which there exists u such that the first letter of u is $\rho_{\sigma_{k_i}}$ and wu is a sorting word of $\sigma_{\leq k_i}$ verifying (P) for $\sigma_{\leq k_i}$ (that is, verifying (P_ℓ) for all ℓ from 1 to i).

The algorithm is based on the following two lemmas:

▶ **Lemma 18** (Consequence of Theorem 9). *For any i from 1 to r, $\sigma_{\leq k_i}$ is 2-stack sortable if and only if the set of P_i-stack configurations of σ is nonempty. In particular, σ is 2-stack sortable if and only if the set of P_r-stack configurations of σ is nonempty.*

▶ **Lemma 19** (Consequence of Lemma 10). *Any P_i-stack configuration of σ is a pushall stack configuration of $\sigma^{(i)}$, accessible from some P_{i-1}-stack configurations of σ.*

As explained above, the algorithm proceeds in r steps such that after step i we know every P_i-stack configuration of σ and we want to compute the P_{i+1}-stack configurations of σ at step $i+1$. As configurations for $i+1$ are a subset of pushall stack configurations of $\sigma^{(i+1)}$, a possible algorithm is to take every pair of configurations (c, c') with c being a P_i-stack configuration of σ (computed at step i) and c' be any pushall stack configuration of $\sigma^{(i+1)}$ (given by Algorithm 5 of [9], see Theorem 6). Then we can use Algorithm 1 to decide whether c' is accessible from c for σ. This leads to an algorithm deciding whether a permutation σ is 2-stack sortable, but this algorithm is not polynomial. Indeed, the number of P_i-stack configurations of σ is possibly exponential. However, this set can be described by a polynomial representation as a graph.

3.2 Towards the sorting graph

We now explain how to adapt the previous idea to obtain a polynomial algorithm. Instead of computing all P_i-stack configurations of σ (which are pushall stack configurations of $\sigma^{(i)}$), we compute the restriction of such configurations to blocks $B_j^{(i)}$ of the $\ominus$-decomposition of $\sigma^{(i)}$. By Lemma 11, those configurations are stacked one upon the others to give a P_i-stack configuration. The stack configurations of any block $B_j^{(i)}$ are labeled with an integer which is assigned when the configuration is computed. Those pairs (configurations, integer) will be the vertices of the graph $\mathcal{G}^{(i)}$ which we call a *sorting graph*, the edges of which representing the configurations that can be stacked one upon the other. Vertices of the graph $\mathcal{G}^{(i)}$ are partitioned into levels corresponding to blocks $B_j^{(i)}$. The integer labels allows us to ensure the polynomiality of the representation. Indeed, a given label can only appear once per level of the graph $\mathcal{G}^{(i)}$. As those labels are assigned to configurations when they are created, each label corresponding to a pushall stack configuration, from Theorem 4.4 of [9] there are at most $9|\sigma|$ distinct labels thus at most $9|\sigma|$ vertices per level of the graph $\mathcal{G}^{(i)}$. This is formalized in Lemma 22. The label can be seen as the memory of the configuration that encodes its history since it has been created: two configurations having the same label come from the same initial pushall configuration.

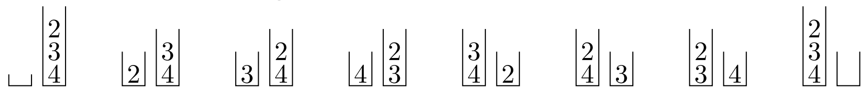
More precisely, a sorting graph $\mathcal{G}^{(i)}$ for a permutation σ and an index i verifies:

- Vertices of $\mathcal{G}^{(i)}$ are partitioned into s_i subsets $V_j^{(i)}$ with $j \in [1 \ldots s_i]$ called levels.
- For any $j \in [1 \ldots s_i]$, the number of vertices in level $V_j^{(i)}$ is less than $9|\sigma|$.
- Each vertex $v \in \mathcal{G}^{(i)}$ is a pair $(c, \textcircled{\ell})$ with c a stack configuration and $\textcircled{\ell}$ an index called *configuration index.*
- All configuration indices are distinct inside a graph level $V_j^{(i)}$.
- $(c, \textcircled{\ell}) \in V_j^{(i)} \Rightarrow c$ is a pushall stack configuration of $B_j^{(i)}$ accessible for σ.

- There are edges only between vertices of adjacent levels $V_j^{(i)}$, $V_{j+1}^{(i)}$ (this implies Lemma 23).
- Paths between vertices of $V_1^{(i)}$ and $V_{s_i}^{(i)}$ correspond to stack configurations of $\sigma^{(i)}$. Precisely, and that is why the algorithm is correct, *such paths are in bijection with the P_i-stack configurations of σ* (i.e., stack configurations corresponding to a sorting of $\sigma_{\leq k_i}$ satisfying (P) just before σ_{k_i} is pushed to H) by stacking one upon the other the configurations of the vertices of a path.
- For any vertex v of $\mathcal{G}^{(i)}$, there is a path between vertices of $V_1^{(i)}$ and $V_{s_i}^{(i)}$ going through v.

Take for example the permutation $\sigma = 4321$. There is only one right-to-left minimum, which is 1. The sorting graph $\mathcal{G}^{(1)}$ for $\sigma = 4321$ encodes pushall stack configurations of $\sigma^{(1)} = 432$, corresponding to stack configurations just *before* 1 enters H.

There are 8 different such configurations, which are:



As the $\ominus$-decomposition of $\sigma^{(1)}$ is $\sigma^{(1)} = \ominus[4,3,2]$, the sorting graph $\mathcal{G}^{(1)}$ has 3 levels (Fig. 3).

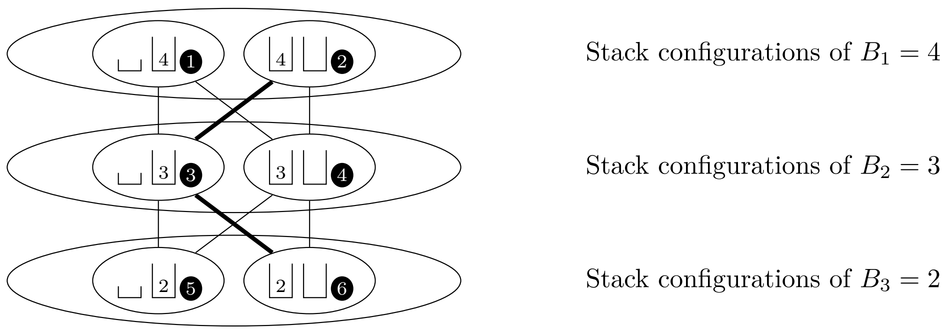
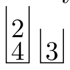

1
2
3
4
5
6
Stack configurations of $B_1 = 4$
Stack configurations of $B_2 = 3$
Stack configurations of $B_3 = 2$

Figure 3 Sorting graph $\mathcal{G}^{(1)}$ of $\sigma = 4321$.

Then the 8 configurations of σ are found taking each of the 8 different paths going from any configuration of B_1 to any configuration of B_3. For example, in Figure 3, the thick path gives the stack configuration  by stacking the selected configuration of B_3 above the configuration of B_2 and so on.

Our algorithm computes iteratively the graph $\mathcal{G}^{(i)}$ from the graph $\mathcal{G}^{(i-1)}$ for all i from 2 to r. The way $\mathcal{G}^{(i)}$ is computed from $\mathcal{G}^{(i-1)}$ depends on the relative values of $p^{(i)}$ and $q^{(i+1)}$. By definition of $\mathcal{G}^{(i)}$, if at any step $\mathcal{G}^{(i)}$ is empty, it means that $\sigma_{\leq k_i}$ is not sortable (from Theorem 9), so σ is not sortable either, and the algorithm returns `false`. This is summarized in Algorithm 2.

Algorithm 2: *isSortable*

```
Data: $\sigma$ a permutation
Result: true or false depending on whether $\sigma$ is 2-stack sortable
$\mathcal{G} \leftarrow ComputeG1$
for $i$ from 2 to $r$ do
    if $p^{(i)} = q^{(i+1)}$ then $\mathcal{G} \leftarrow iteratepEqualsq(\mathcal{G})$ or return false
    else
        if $p^{(i)} < q^{(i+1)}$ then $\mathcal{G} \leftarrow iteratepLessThanq(\mathcal{G})$ or return false
        else $\mathcal{G} \leftarrow iteratepGreaterThanq(\mathcal{G})$ or return false
return true
```

In the next subsections we describe the sub-procedures used in our main algorithm *isSortable*(σ).

3.3 First step: $\mathcal{G}^{(1)}$

In this subsection, we show how to compute the P_1-stack configurations of σ, that is, the stack configurations corresponding to time t_1 for sorting words of $\sigma_{\leq k_1}$ that satisfy (P) for $\sigma_{\leq k_1}$.

From Lemma 19, such a stack configuration is a pushall stack configuration of $\sigma^{(1)}$. Conversely, since $\sigma_{k_1} = 1$, $\sigma^{(1)} = \sigma_{<k_1}$ and each sorting word of $\sigma_{\leq k_1}$ satisfies (P_1) for $\sigma_{\leq k_1}$. Thus the set of P_1-stack configurations of σ is the set of pushall stack configurations of $\sigma^{(1)}$.

By Proposition 4.7 of [9], these stack configurations are described by the set of stack configurations for each block of the $\ominus$-decomposition of $\sigma^{(1)}$. More precisely, with $\sigma^{(1)} = \ominus[B_1^{(1)}, \dots, B_{s_1}^{(1)}]$, there is a bijection from $pushallConfigs(B_1^{(1)}) \times \cdots \times pushallConfigs(B_{s_1}^{(1)})$ onto $pushallConfigs(\sigma^{(1)})$ by stacking configurations one upon the other (as in Lemma 11). As a consequence, from Lemma 18 $\sigma_{\leq k_1}$ is not sortable if and only if a set $pushallConfigs(B_j^{(1)})$ is empty.

Moreover, it will be useful to label the configurations computed so that we attach a distinct integer to each stack configuration when computed.

At this point, we have encoded all configurations corresponding to words satisfying P up to the factor $\rho_1 \lambda_1 \mu_1$.

The obtained graph is $\mathcal{G}^{(1)}$. This step is summarized in Algorithm 3.

Algorithm 3: ComputeG1

```
Data: $\sigma$ a permutation, $num$ a global integer variable
Result: false if $\sigma_{\leq k_1}$ is not sortable, the sorting graph $\mathcal{G}^{(1)}$ otherwise.
$E = \varnothing$
Compute $\sigma^{(1)}$ and its $\ominus$-decomposition $\ominus[B_1^{(1)}, \dots, B_{s_1}^{(1)}]$
for $j$ from 1 to $s_1^{(1)}$ do
    $V_j^{(1)} \leftarrow \varnothing$
    $S = pushallConfigs(B_j^{(1)})$
    if $S = \varnothing$ then return false
    else
        for $s \in S$ do
            $V_j^{(1)} \leftarrow V_j^{(1)} \bigcup \{(s, num)\}$
            $num \leftarrow num + 1$
        if $j > 1$ then $E = E \bigcup \{(s, s'), s \in V_j^{(1)}, s' \in V_{j-1}^{(1)}\}$
return $\mathcal{G}^{(1)} = (\bigcup_{j \in [1..s_1^{(1)}]} V_j^{(1)}, E)$
```

3.4 From step i to step $i+1$

After step i we know the graph $\mathcal{G}^{(i)}$ encoding every P_i-stack configuration of σ and we want to compute the graph $\mathcal{G}^{(i+1)}$ encoding P_{i+1}-stack configurations of σ at step $i+1$. From Lemma 19 it is enough to check the accessibility of pushall stack configuration of $\sigma^{(i+1)}$ from P_i-stack configurations of σ. We cannot check every pair of configurations (c, c') with c being a P_i-stack configuration and c' be a pushall stack configuration of $\sigma^{(i+1)}$, because the number of such pair of configurations is possibly exponential. Thus our algorithm focuses not on stack configurations of some $\sigma^{(\ell)}$ but on sets of stack configurations of blocks $B_j^{(\ell)}$, making use of Lemma 14.

Using Lemma 19, Lemma 14 can be rephrased as:

▶ **Lemma 20.** *Let c' be a total stack configuration of $\sigma^{(i+1)}$, $p = p^{(i)}$ and $q = q^{(i+1)}$. Then c' is a P_{i+1}-stack configuration of σ if and only if:*

- *For any $j \leq q$, $c'_{|B_j^{(i+1)}}$ is a pushall stack configuration of $\sigma_{|B_j^{(i+1)}}$, and*
- *there exists a P_i-stack configuration c of σ such that:*
  - *$c'_{|B_{\min(p,q)}^{(i)} \cup \cdots \cup B_q^{(i)}}$ is accessible from $c_{|B_{\min(p,q)}^{(i+1)} \cup \cdots \cup B_p^{(i+1)}}$ for $\sigma_{|B_p^{(i)} \bigcup B_q^{(i+1)}}$ and*
  - *$c'_{|B_1^{(i+1)} \cup \cdots \cup B_{\min(p,q)-1}^{(i+1)}} = c_{|B_1^{(i)} \cup \cdots \cup B_{\min(p,q)-1}^{(i)}}$*

Recall that a P_i-stack configuration of σ is encoded by a path in the sorting graph $\mathcal{G}^{(i)}$, corresponding to the $\ominus$-decomposition of the permutation $\sigma^{(i)}$ into blocks $B_j^{(i)}$. The last point of

Lemma 20 ensures that the first levels (1 to $min(p^{(i)}, q^{(i+1)}) - 1$) in $\mathcal{G}^{(i+1)}$ are the same as the ones in $\mathcal{G}^{(i)}$. The first point of Lemma 20 ensures that the last levels ($> q^{(i+1)}$) of $\mathcal{G}^{(i+1)}$ form a complete partitioned graph whose vertices are all pushall stack configurations of corresponding blocks. So the only unknown levels for $\mathcal{G}^{(i+1)}$ are those between $\min(p^{(i)}, q^{(i+1)})$ and $q^{(i+1)}$ and we can compute them by testing accessibility.

There are distinct cases depending on the relative values of $p^{(i)}$ and $q^{(i+1)}$. To lighten the notations in the following, we sometimes write p (resp. q) instead of $p^{(i)}$ (resp. $q^{(i+1)}$).

3.4.1 Case $p^{(i)} = q^{(i+1)}$

If $p^{(i)} = q^{(i+1)}$ then $B^{(i+1)}_{q^{(i+1)}} \cap A^{(i)} = B^{(i)}_{p^{(i)}} \cap A^{(i)}$ (see Figure 2).

We have the sorting graph $\mathcal{G}^{(i)}$ encoding all P_i-stack configurations of σ and we want to compute the sorting graph $\mathcal{G}^{(i+1)}$ encoding all P_{i+1}-stack configurations of σ assuming that $p^{(i)} = q^{(i+1)} = min(p^{(i)}, q^{(i+1)})$.

In this case, from Lemma 20, we only have to check accessibility of pushall configurations of $B^{(i+1)}_q$ from configurations of $B^{(i)}_p$ belonging to level p of $\mathcal{G}^{(i)}$. Indeed, from the definition of a sorting graph given p.7, for any vertex v of $\mathcal{G}^{(i)}$, there is a path between vertices of $V^{(i)}_1$ and $V^{(i)}_{s_i}$ going through v, and such a path corresponds to a P_i-stack configuration of σ. Thus for any configurations x of $B^{(i)}_p$ belonging to a vertex v of level p of $\mathcal{G}^{(i)}$, there is at least one P_i-stack configuration c of σ such that $c_{|B^{(i)}_p} = x$, and $c_{|B^{(i)}_1 \cup \dots \cup B^{(i)}_{\min(p,q)-1}}$ is encoded by a path from v to level p of $\mathcal{G}^{(i)}$ (which goes through each level $< p$).

If there is no pushall configuration of $B^{(i+1)}_q$ accessible from some configurations of $B^{(i)}_p$ belonging to level p of $\mathcal{G}^{(i)}$, or if $\sigma^{(i+1)}$ has no pushall configuration, then σ has no P_{i+1}-stack configuration and $\sigma_{\leq k_{i+1}}$ is not sortable (from Lemma 18).

This leads to algorithm 4.

```
Algorithm 4: $iteratepEqualsq(\mathcal{G}^{(i)})$
Data: $\sigma$ a permutation and $\mathcal{G}^{(i)}$ the sorting graph at step $i$
Result: false if $\sigma_{\leq k_{i+1}}$ is not sortable, the sorting graph $\mathcal{G}^{(i+1)}$ otherwise.
$\mathcal{G}$ an empty sorting graph with $s_{i+1}$ levels
$\mathcal{G}' \leftarrow ComputeG1(\sigma^{(i+1)})$ (pushall sorting graph of $\sigma^{(i+1)}$) or return false
Copy levels $q+1 \dots s_{i+1}$ of $\mathcal{G}'$ into the same levels of $\mathcal{G}$
for $(c, \ell)$ in level $p$ of $\mathcal{G}^{(i)}$ do
    $\mathcal{H}$ the subgraph of $\mathcal{G}^{(i)}$ induced by $(c, \ell)$ in levels $< p$
    for $(c', \ell')$ in level $q$ of $\mathcal{G}'$ do
        if $isAccessible(c, c', \sigma_{|B^{(i)}_p \bigcup B^{(i+1)}_q})$ then
            Add $(c', \ell')$ in level $q$ of $\mathcal{G}$ (if not already done)
            Merge $\mathcal{H}$ in levels $\leq q$ of $\mathcal{G}$ with $(c', \ell')$ as origin
if level $q$ of $\mathcal{G}$ is empty then return false
for $(c', \ell')$ in level $q$ of $\mathcal{G}$ do
    Add all edges from $(c', \ell')$ to each vertex of level $q+1$ of $\mathcal{G}$;
return $\mathcal{G}$
```

3.4.2 Case $p^{(i)} < q^{(i+1)}$

If $p^{(i)} < q^{(i+1)}$ then $B^{(i+1)}_{q^{(i+1)}} \cap A^{(i)} \subsetneq B^{(i)}_{p^{(i)}} \cap A^{(i)}$ (see Figure 2).

Again, Lemma 20 ensures that the first $p-1$ levels of $\mathcal{G}^{(i+1)}$ come from those of $\mathcal{G}^{(i)}$ and the levels $> q$ are all pushall stack configurations of the blocks $B^{(i+1)}_{>q}$ of $\sigma^{(i+1)}$. The difficult part is from level p to level q. As in the preceding case, by Lemma 20, we have to select among pushall stack configurations of blocks $p, p+1, \dots, q$ of $\sigma^{(i+1)}$ those accessible from a configuration of $B^{(i)}_p$ that appears at level p in $\mathcal{G}^{(i)}$. We can restrict the accessibility test from configurations of $B^{(i)}_p$

appearing in graph $\mathcal{G}^{(i)}$ to pushall stack configurations of $B_q^{(i+1)}$. Indeed, Lemma 15 ensures that the elements of blocks $B_j^{(i+1)}$ for j from p to $q-1$ are in the same stack at time t_i and at time t_{i+1}. Thus configurations of $B_j^{(i+1)}$ for j from p to $q-1$ are restrictions of configurations of $B_p^{(i)}$. We keep the same label in the vertex to encode that those configurations of $B_p^{(i+1)}, B_{p+1}^{(i+1)}, \dots, B_{q-1}^{(i+1)}$ come from the same configuration of $B_p^{(i)}$ and we build edges between vertices of $B_{j+1}^{(i+1)}$ and $B_j^{(i+1)}$ that come from the same configuration of $B_p^{(i)}$. It is because of this case $p = q$ that we have to label configurations in our sorting graph. Indeed, two different stack configurations c_1 and c_2 of $B_p^{(i)}$ may have the same restriction to some block $B_j^{(i+1)}$ but not be compatible with the same configurations, thus we want the corresponding vertices of level j of $\mathcal{G}^{(i+1)}$ to be distinct, that's why we use labels.

More precisely, we have algorithm 5.

Algorithm 5: *iteratepLessThanq*$(\mathcal{G}^{(i)})$

```
Data: $\sigma$ a permutation and $\mathcal{G}^{(i)}$ the sorting graph at step $i$
Result: false if $\sigma_{\leq k_{i+1}}$ is not sortable, the sorting graph $\mathcal{G}^{(i+1)}$ otherwise.
$\mathcal{G}$ an empty sorting graph with $s_{i+1}$ levels
$\mathcal{G}' \leftarrow ComputeG1(\sigma^{(i+1)})$ (pushall sorting graph of $\sigma^{(i+1)}$) or return false
Copy levels $q+1, \dots, s_{i+1}$ of $\mathcal{G}'$ into the same levels of $\mathcal{G}$
for $(c, \ell)$ in level $p$ of $\mathcal{G}^{(i)}$ do
    $\mathcal{H}$ the subgraph of $\mathcal{G}^{(i)}$ induced by $(c, \ell)$ in levels $< p$
    for $(c', \ell')$ in level $q$ of $\mathcal{G}'$ do
        if $isAccessible(c, c', \sigma_{|B_p^{(i)} \bigcup B_q^{(i+1)}})$ then
            Add $(c', \ell')$ in level $q$ of $\mathcal{G}$ (if not already done)
            for $j$ from $q-1$ downto $p$ do
                Add $(c_{|B_j^{(i+1)}}, \ell)$ in level $j$ of $\mathcal{G}$
                Add an edge between $(c_{|B_j^{(i+1)}}, \ell)$ and $(c_{|B_{j+1}^{(i+1)}}, \ell)$ in $\mathcal{G}$.
            Merge $\mathcal{H}$ in levels $\leq p$ of $\mathcal{G}$ with $(c_{|B_p^{(i+1)}}, \ell)$ as origin
if level $q$ of $\mathcal{G}$ is empty then return false
for $(c', \ell')$ in level $q$ of $\mathcal{G}$ do Add all edges from $(c', \ell')$ to each vertex of level $q+1$ of $\mathcal{G}$
return $\mathcal{G}$
```

Note that in Algorithm 5, before calling $isAccessible(c, c', \sigma_{|B_p^{(i)} \bigcup B_q^{(i+1)}})$, we extend configuration c' to $D^{(i)} \bigcup B_q^{(i+1)}$ by assigning the same stack than in c to points of $D^{(i)} \setminus B_q^{(i+1)}$. This is justified by Lemma 15.

3.4.3 Case $p^{(i)} > q^{(i+1)}$

If $p^{(i)} > q^{(i+1)}$ then $B_{p^{(i)}}^{(i)} \cap A^{(i)} \subsetneq B_{q^{(i+1)}}^{(i+1)} \cap A^{(i)}$ (see Figure 2).

This case is very similar to the preceding one except that $B_p^{(i)}$ is not cut into pieces but glued with preceding blocks. As a consequence, when testing accessibility of a configuration of $B_q^{(i+1)}$, we should consider every corresponding configuration in $\mathcal{G}^{(i)}$, that is, every configuration obtained by stacking configurations at level $q, q+1, \dots, p$ in $\mathcal{G}^{(i)}$. Unfortunately, this may give an exponential number of configurations; but noticing that by Lemma 15 the elements of blocks $B_q^{(i)}, B_{q+1}^{(i)} \dots B_{p-1}^{(i)}$ are exactly in the same stack at time t_i and at time t_{i+1}, it is sufficient to check the accessibility of a pushall configuration c' of $B_q^{(i+1)}$ from a configuration c of $B_p^{(i)}$ and verify afterwards whether the configuration c has ancestors in $\mathcal{G}^{(i)}$ that match exactly the configuration c'. This leads to algorithm 6.

Note that in Algorithm 6, before calling $isAccessible(c, c', \sigma_{|B_p^{(i)} \bigcup B_q^{(i+1)}})$, we extend configuration c to $D^{(i)} \bigcup B_p^{(i)}$ by assigning the same stack than in c' to points of $D^{(i)} \setminus B_p^{(i)}$. This is justified by Lemma 15.

Now that we have described all steps of our algorithm, we turn to the study of its complexity.

Algorithm 6: *iteratepGreaterThanq*$(\mathcal{G}^{(i)})$

```
Data: $\sigma$ a permutation and $\mathcal{G}^{(i)}$ the sorting graph at step $i$
Result: false if $\sigma_{\leq k_{i+1}}$ is not sortable, the sorting graph $\mathcal{G}^{(i+1)}$ otherwise
$\mathcal{G}$ an empty sorting graph with $s_{i+1}$ levels
$\mathcal{G}' \leftarrow ComputeG1(\sigma^{(i+1)})$ (pushall sorting graph of $\sigma^{(i+1)}$) or return false
Copy levels $q+1,\dots,s_{i+1}$ of $\mathcal{G}'$ into the same levels of $\mathcal{G}$
for $(c, \ell)$ in level $p$ of $\mathcal{G}^{(i)}$ do
    for $(c', \ell')$ in level $q$ of $\mathcal{G}'$ do
        if $isAccessible(c, c', \sigma_{|B_p^{(i)} \bigcup B_q^{(i+1)}})$ then
            if there is a path $(c, \ell) \leftrightarrow (c'_{|B_{p-1}^{(i)}}, \ell_1) \leftrightarrow \dots \leftrightarrow (c'_{|B_q^{(i)}}, \ell_k)$ in $\mathcal{G}^{(i)}$ then
                Add $(c', \ell')$ in level $q$ of $\mathcal{G}$ (if not already done)
                $\mathcal{H}$ the subgraph of $\mathcal{G}^{(i)}$ induced by $(c'_{|B_q^{(i)}}, \ell_k)$ in levels $< q$
                Merge $\mathcal{H}$ in levels $\leq q$ of $\mathcal{G}$ with $(c', \ell')$ as origin
if level $q$ of $\mathcal{G}$ is empty then return false
for $(c', \ell')$ in level $q$ of $\mathcal{G}$ do Add all edges from $(c', \ell')$ to each vertex of level $q+1$ of $\mathcal{G}$
return $\mathcal{G}$
```

4 Complexity Analysis

In this section we state the complexity of *isSortable(σ)*, our main algorithm (Algorithm 2).

▶ **Theorem 21.** *Given a permutation σ, Algorithm 2* isSortable(σ) *decides whether σ is sortable with two stacks in series in polynomial time w.r.t. $|\sigma|$.*

The key idea to prove this theorem relies on bounding the size of each graph $\mathcal{G}^{(i)}$:

▶ **Lemma 22.** *For any $i \in [1..r]$, the maximal number of vertices in a level of $\mathcal{G}^{(i)}$ is $9n$ where n is the size of the input permutation.*

▶ **Lemma 23.** *For any $i \in [1..r]$, the number of vertices of $\mathcal{G}^{(i)}$ is $\mathcal{O}(n^2)$ and the number of edges of $\mathcal{G}^{(i)}$ is $\mathcal{O}(n^3)$, where n is the size of the input permutation.*

References

1 Michael Albert, Mike D. Atkinson, and Steve Linton. Permutations generated by stacks and deques. *Annals of Combinatorics*, 14:3–16, 2010.

2 Mike D. Atkinson, M. M. Murphy, and N. Ruskuc. Sorting with two ordered stacks in series. *Theor. Comput. Sci.*, 289:205–223, October 2002.

3 Miklós Bóna. A survey of stack-sorting disciplines. *Electr. J. Comb.*, 9(2), 2002.

4 S. Kitaev. *Patterns in Permutations and Words*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2011.

5 Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.

6 Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

7 Maximillian M. Murphy. *Restricted permutations, anti chains, atomic classes and stack sorting*. PhD thesis, University of St Andrews, 2002.

8 A. Pierrot. *Combinatoire et algorithmique dans les classes de permutations*. PhD thesis, Université Paris Diderot - Paris 7, 2013. (in English).

9 A. Pierrot and D. Rossin. 2-stack pushall sortable permutations, 2013. arxiv:1303.4376.

10 Robert Endre Tarjan. Sorting using networks of queues and stacks. *J. ACM*, 19(2):341–346, 1972.

11 Julian West. Sorting twice through a stack. *Theor. Comput. Sci.*, 117(1&2):303–313, 1993.