

On two-stack sorting

Adeline Pierrot Dominique Rossin

LIAFA, Université Paris Diderot - Paris 7

Journées combinatoires de Bordeaux 2013

- 1 Introduction to stack sorting
- 2 Pushall sorting
- 3 General sorting

Representations of permutations

Permutation: Bijective map from $[1..n]$ to itself

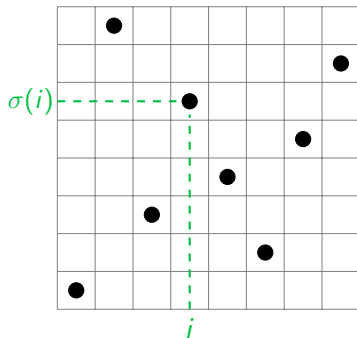
- Two-line representation:

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 8 & 3 & 6 & 4 & 2 & 5 & 7 \end{pmatrix}$$

- One-line representation:

$$\sigma = 1 \ 8 \ 3 \ 6 \ 4 \ 2 \ 5 \ 7$$

- Graphical representation:



Patterns in permutations

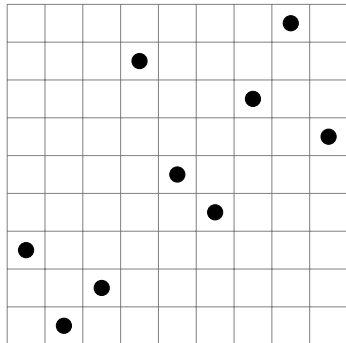
Pattern relation \preceq :

$\pi \in S_k$ is a pattern of $\sigma \in S_n$ if
 $\exists 1 \leq i_1 < \dots < i_k \leq n$ such that
 $\sigma_{i_1} \dots \sigma_{i_k}$ is order-isomorphic to π .
We write $\pi \preceq \sigma$.

Equivalently: Normalizing $\sigma_{i_1} \dots \sigma_{i_k}$
on $[1..k]$ yields π .

Example: $1324 \preceq 312854796$
since $2549 \equiv 1324$.

Remark: $\pi \not\preceq \sigma = \sigma$ avoids π .



Patterns in permutations

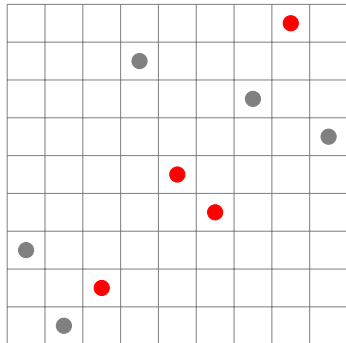
Pattern relation \preceq :

$\pi \in S_k$ is a pattern of $\sigma \in S_n$ if
 $\exists 1 \leq i_1 < \dots < i_k \leq n$ such that
 $\sigma_{i_1} \dots \sigma_{i_k}$ is order-isomorphic to π .
We write $\pi \preceq \sigma$.

Equivalently: Normalizing $\sigma_{i_1} \dots \sigma_{i_k}$
on $[1..k]$ yields π .

Example: $1324 \preceq 312854796$
since $2549 \equiv 1324$.

Remark: $\pi \not\preceq \sigma = \sigma$ avoids π .



Class of permutations: set downward closed for \preceq

Equivalently: $\sigma \in \mathcal{C}$ and $\pi \preceq \sigma \Rightarrow \pi \in \mathcal{C}$

Permutation Classes

Class of permutations: set downward closed for \preceq

Equivalently: $\sigma \in \mathcal{C}$ and $\pi \preceq \sigma \Rightarrow \pi \in \mathcal{C}$

$Av(B)$: the class of perm. avoiding all the patterns in the set B .

Permutation Classes

Class of permutations: set downward closed for \preceq

Equivalently: $\sigma \in \mathcal{C}$ and $\pi \preceq \sigma \Rightarrow \pi \in \mathcal{C}$

$Av(B)$: the class of perm. avoiding all the patterns in the set B .

Prop.: Every class \mathcal{C} is characterized by its **basis**:

$$\mathcal{C} = Av(B) \text{ for } B = \{\sigma \notin \mathcal{C} \mid \forall \pi \preceq \sigma \text{ with } \pi \neq \sigma, \pi \in \mathcal{C}\}$$

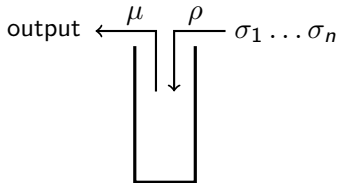
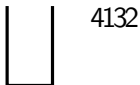
Basis may be finite or infinite.

Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

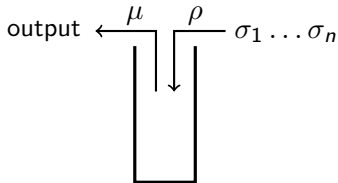
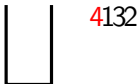


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

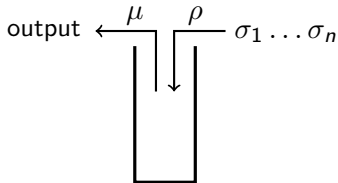
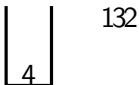


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

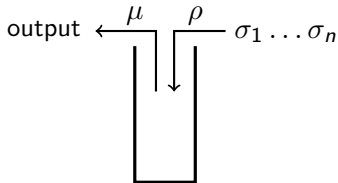
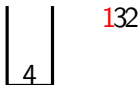


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

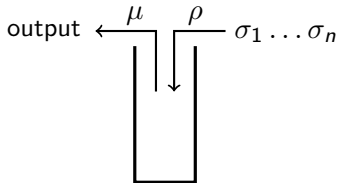
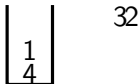


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

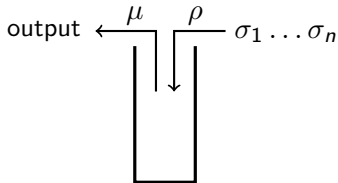
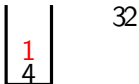


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

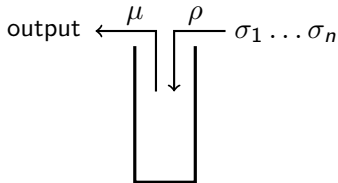


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

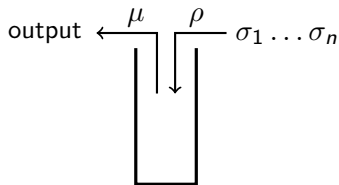
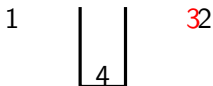


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

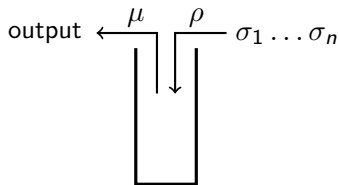
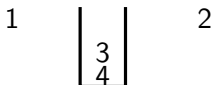


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

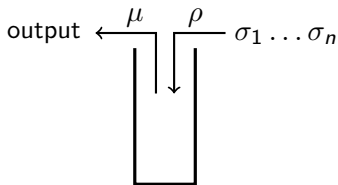
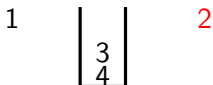


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:



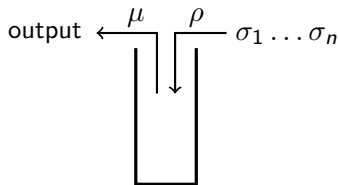
Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

1 $\begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix}$



Stack sorting

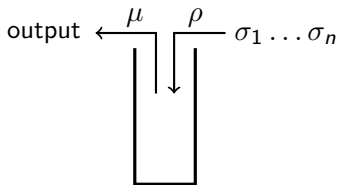
Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

1

2
3
4

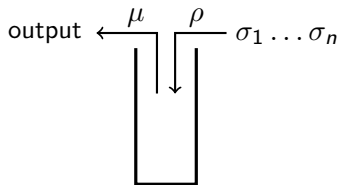
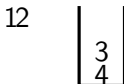


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

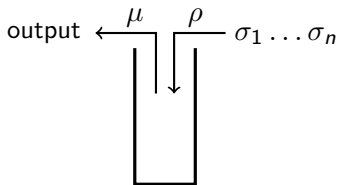
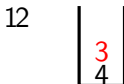


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

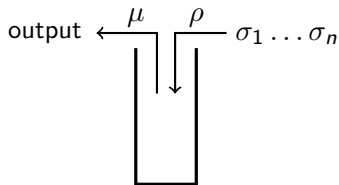
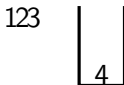


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

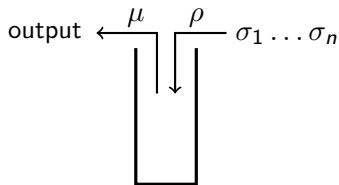


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

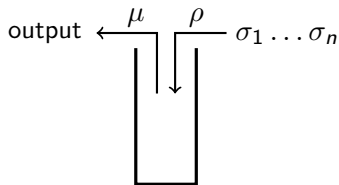
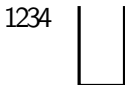


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

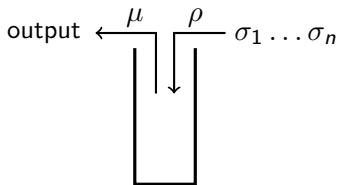
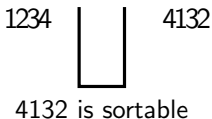


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

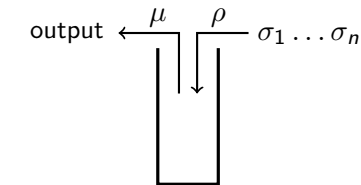
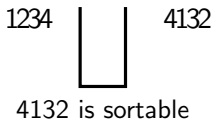


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:




Stack sorting

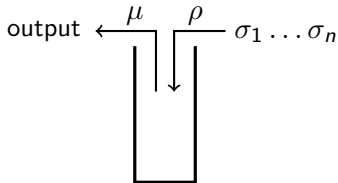
Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

1234  4132
4132 is sortable

 2413

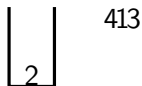
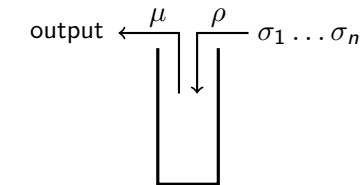
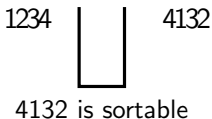


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

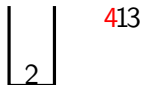
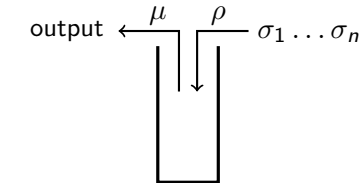
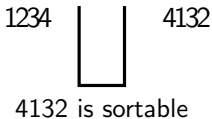


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

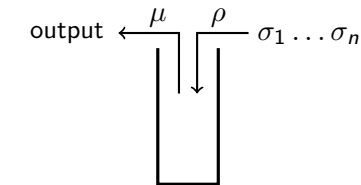
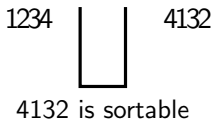


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

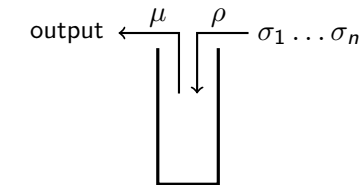
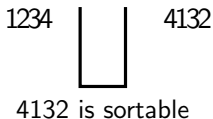


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

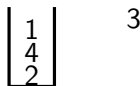
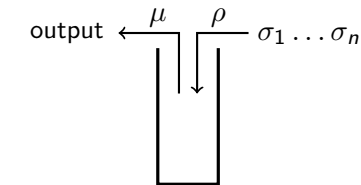
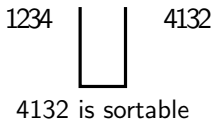


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

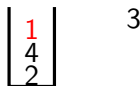
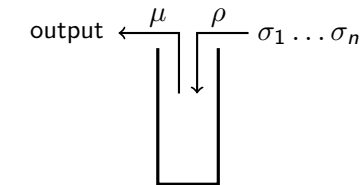
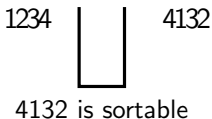


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

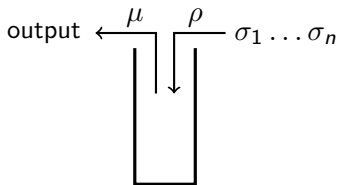
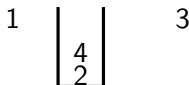
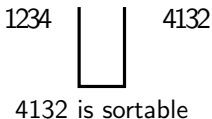


Stack sorting

Stack: last-in first-out device introduced by Knuth.

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

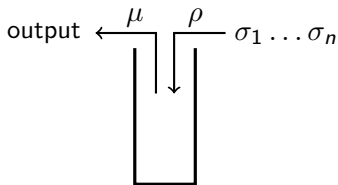
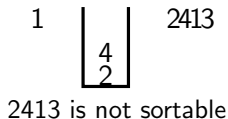
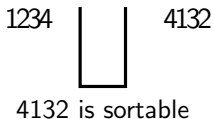


Stack sorting

Stack: last-in first-out device introduced by Knuth.

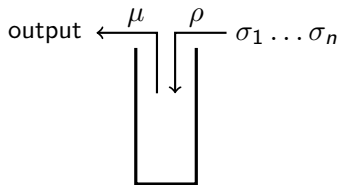
Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:



Sorting with one stack: a linear algorithm

Question: How to decide if σ is sortable?

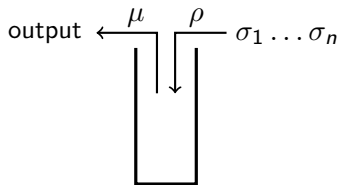


Sorting with one stack: a linear algorithm

Question: How to decide if σ is sortable?

Find $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity

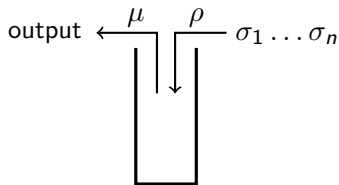
$|\sigma| = n \Rightarrow |m(\sigma)|_\rho = |m(\sigma)|_\mu = n.$



Sorting with one stack: a linear algorithm

Question: How to decide if σ is sortable?

Find $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity
 $|\sigma| = n \Rightarrow |m(\sigma)|_\rho = |m(\sigma)|_\mu = n.$

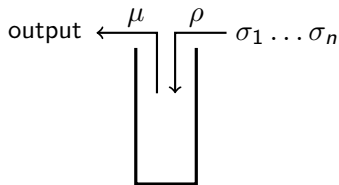


Naive algorithm: Check if $m(\sigma)$ is the identity $\forall m \in \{\rho, \mu\}^{2n}$
 \rightarrow exponential algorithm.

Sorting with one stack: a linear algorithm

Question: How to decide if σ is sortable?

Find $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity
 $|\sigma| = n \Rightarrow |m(\sigma)|_\rho = |m(\sigma)|_\mu = n.$



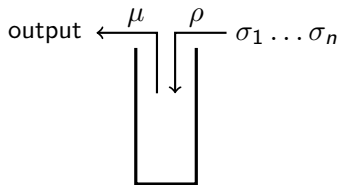
Naive algorithm: Check if $m(\sigma)$ is the identity $\forall m \in \{\rho, \mu\}^{2n}$
 \rightarrow exponential algorithm.

Key: At most **one** way to sort a permutation:
Do move μ if and only if the top of the stack is the next element to be output.

Sorting with one stack: a linear algorithm

Question: How to decide if σ is sortable?

Find $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity
 $|\sigma| = n \Rightarrow |m(\sigma)|_\rho = |m(\sigma)|_\mu = n.$



Naive algorithm: Check if $m(\sigma)$ is the identity $\forall m \in \{\rho, \mu\}^{2n}$
→ exponential algorithm.

Key: At most **one** way to sort a permutation:
Do move μ if and only if the top of the stack is the next element to be output.

→ A **linear** algorithm to test whether a permutation is sortable.

Sorting with one stack: a mathematical characterization

Question: How many sortable permutations of size n ?

Sorting with one stack: a mathematical characterization

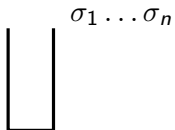
Question: How many sortable permutations of size n ?

σ sortable $\Leftrightarrow \sigma$ avoids 231

Sorting with one stack: a mathematical characterization

Question: How many sortable permutations of size n ?

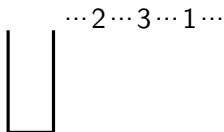
σ sortable $\Leftrightarrow \sigma$ avoids 231



Sorting with one stack: a mathematical characterization

Question: How many sortable permutations of size n ?

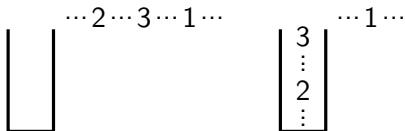
σ sortable $\Leftrightarrow \sigma$ avoids 231



Sorting with one stack: a mathematical characterization

Question: How many sortable permutations of size n ?

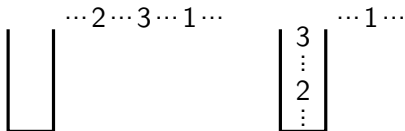
σ sortable $\Leftrightarrow \sigma$ avoids 231



Sorting with one stack: a mathematical characterization

Question: How many sortable permutations of size n ?

σ sortable $\Leftrightarrow \sigma$ avoids 231



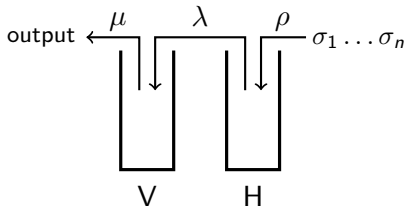
The set of permutations sortable with one stack :

$Av(231)$, enumerated by Catalan numbers: $c_n = \frac{1}{n+1} \binom{2n}{n}$

Sorting with two stacks in serie

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \lambda, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Question: σ a given permutation, is σ sortable with two stacks?



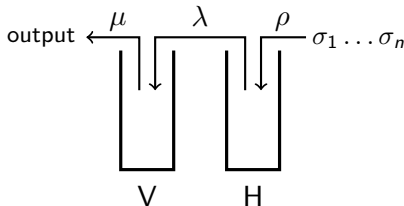
Sorting with two stacks in serie

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \lambda, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Question: σ a given permutation, is σ sortable with two stacks?

Naive algorithm: Check if $m(\sigma)$ is the identity $\forall m \in \{\rho, \lambda, \mu\}^{3n}$
 $|m(\sigma)|_\rho = |m(\sigma)|_\lambda = |m(\sigma)|_\mu = n$

→ exponential algorithm.



Sorting with two stacks in serie

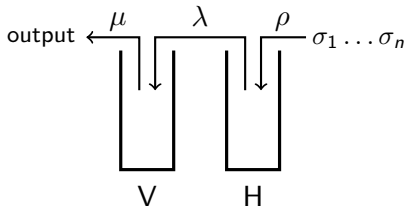
Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \lambda, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Question: σ a given permutation, is σ sortable with two stacks?

Naive algorithm: Check if $m(\sigma)$ is the identity $\forall m \in \{\rho, \lambda, \mu\}^{3n}$
 $|m(\sigma)|_\rho = |m(\sigma)|_\lambda = |m(\sigma)|_\mu = n$

→ exponential algorithm.

Is there a **polynomial** algorithm?



Sorting with two stacks in serie

Definition: σ is sortable if \exists a sequence of moves $m \in \{\rho, \lambda, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

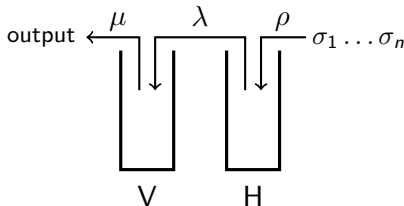
Question: σ a given permutation, is σ sortable with two stacks?

Naive algorithm: Check if $m(\sigma)$ is the identity $\forall m \in \{\rho, \lambda, \mu\}^{3n}$
 $|m(\sigma)|_\rho = |m(\sigma)|_\lambda = |m(\sigma)|_\mu = n$

→ exponential algorithm.

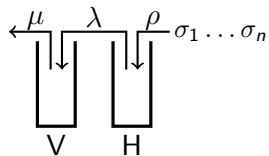
Is there a **polynomial** algorithm?

Try to limit the number of sortings to test.



A canonical way to sort?

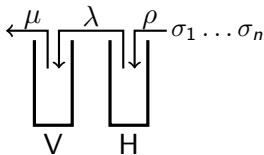
No longer a **unique** way to sort.



A canonical way to sort?

No longer a **unique** way to sort.

As example moves μ and ρ commute.

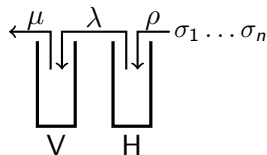


A canonical way to sort?

No longer a **unique** way to sort.

As example moves μ and ρ commute.

Find a **canonical** sorting?



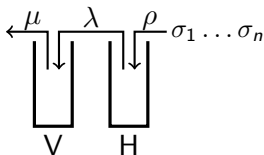
A canonical way to sort?

No longer a **unique** way to sort.

As example moves μ and ρ commute.

Find a **canonical** sorting?

Choose move $\mu \Leftrightarrow$ top of V is the next element to be output.



A canonical way to sort?

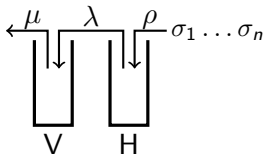
No longer a **unique** way to sort.

As example moves μ and ρ commute.

Find a **canonical** sorting?

Choose move $\mu \Leftrightarrow$ top of V is the next element to be output.

Some permutations still have an **exponential number of sortings**:
 $n(n-1) \dots 1$ can be sorted in $2^{(n-1)}$ different ways.



A canonical way to sort?

No longer a **unique** way to sort.

As example moves μ and ρ commute.

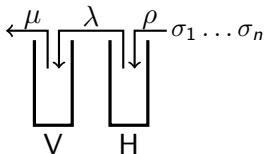
Find a **canonical** sorting?

Choose move $\mu \Leftrightarrow$ top of V is the next element to be output.

Some permutations still have an **exponential number of sortings**:
 $n(n-1)\dots 1$ can be sorted in $2^{(n-1)}$ different ways.

Note: During all the sorting procedure:

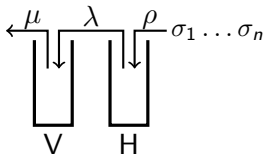
- Stack H is in **increasing** order of **indices** from bottom to top
- If the output is the identity, stack V is in **decreasing** order of **values** from bottom to top



A canonical way to sort?

No longer a **unique** way to sort.

As example moves μ and ρ commute.



Find a **canonical** sorting?

Choose move $\mu \Leftrightarrow$ top of V is the next element to be output.

Some permutations still have an **exponential number of sortings**:
 $n(n-1) \dots 1$ can be sorted in $2^{(n-1)}$ different ways.

Note: During all the sorting procedure:

- Stack H is in **increasing** order of **indices** from bottom to top
- If the output is the identity, stack V is in **decreasing** order of **values** from bottom to top

But **no way to choose** between move λ and move ρ .

A permutation class

Let $\pi \prec \sigma$ (pattern)

sorting procedure for $\sigma \rightarrow$ sorting procedure for π

A permutation class

Let $\pi \prec \sigma$ (pattern)

sorting procedure for $\sigma \rightarrow$ sorting procedure for π

$\hookrightarrow \sigma$ sortable and $\pi \prec \sigma \Rightarrow \pi$ sortable

\hookrightarrow sortable permutations form a **class** $Av(B)$

A permutation class

Let $\pi \prec \sigma$ (pattern)

sorting procedure for $\sigma \rightarrow$ sorting procedure for π

$\hookrightarrow \sigma$ sortable and $\pi \prec \sigma \Rightarrow \pi$ sortable

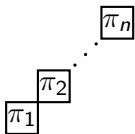
\hookrightarrow sortable permutations form a class $Av(B)$

But B infinite and not characterised

length	sortable	unsortable	basis
$n \leq 6$	$n!$	0	0
7	5018	22	22
8	39374	946	51
9	336870	26010	146
10	3066695	562105	604

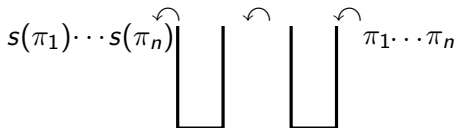
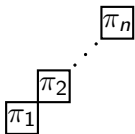
Decomposition

- $\sigma = \oplus[\pi_1, \dots, \pi_n]$



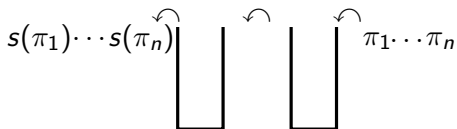
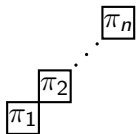
Decomposition

- $\sigma = \oplus[\pi_1, \dots, \pi_n]$ is sortable $\Leftrightarrow \forall i, \pi_i$ is sortable.

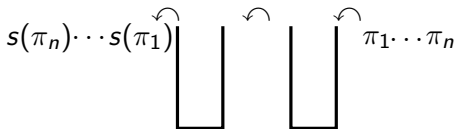
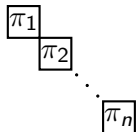


Decomposition

- $\sigma = \oplus[\pi_1, \dots, \pi_n]$ is sortable $\Leftrightarrow \forall i, \pi_i$ is sortable.

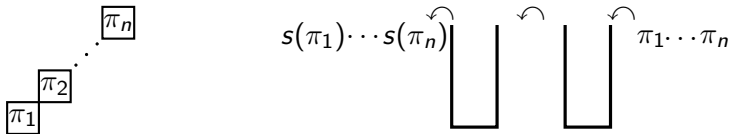


- $\sigma = \ominus[\pi_1, \dots, \pi_n]$ is sortable $\Rightarrow \forall i, \pi_i$ is sortable.

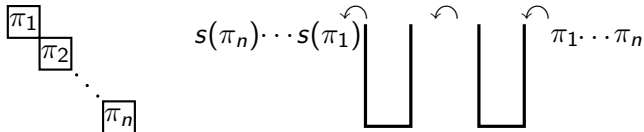


Decomposition

- $\sigma = \oplus[\pi_1, \dots, \pi_n]$ is sortable $\Leftrightarrow \forall i, \pi_i$ is sortable.



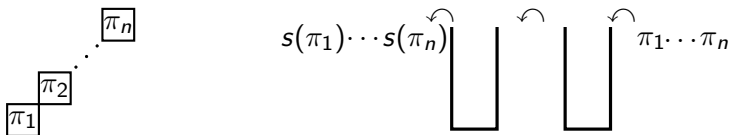
- $\sigma = \ominus[\pi_1, \dots, \pi_n]$ is sortable $\Rightarrow \forall i, \pi_i$ is sortable.



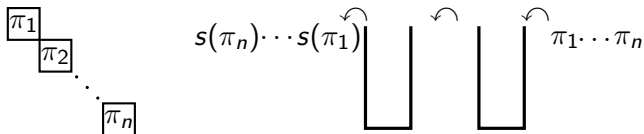
Converse not true: π_i has to admit a special sorting in 2 steps.

Decomposition

- $\sigma = \oplus[\pi_1, \dots, \pi_n]$ is sortable $\Leftrightarrow \forall i, \pi_i$ is sortable.



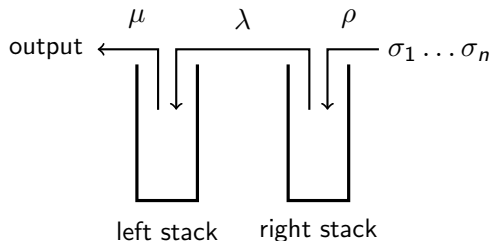
- $\sigma = \ominus[\pi_1, \dots, \pi_n]$ is sortable $\Rightarrow \forall i, \pi_i$ is sortable.



Converse not true: π_i has to admit a special sorting in 2 steps.

$\sigma = \ominus[\pi_1, \dots, \pi_n]$ is sortable $\Leftrightarrow \forall i < n, \pi_i$ is pushall sortable and π_n is sortable.

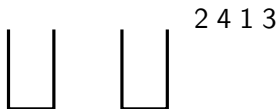
Pushall sorting



Definition: σ is pushall sortable if $\exists m_1 \in \{\rho, \lambda\}^*$ and $m_2 \in \{\lambda, \mu\}^*$ s.t. the output $m_2(m_1(\sigma))$ is the identity.

→ A sorting in 2 parts.

Example and property



Example and property



4 1 3

Example and property



1 3

Example and property



1 3

Example and property



3

Example and property



Example and property



Example and property



Example and property

1



Example and property

1



Example and property

1 2



Example and property

1 2 3



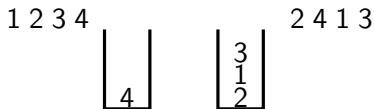
Example and property



Example and property

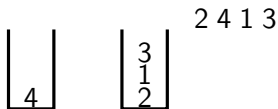


Example and property



Stack configuration c just after the last element enter the right stack gives all information about the pushall sorting procedure:

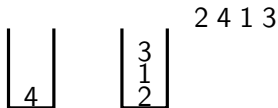
Example and property



Stack configuration c just after the last element enter the right stack gives all information about the pushall sorting procedure:

- First part of the procedure: do only moves ρ and λ to reach configuration c

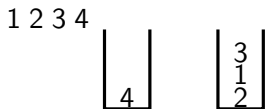
Example and property



Stack configuration c just after the last element enter the right stack gives all information about the pushall sorting procedure:

- First part of the procedure: do only moves ρ and λ to reach configuration $c =$ sorting with one stack (the right one) considering the left stack as the output.

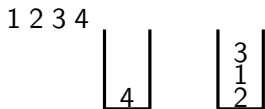
Example and property



Stack configuration c just after the last element enter the right stack gives all information about the pushall sorting procedure:

- First part of the procedure: do only moves ρ and λ to reach configuration $c =$ sorting with one stack (the right one) considering the left stack as the output.
- Second part of the procedure: do only moves λ and μ to pop out in increasing order configuration c

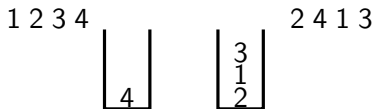
Example and property



Stack configuration c just after the last element enter the right stack gives all information about the pushall sorting procedure:

- First part of the procedure: do only moves ρ and λ to reach configuration $c =$ sorting with one stack (the right one) considering the left stack as the output.
- Second part of the procedure: do only moves λ and μ to pop out in increasing order configuration $c =$ sorting with one stack (the left one) considering the right stack as the input.

Example and property



Stack configuration c just after the last element enter the right stack gives all information about the pushall sorting procedure:

- First part of the procedure: do only moves ρ and λ to reach configuration $c =$ sorting with one stack (the right one) considering the left stack as the output.
- Second part of the procedure: do only moves λ and μ to pop out in increasing order configuration $c =$ sorting with one stack (the left one) considering the right stack as the input.

→ Test in **linear time** whether a stack configuration is valid

A first naive algorithm

Bijection between

- pushall sorting procedures of σ and
- valid stack configurations of σ

A first naive algorithm

Bijection between

- pushall sorting procedures of σ and
- valid stack configurations of σ

Test in linear time whether a stack configuration is valid for σ

A first naive algorithm

Bijection between

- pushall sorting procedures of σ and
- valid stack configurations of σ

Test in linear time whether a stack configuration is valid for σ

→ Algorithm deciding whether a permutation is pushall sortable

A first naive algorithm

Bijection between

- pushall sorting procedures of σ and
- valid stack configurations of σ

Test in linear time whether a stack configuration is valid for σ

→ Algorithm deciding whether a permutation is pushall sortable

★ **Naive algorithm:**

Try all stack configurations of σ and test whether it is valid

→ exponential algorithm: 2^n bicolorings (\sim stack config.) to test

A first naive algorithm

Bijection between

- pushall sorting procedures of σ and
- valid stack configurations of σ

Test in linear time whether a stack configuration is valid for σ

→ Algorithm deciding whether a permutation is pushall sortable

★ **Naive algorithm:**

Try all stack configurations of σ and test whether it is valid

→ exponential algorithm: 2^n bicolorings (\sim stack config.) to test

★ Limit the number of bicolorings to test by studying properties of valid configurations (and bicolorings).

Stack-patterns

Unsortable stack-patterns: $\begin{array}{|c|} \hline \\ \hline \end{array}$ $\begin{array}{|c|} \hline 2 \\ \hline 3 \\ \hline 1 \\ \hline \end{array}$, $\begin{array}{|c|} \hline 2 \\ \hline 1 \\ \hline \end{array}$ and $\begin{array}{|c|} \hline \\ \hline 2 \\ \hline \end{array}$ $\begin{array}{|c|} \hline \\ \hline 3 \\ \hline 1 \\ \hline \end{array}$

Theorem: a stack configuration can be popped out in increasing order \Leftrightarrow it avoids the 3 unsortable stack-patterns.

Stack-patterns

Unsortable stack-patterns: $\begin{array}{|c|} \hline \\ \hline \end{array}$, $\begin{array}{|c|} \hline 2 \\ \hline 3 \\ \hline 1 \\ \hline \end{array}$, $\begin{array}{|c|} \hline 2 \\ \hline 1 \\ \hline \end{array}$ and $\begin{array}{|c|} \hline 2 \\ \hline \end{array}$ $\begin{array}{|c|} \hline 3 \\ \hline 1 \\ \hline \end{array}$

Theorem: a stack configuration can be popped out in increasing order \Leftrightarrow it avoids the 3 unsortable stack-patterns.

- Necessary to avoid these patterns

Stack-patterns

Unsortable stack-patterns: $\begin{array}{|c|} \hline \\ \hline \end{array}$, $\begin{array}{|c|} \hline 2 \\ \hline 3 \\ \hline 1 \\ \hline \end{array}$, $\begin{array}{|c|} \hline 2 \\ \hline 1 \\ \hline \end{array}$ and $\begin{array}{|c|} \hline 2 \\ \hline \end{array}$ $\begin{array}{|c|} \hline 3 \\ \hline 1 \\ \hline \end{array}$

Theorem: a stack configuration can be popped out in increasing order \Leftrightarrow it avoids the 3 unsortable stack-patterns.

- Necessary to avoid these patterns
- Sufficient: you can put the smallest element out and keep avoiding stack-patterns

Sorting as a coloring

Right coloring: coloring of σ with two colors **G** and **R** s.t.

- no pattern **132** in **R**
- no pattern **213** in **G**
- no point of **R** lying vertically between a pattern **12** of **G**
- no point of **G** lying horizontally between a pattern **12** of **R**

\Rightarrow coloring with forbidden patterns **132**, **213**, **1X2** and **2/13**

Sorting as a coloring

Right coloring: coloring of σ with two colors **G** and **R** s.t.

- no pattern **132** in **R**
- no pattern **213** in **G**
- no point of **R** lying vertically between a pattern **12** of **G**
- no point of **G** lying horizontally between a pattern **12** of **R**

\Rightarrow coloring with forbidden patterns **132**, **213**, **1X2** and **2/13**

Theorem: σ is pushall sortable $\Leftrightarrow \sigma$ has a right coloring.

proof: **R** = right stack and **G** = left stack \Rightarrow bijection between right colorings and valid stack configurations.

Recall: Unsortable stack-patterns: $\left[\begin{array}{|c|} \hline \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 2 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 3 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 1 \\ \hline \end{array} \right]$, $\left[\begin{array}{|c|} \hline 2 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 1 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline \\ \hline \end{array} \right]$ and $\left[\begin{array}{|c|} \hline 2 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 3 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 1 \\ \hline \end{array} \right]$
 Forbidden colored patterns: 132, 213, 1X2 and 2/13.

- If σ is sortable, put color **R** to the elements of the right stack and color **G** to those of the left stack.



- If σ has a right coloring, put the elements of **R** in the right stack and thoses of **G** in the left stack.

Recall: Unsortable stack-patterns: $\left[\right] \left[\begin{array}{c} 2 \\ 3 \\ 1 \end{array} \right]$, $\left[\begin{array}{c} 2 \\ 1 \end{array} \right] \left[\right]$ and $\left[2 \right] \left[\begin{array}{c} 3 \\ 1 \end{array} \right]$
Forbidden colored patterns: **132**, **213**, **1X2** and **2/13**.

- If σ is sortable, put color **R** to the elements of the right stack and color **G** to those of the left stack.



- If σ has a right coloring, put the elements of **R** in the right stack and thoses of **G** in the left stack.

Recall: Unsortable stack-patterns: $\left[\begin{array}{|c|} \hline \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 2 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 3 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 1 \\ \hline \end{array} \right]$, $\left[\begin{array}{|c|} \hline 2 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 1 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline \\ \hline \end{array} \right]$ and $\left[\begin{array}{|c|} \hline 2 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 3 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 1 \\ \hline \end{array} \right]$
 Forbidden colored patterns: **132**, **213**, **1X2** and **2/13**.

- If σ is sortable, put color **R** to the elements of the right stack and color **G** to those of the left stack.



- If σ has a right coloring, put the elements of **R** in the right stack and thoses of **G** in the left stack.

Recall: Unsortable stack-patterns: $\left[\begin{array}{|c|} \hline \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 2 \\ \hline 3 \\ \hline 1 \\ \hline \end{array} \right], \left[\begin{array}{|c|} \hline 2 \\ \hline 1 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline \\ \hline \end{array} \right]$ and $\left[\begin{array}{|c|} \hline 2 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 3 \\ \hline 1 \\ \hline \end{array} \right]$
Forbidden colored patterns: **132**, **213**, **1X2** and **2/13**.

- If σ is sortable, put color **R** to the elements of the right stack and color **G** to those of the left stack.



- If σ has a right coloring, put the elements of **R** in the right stack and thoses of **G** in the left stack.

Recall: Unsortable stack-patterns: $\left[\begin{array}{|c|} \hline \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 2 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 3 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 1 \\ \hline \end{array} \right]$, $\left[\begin{array}{|c|} \hline 2 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 1 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline \\ \hline \end{array} \right]$ and $\left[\begin{array}{|c|} \hline 2 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 3 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 1 \\ \hline \end{array} \right]$
Forbidden colored patterns: **132**, **213**, **1X2** and **2/13**.

- If σ is sortable, put color **R** to the elements of the right stack and color **G** to those of the left stack.



- If σ has a right coloring, put the elements of **R** in the right stack and thoses of **G** in the left stack.

Recall: Unsortable stack-patterns: $\left[\begin{array}{|c|} \hline \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 2 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 3 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 1 \\ \hline \end{array} \right]$, $\left[\begin{array}{|c|} \hline 2 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 1 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline \\ \hline \end{array} \right]$ and $\left[\begin{array}{|c|} \hline 2 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 3 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 1 \\ \hline \end{array} \right]$
 Forbidden colored patterns: 132, 213, 1X2 and 2/13.

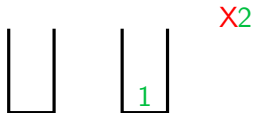
- If σ is sortable, put color **R** to the elements of the right stack and color **G** to those of the left stack.



- If σ has a right coloring, put the elements of **R** in the right stack and thoses of **G** in the left stack.

Recall: Unsortable stack-patterns: $\left[\begin{array}{|c|} \hline \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 2 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 3 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 1 \\ \hline \end{array} \right]$, $\left[\begin{array}{|c|} \hline 2 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 1 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline \\ \hline \end{array} \right]$ and $\left[\begin{array}{|c|} \hline 2 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 3 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 1 \\ \hline \end{array} \right]$
 Forbidden colored patterns: 132 , 213 , $1X2$ and $2/13$.

- If σ is sortable, put color **R** to the elements of the right stack and color **G** to those of the left stack.



- If σ has a right coloring, put the elements of **R** in the right stack and thoses of **G** in the left stack.

Recall: Unsortable stack-patterns: $\left[\begin{array}{|c|} \hline \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 2 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 3 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 1 \\ \hline \end{array} \right]$, $\left[\begin{array}{|c|} \hline 2 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 1 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline \\ \hline \end{array} \right]$ and $\left[\begin{array}{|c|} \hline 2 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 3 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 1 \\ \hline \end{array} \right]$
 Forbidden colored patterns: 132, 213, 1X2 and 2/13.

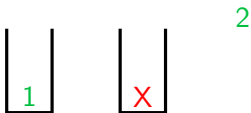
- If σ is sortable, put color **R** to the elements of the right stack and color **G** to those of the left stack.



- If σ has a right coloring, put the elements of **R** in the right stack and thoses of **G** in the left stack.

Recall: Unsortable stack-patterns: $\left[\begin{array}{|c|} \hline \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 2 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 3 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 1 \\ \hline \end{array} \right]$, $\left[\begin{array}{|c|} \hline 2 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 1 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline \\ \hline \end{array} \right]$ and $\left[\begin{array}{|c|} \hline 2 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 3 \\ \hline \end{array} \right] \left[\begin{array}{|c|} \hline 1 \\ \hline \end{array} \right]$
 Forbidden colored patterns: 132, 213, 1X2 and 2/13.

- If σ is sortable, put color **R** to the elements of the right stack and color **G** to those of the left stack.



- If σ has a right coloring, put the elements of **R** in the right stack and thoses of **G** in the left stack.

Recall: Unsortable stack-patterns: $\left[\right] \left[\begin{array}{c} 2 \\ 3 \\ 1 \end{array} \right]$, $\left[\begin{array}{c} 2 \\ 1 \end{array} \right] \left[\right]$ and $\left[2 \right] \left[\begin{array}{c} 3 \\ 1 \end{array} \right]$
Forbidden colored patterns: **132**, **213**, **1X2** and **2/13**.

- If σ is sortable, put color **R** to the elements of the right stack and color **G** to those of the left stack.



- If σ has a right coloring, put the elements of **R** in the right stack and thoses of **G** in the left stack.

Recall: Unsortable stack-patterns: $\begin{array}{|c|} \hline \\ \hline \end{array} \begin{array}{|c|} \hline 2 \\ \hline \end{array} \begin{array}{|c|} \hline 3 \\ \hline \end{array} \begin{array}{|c|} \hline 1 \\ \hline \end{array}$, $\begin{array}{|c|} \hline 2 \\ \hline \end{array} \begin{array}{|c|} \hline 1 \\ \hline \end{array} \begin{array}{|c|} \hline \\ \hline \end{array}$ and $\begin{array}{|c|} \hline 2 \\ \hline \end{array} \begin{array}{|c|} \hline 3 \\ \hline \end{array} \begin{array}{|c|} \hline 1 \\ \hline \end{array}$
 Forbidden colored patterns: **132**, **213**, **1X2** and **2/13**.

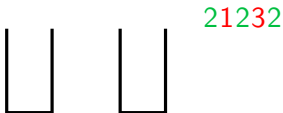
- If σ is sortable, put color **R** to the elements of the right stack and color **G** to those of the left stack.



- If σ has a right coloring, put the elements of **R** in the right stack and thoses of **G** in the left stack.

Recall: Unsortable stack-patterns: $\begin{array}{|c|} \hline \\ \hline \end{array} \begin{array}{|c|} \hline 2 \\ \hline \end{array} \begin{array}{|c|} \hline 3 \\ \hline \end{array} \begin{array}{|c|} \hline 1 \\ \hline \end{array}$, $\begin{array}{|c|} \hline 2 \\ \hline \end{array} \begin{array}{|c|} \hline 1 \\ \hline \end{array} \begin{array}{|c|} \hline \\ \hline \end{array}$ and $\begin{array}{|c|} \hline 2 \\ \hline \end{array} \begin{array}{|c|} \hline 3 \\ \hline \end{array} \begin{array}{|c|} \hline 1 \\ \hline \end{array}$
 Forbidden colored patterns: 132, 213, 1X2 and 2/13.

- If σ is sortable, put color **R** to the elements of the right stack and color **G** to those of the left stack.



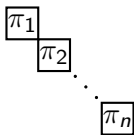
- If σ has a right coloring, put the elements of **R** in the right stack and thoses of **G** in the left stack.

Recall: Unsortable stack-patterns: $\begin{array}{|c|} \hline \\ \hline \end{array} \begin{array}{|c|} \hline 2 \\ \hline \end{array} \begin{array}{|c|} \hline 3 \\ \hline \end{array} \begin{array}{|c|} \hline 1 \\ \hline \end{array}$, $\begin{array}{|c|} \hline 2 \\ \hline \end{array} \begin{array}{|c|} \hline 1 \\ \hline \end{array} \begin{array}{|c|} \hline \\ \hline \end{array}$ and $\begin{array}{|c|} \hline 2 \\ \hline \end{array} \begin{array}{|c|} \hline 3 \\ \hline \end{array} \begin{array}{|c|} \hline 1 \\ \hline \end{array}$
Forbidden colored patterns: 132, 213, 1X2 and 2/13.

- If σ is sortable, put color **R** to the elements of the right stack and color **G** to those of the left stack.



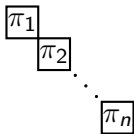
- If σ has a right coloring, put the elements of **R** in the right stack and thoses of **G** in the left stack.



$$\sigma = \ominus[\pi_1, \dots, \pi_k] \Rightarrow \text{Col}(\sigma) \approx \text{Col}(\pi_1) \times \dots \times \text{Col}(\pi_k)$$

$\text{Col}(\sigma)$ = the set of **right colorings** of σ

\ominus -decomposable permutations

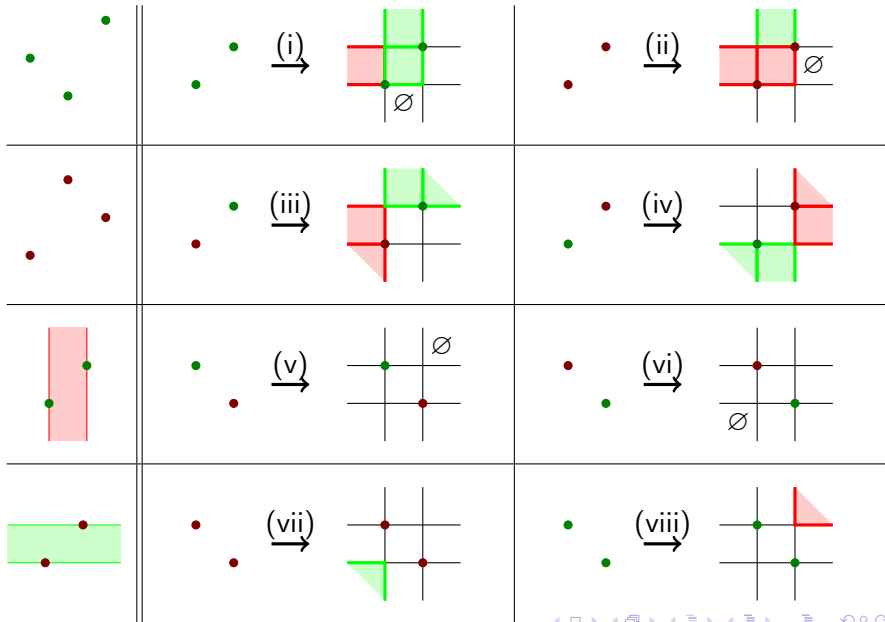


$$\sigma = \ominus[\pi_1, \dots, \pi_k] \Rightarrow Col(\sigma) \approx Col(\pi_1) \times \dots \times Col(\pi_k)$$

$Col(\sigma)$ = the set of **right colorings** of σ



Forbidden colored patterns \Rightarrow implication rules



Restrict the number of bicolorings to test

Add hypothesis

to ensure a polynomial number of bicolorings to test.

Restrict the number of bicolourings to test

Add hypothesis

to ensure a polynomial number of bicolourings to test.

- Assume σ is \ominus -indecomposable.

Otherwise $\sigma = \ominus[\pi_1 \dots \pi_n]$ with π_i \ominus -indecomposable

$$\sigma = \ominus[\pi_1, \dots, \pi_k] \Rightarrow \text{Col}(\sigma) \approx \text{Col}(\pi_1) \times \dots \times \text{Col}(\pi_k)$$

So replace σ by the π_i .

Restrict the number of bicolourings to test

Add hypothesis

to ensure a polynomial number of bicolourings to test.

- Assume σ is \ominus -indecomposable.
Otherwise $\sigma = \ominus[\pi_1 \dots \pi_n]$ with π_i \ominus -indecomposable
 $\sigma = \ominus[\pi_1, \dots, \pi_k] \Rightarrow Col(\sigma) \approx Col(\pi_1) \times \dots \times Col(\pi_k)$
So replace σ by the π_i .
- Separate distinct cases:
Each pattern 12 is unicolor
There are patterns 12 but no pattern 12
There are patterns 12 but no pattern 12
There are patterns 12 and patterns 12.

First case: Each pattern 12 is unicolor

Proposition: σ \ominus -indecomposable and C a right coloring of σ where each pattern 12 is unicolor $\Rightarrow C$ is unicolor.

First case: Each pattern 12 is unicolor

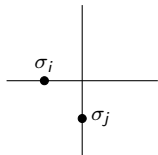
Proposition: σ \ominus -indecomposable and C a right coloring of σ where each pattern 12 is unicolor $\Rightarrow C$ is unicolor.

Proof: Left-to-right minima of σ are unicolor.

First case: Each pattern 12 is unicolor

Proposition: σ \ominus -indecomposable and C a right coloring of σ where each pattern 12 is unicolor $\Rightarrow C$ is unicolor.

Proof: Left-to-right minima of σ are unicolor.

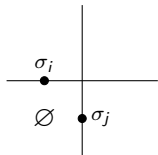


Let σ_i and σ_j consecutive left-to-right minima of σ .

First case: Each pattern 12 is unicolor

Proposition: σ \ominus -indecomposable and C a right coloring of σ where each pattern 12 is unicolor $\Rightarrow C$ is unicolor.

Proof: Left-to-right minima of σ are unicolor.

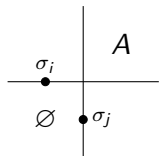


Let σ_i and σ_j consecutive left-to-right minima of σ .

First case: Each pattern 12 is unicolor

Proposition: σ \ominus -indecomposable and C a right coloring of σ where each pattern 12 is unicolor $\Rightarrow C$ is unicolor.

Proof: Left-to-right minima of σ are unicolor.



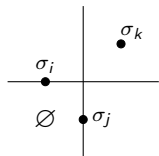
Let σ_i and σ_j consecutive left-to-right minima of σ .

Zone A is non-empty as σ is \ominus -indecomposable.

First case: Each pattern 12 is unicolor

Proposition: σ \ominus -indecomposable and C a right coloring of σ where each pattern 12 is unicolor $\Rightarrow C$ is unicolor.

Proof: Left-to-right minima of σ are unicolor.



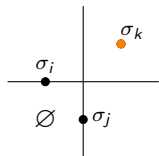
Let σ_i and σ_j consecutive left-to-right minima of σ .

Zone A is non-empty as σ is \ominus -indecomposable.

First case: Each pattern 12 is unicolor

Proposition: σ \ominus -indecomposable and C a right coloring of σ where each pattern 12 is unicolor $\Rightarrow C$ is unicolor.

Proof: Left-to-right minima of σ are unicolor.



Let σ_i and σ_j consecutive left-to-right minima of σ .

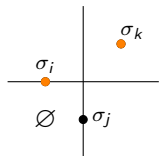
Zone A is non-empty as σ is \ominus -indecomposable.

Let σ_k in this zone and c its color,

First case: Each pattern 12 is unicolor

Proposition: σ \ominus -indecomposable and C a right coloring of σ where each pattern 12 is unicolor $\Rightarrow C$ is unicolor.

Proof: Left-to-right minima of σ are unicolor.



Let σ_i and σ_j consecutive left-to-right minima of σ .

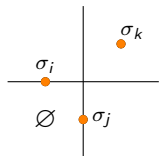
Zone A is non-empty as σ is \ominus -indecomposable.

Let σ_k in this zone and c its color,

First case: Each pattern 12 is unicolor

Proposition: σ \ominus -indecomposable and C a right coloring of σ where each pattern 12 is unicolor $\Rightarrow C$ is unicolor.

Proof: Left-to-right minima of σ are unicolor.



Let σ_i and σ_j consecutive left-to-right minima of σ .

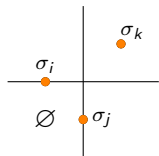
Zone A is non-empty as σ is \ominus -indecomposable.

Let σ_k in this zone and c its color, then σ_i and σ_j also have color c .

First case: Each pattern 12 is unicolor

Proposition: σ \ominus -indecomposable and C a right coloring of σ where each pattern 12 is unicolor $\Rightarrow C$ is unicolor.

Proof: Left-to-right minima of σ are unicolor.



Let σ_i and σ_j consecutive left-to-right minima of σ .

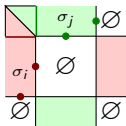
Zone A is non-empty as σ is \ominus -indecomposable.

Let σ_k in this zone and c its color, then σ_i and σ_j also have color c .

Consequence: We just have to check the 2 unicolor colorings (all points in **R** or all points in **G**).

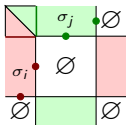
Other cases

- There are patterns 12 but no pattern 12:
Position of the up-leftmost pattern 12 determines all colors:

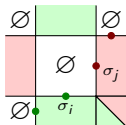


Other cases

- There are patterns 12 but no pattern 12: Position of the up-leftmost pattern 12 determines all colors:

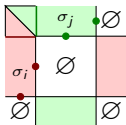


- There are patterns 12 but no pattern 12: Position of the down-rightmost pattern 12 determines all colors:

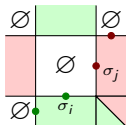


Other cases

- There are patterns 12 but no pattern 12: Position of the up-leftmost pattern 12 determines all colors:



- There are patterns 12 but no pattern 12: Position of the down-rightmost pattern 12 determines all colors:



- There are patterns 12 and patterns 12: Position of the up-leftmost pattern 12 and the down-rightmost pattern 12 determines all colors.

Polynomial algorithm

σ \ominus -indecomposable. To compute all its right colorings:

Polynomial algorithm

σ \ominus -indecomposable. To compute all its right colorings:

- Check if unicolor coloring \mathbf{R} is right

Polynomial algorithm

σ \ominus -indecomposable. To compute all its right colorings:

- Check if unicolor coloring **R** is right
- Check if unicolor coloring **G** is right

Polynomial algorithm

σ \ominus -indecomposable. To compute all its right colorings:

- Check if unicolor coloring **R** is right
- Check if unicolor coloring **G** is right
- Choose a pattern 12 as the down-rightmost pattern **12** and test if the coloring without pattern **12** is right

Polynomial algorithm

σ \ominus -indecomposable. To compute all its right colorings:

- Check if unicolor coloring **R** is right
- Check if unicolor coloring **G** is right
- Choose a pattern 12 as the down-rightmost pattern **12** and test if the coloring without pattern **12** is right
- Choose a pattern 12 as the up-leftmost pattern **12** and test if the coloring without pattern **12** is right

σ \ominus -indecomposable. To compute all its right colorings:

- Check if unicolor coloring **R** is right
- Check if unicolor coloring **G** is right
- Choose a pattern 12 as the down-rightmost pattern **12** and test if the coloring without pattern **12** is right
- Choose a pattern 12 as the up-leftmost pattern **12** and test if the coloring without pattern **12** is right
- Choose a pattern 12 as the down-rightmost pattern **12** and choose a pattern 12 as the up-leftmost pattern **12** and test if the coloring is right.

Polynomial algorithm

σ \ominus -indecomposable. To compute all its right colorings:

- Check if unicolor coloring **R** is right
- Check if unicolor coloring **G** is right
- Choose a pattern 12 as the down-rightmost pattern **12** and test if the coloring without pattern **12** is right
- Choose a pattern 12 as the up-leftmost pattern **12** and test if the coloring without pattern **12** is right
- Choose a pattern 12 as the down-rightmost pattern **12** and choose a pattern 12 as the up-leftmost pattern **12** and test if the coloring is right.

Test if coloring right = linear

Polynomial algorithm

σ \ominus -indecomposable. To compute all its right colorings:

- Check if unicolor coloring **R** is right
- Check if unicolor coloring **G** is right
- Choose a pattern 12 as the down-rightmost pattern **12** and test if the coloring without pattern **12** is right
- Choose a pattern 12 as the up-leftmost pattern **12** and test if the coloring without pattern **12** is right
- Choose a pattern 12 as the down-rightmost pattern **12** and choose a pattern 12 as the up-leftmost pattern **12** and test if the coloring is right.

Test if coloring right = linear

→ $\mathcal{O}(n^5)$ algorithm

Refinement

$Col(\sigma) =$ the set of **right colorings** of σ

Separate 9 kinds of colorings (instead of the 4 previous one) for σ
 \ominus -indecomposable

Refinement

$Col(\sigma) =$ the set of **right colorings** of σ

Separate 9 kinds of colorings (instead of the 4 previous one) for σ
 \ominus -indecomposable

Then the position of **one** particular point determines all the coloring

$Col(\sigma) =$ the set of **right colorings** of σ

Separate 9 kinds of colorings (instead of the 4 previous one) for σ
 \ominus -indecomposable

Then the position of **one** particular point determines all the coloring

- σ \ominus -indecomposable $\Rightarrow |Col(\sigma)| \leq 9|\sigma|$ computed in $\mathcal{O}(|\sigma|^2)$

$Col(\sigma)$ = the set of **right colorings** of σ

Separate 9 kinds of colorings (instead of the 4 previous one) for σ \ominus -indecomposable

Then the position of **one** particular point determines all the coloring

- σ \ominus -indecomposable $\Rightarrow |Col(\sigma)| \leq 9|\sigma|$ computed in $\mathcal{O}(|\sigma|^2)$
- $\sigma = \ominus[\pi_1, \dots, \pi_k] \Rightarrow Col(\sigma) \approx Col(\pi_1) \times \dots \times Col(\pi_k)$

$Col(\sigma)$ = the set of **right colorings** of σ

Separate 9 kinds of colorings (instead of the 4 previous one) for σ
 \ominus -indecomposable

Then the position of **one** particular point determines all the coloring

- σ \ominus -indecomposable $\Rightarrow |Col(\sigma)| \leq 9|\sigma|$ computed in $\mathcal{O}(|\sigma|^2)$
- $\sigma = \ominus[\pi_1, \dots, \pi_k] \Rightarrow Col(\sigma) \approx Col(\pi_1) \times \dots \times Col(\pi_k)$

\rightarrow Description of $Col(\sigma)$ in **linear space**: $9|\pi_1| + \dots + 9|\pi_k| = 9|\sigma|$

$Col(\sigma)$ = the set of **right colorings** of σ

Separate 9 kinds of colorings (instead of the 4 previous one) for σ \ominus -indecomposable

Then the position of **one** particular point determines all the coloring

- σ \ominus -indecomposable $\Rightarrow |Col(\sigma)| \leq 9|\sigma|$ computed in $\mathcal{O}(|\sigma|^2)$
- $\sigma = \ominus[\pi_1, \dots, \pi_k] \Rightarrow Col(\sigma) \approx Col(\pi_1) \times \dots \times Col(\pi_k)$

\rightarrow Description of $Col(\sigma)$ in **linear space**: $9|\pi_1| + \dots + 9|\pi_k| = 9|\sigma|$

\rightarrow Algorithm giving $Col(\sigma)$ in **quadratic time**.

From pushall sorting to general sorting

Algorithm to decide if σ is pushall sortable in quadratic time

Use it to decide if σ is sortable (general sorting)

From pushall sorting to general sorting

Algorithm to decide if σ is pushall sortable in quadratic time

Use it to decide if σ is sortable (general sorting)

- $\sigma = \oplus[\pi_1, \dots, \pi_n]$ is sortable $\Leftrightarrow \forall i, \pi_i$ is sortable.
- $\sigma = \ominus[\pi_1, \dots, \pi_n]$ is sortable $\Leftrightarrow \forall i < n, \pi_i$ is pushall sortable and π_n is sortable.

From pushall sorting to general sorting

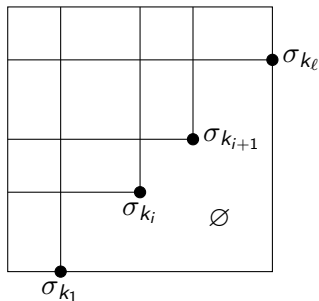
Algorithm to decide if σ is pushall sortable in quadratic time

Use it to decide if σ is sortable (general sorting)

- $\sigma = \oplus[\pi_1, \dots, \pi_n]$ is sortable $\Leftrightarrow \forall i, \pi_i$ is sortable.
- $\sigma = \ominus[\pi_1, \dots, \pi_n]$ is sortable $\Leftrightarrow \forall i < n, \pi_i$ is pushall sortable and π_n is sortable.
- What if σ is not decomposable?

Right-to-left minima

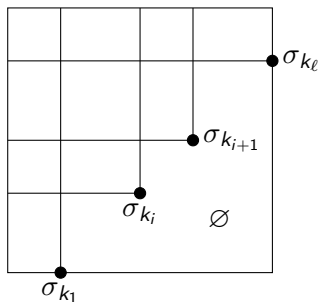
We decompose $\sigma = \dots \sigma_{k_1} \dots \sigma_{k_2} \dots \sigma_{k_\ell}$ where σ_{k_i} are **right-to-left minima** of σ ($\sigma_{k_1} = 1$ and $\sigma_{k_\ell} = \sigma_n$)



Right-to-left minima

We decompose $\sigma = \dots \sigma_{k_1} \dots \sigma_{k_2} \dots \sigma_{k_\ell}$ where σ_{k_i} are **right-to-left minima** of σ ($\sigma_{k_1} = 1$ and $\sigma_{k_\ell} = \sigma_n$)

Key idea: look at the **stack configuration** when σ_{k_i} enters stack H



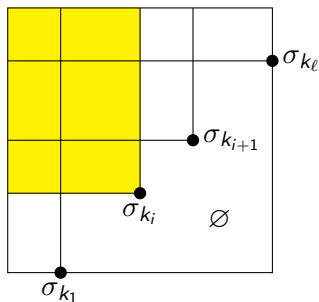
Right-to-left minima

We decompose $\sigma = \dots \sigma_{k_1} \dots \sigma_{k_2} \dots \sigma_{k_\ell}$ where σ_{k_i} are **right-to-left minima** of σ ($\sigma_{k_1} = 1$ and $\sigma_{k_\ell} = \sigma_n$)

Key idea: look at the **stack configuration** when σ_{k_i} enters stack H

$$\sigma^{(i)} = \{\sigma_j \mid j < k_i \text{ et } \sigma_j > \sigma_{k_i}\}$$

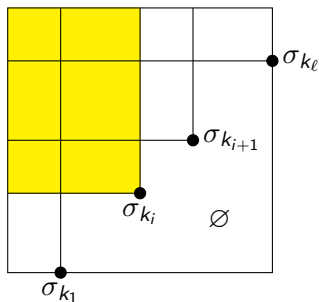
σ sortable $\Rightarrow \sigma^{(i)}$ pushall sortable $\forall i$



Right-to-left minima

We decompose $\sigma = \dots \sigma_{k_1} \dots \sigma_{k_2} \dots \sigma_{k_\ell}$ where σ_{k_i} are **right-to-left minima** of σ ($\sigma_{k_1} = 1$ and $\sigma_{k_\ell} = \sigma_n$)

Key idea: look at the **stack configuration** when σ_{k_i} enters stack H



$$\sigma^{(i)} = \{\sigma_j \mid j < k_i \text{ et } \sigma_j > \sigma_{k_i}\}$$

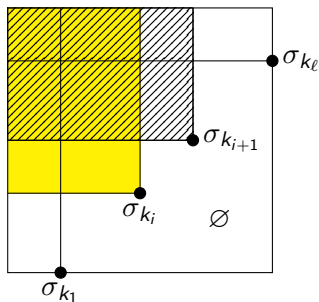
σ sortable $\Rightarrow \sigma^{(i)}$ pushall sortable $\forall i$

converse not true : pushall sortings
have to be **compatibles**

Right-to-left minima

We decompose $\sigma = \dots \sigma_{k_1} \dots \sigma_{k_2} \dots \sigma_{k_\ell}$ where σ_{k_i} are **right-to-left minima** of σ ($\sigma_{k_1} = 1$ and $\sigma_{k_\ell} = \sigma_n$)

Key idea: look at the **stack configuration** when σ_{k_i} enters stack H



$$\sigma^{(i)} = \{\sigma_j \mid j < k_i \text{ et } \sigma_j > \sigma_{k_i}\}$$

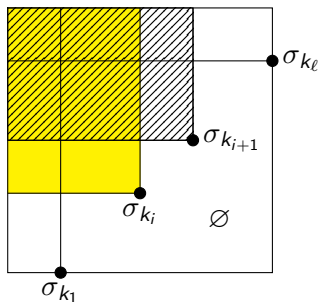
σ sortable $\Rightarrow \sigma^{(i)}$ pushall sortable $\forall i$

converse not true : pushall sortings
have to be **compatibles**

Right-to-left minima

We decompose $\sigma = \dots \sigma_{k_1} \dots \sigma_{k_2} \dots \sigma_{k_\ell}$ where σ_{k_i} are **right-to-left minima** of σ ($\sigma_{k_1} = 1$ and $\sigma_{k_\ell} = \sigma_n$)

Key idea: look at the **stack configuration** when σ_{k_i} enters stack H



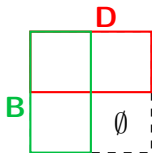
$$\sigma^{(i)} = \{\sigma_j \mid j < k_i \text{ et } \sigma_j > \sigma_{k_i}\}$$

σ sortable $\Rightarrow \sigma^{(i)}$ pushall sortable $\forall i$

converse not true : pushall sortings
have to be **compatibles**

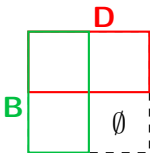
check compatibility between valid
stack configurations

Compatible stack configurations



c stack configuration of B and c' stack configuration of D .

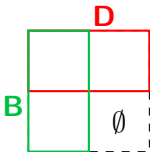
Compatible stack configurations



c stack configuration of B and c' stack configuration of D .

Check compatibility in linear time (no choice to go from c to c'):

Compatible stack configurations

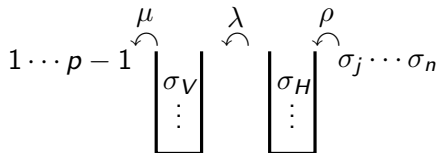
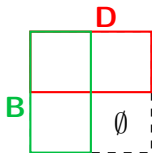


c stack configuration of B and c' stack configuration of D .

Check compatibility in linear time (no choice to go from c to c'):

Start with configuration c

Compatible stack configurations

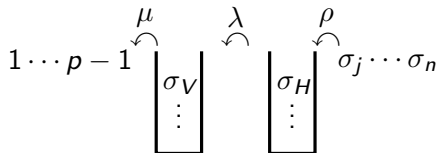
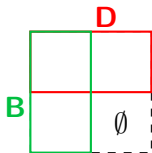


c stack configuration of B and c' stack configuration of D .

Check compatibility in linear time (no choice to go from c to c'):

Start with configuration c

Compatible stack configurations



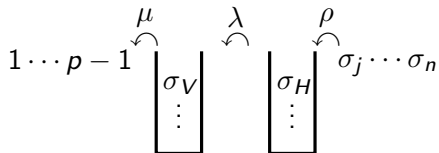
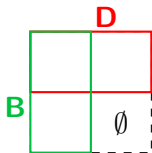
c stack configuration of B and c' stack configuration of D .

Check compatibility in linear time (no choice to go from c to c'):

Start with configuration c

While $j \leq n$ (with $n = \max\{i \mid \sigma_i \in D\}$):

Compatible stack configurations



c stack configuration of B and c' stack configuration of D .

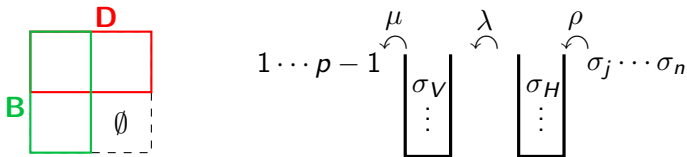
Check compatibility in linear time (no choice to go from c to c'):

Start with configuration c

While $j \leq n$ (with $n = \max\{i \mid \sigma_i \in D\}$):

 If $\sigma_V = p$ then do μ and $p \leftarrow p + 1$.

Compatible stack configurations



c stack configuration of B and c' stack configuration of D .

Check compatibility in linear time (no choice to go from c to c'):

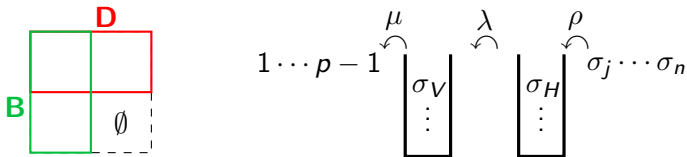
Start with configuration c

While $j \leq n$ (with $n = \max\{i \mid \sigma_i \in D\}$):

 If $\sigma_V = p$ then do μ and $p \leftarrow p + 1$.

 Else if $H = \emptyset$ or $\sigma_H \in H(c')$ do ρ and $j \leftarrow j + 1$.

Compatible stack configurations



c stack configuration of B and c' stack configuration of D .

Check compatibility in linear time (no choice to go from c to c'):

Start with configuration c

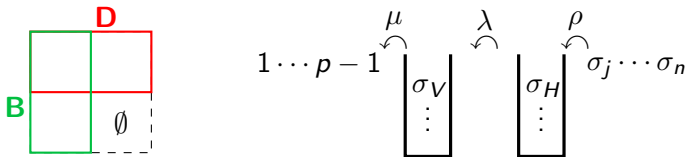
While $j \leq n$ (with $n = \max\{i \mid \sigma_i \in D\}$):

If $\sigma_V = p$ then do μ and $p \leftarrow p + 1$.

Else if $H = \emptyset$ or $\sigma_H \in H(c')$ do ρ and $j \leftarrow j + 1$.

Else if $\sigma_j \in H(c')$ or $\sigma_H > \sigma_j$ then do λ .

Compatible stack configurations



c stack configuration of B and c' stack configuration of D .

Check compatibility in linear time (no choice to go from c to c'):

Start with configuration c

While $j \leq n$ (with $n = \max\{i \mid \sigma_i \in D\}$):

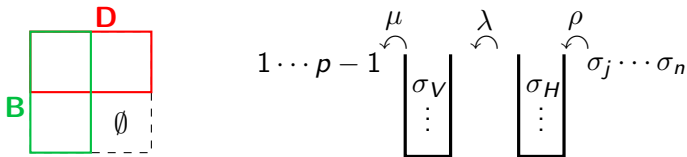
If $\sigma_V = p$ then do μ and $p \leftarrow p + 1$.

Else if $H = \emptyset$ or $\sigma_H \in H(c')$ do ρ and $j \leftarrow j + 1$.

Else if $\sigma_j \in H(c')$ or $\sigma_H > \sigma_j$ then do λ .

Else do ρ and $j \leftarrow j + 1$.

Compatible stack configurations



c stack configuration of B and c' stack configuration of D .

Check compatibility in linear time (no choice to go from c to c'):

Start with configuration c

While $j \leq n$ (with $n = \max\{i \mid \sigma_i \in D\}$):

 If $\sigma_V = p$ then do μ and $p \leftarrow p + 1$.

 Else if $H = \emptyset$ or $\sigma_H \in H(c')$ do ρ and $j \leftarrow j + 1$.

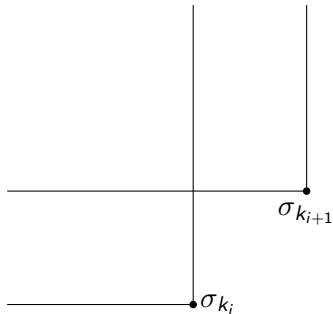
 Else if $\sigma_j \in H(c')$ or $\sigma_H > \sigma_j$ then do λ .

 Else do ρ and $j \leftarrow j + 1$.

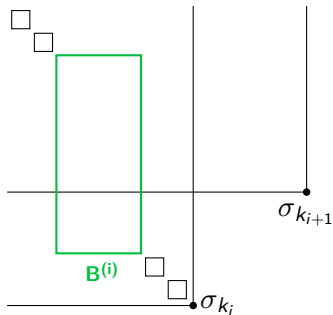
Test whether the configuration obtained is c'

Restrict number of tests

$\sigma^{(i)}$ and $\sigma^{(i+1)}$ may have an exponential number of valid stack configurations

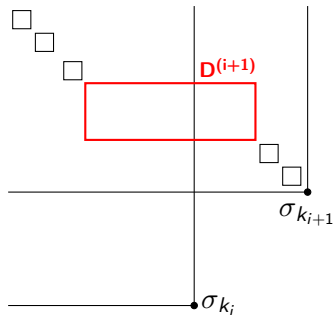


Restrict number of tests



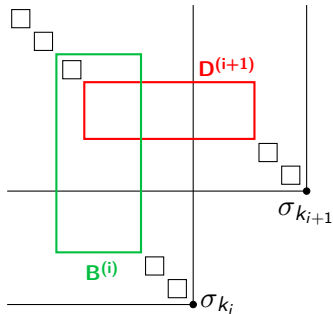
$\sigma^{(i)}$ and $\sigma^{(i+1)}$ may have an exponential number of valid stack configurations

Restrict number of tests



$\sigma^{(i)}$ and $\sigma^{(i+1)}$ may have an exponential number of valid stack configurations

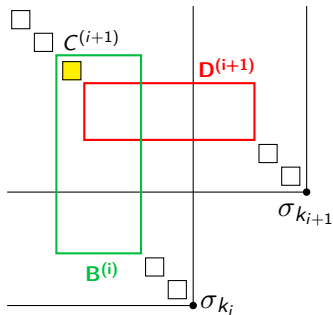
Restrict number of tests



$\sigma^{(i)}$ and $\sigma^{(i+1)}$ may have an **exponential** number of valid stack configurations

It is sufficient to test compatibility on $B^{(i)}$ and $D^{(i+1)}$

Restrict number of tests

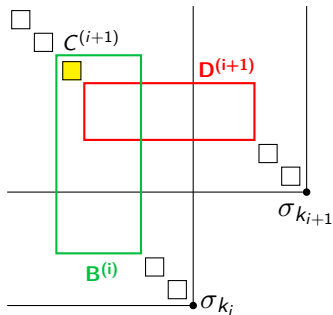


$\sigma^{(i)}$ and $\sigma^{(i+1)}$ may have an **exponential** number of valid stack configurations

It is sufficient to test compatibility on $B^{(i)}$ and $D^{(i+1)}$

Then stack configurations of $C^{(i+1)}$ are **linked** with those of $D^{(i+1)}$

Restrict number of tests



$\sigma^{(i)}$ and $\sigma^{(i+1)}$ may have an **exponential** number of valid stack configurations

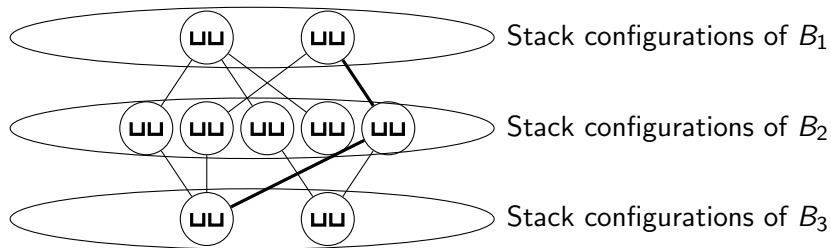
It is sufficient to test compatibility on $B^{(i)}$ and $D^{(i+1)}$

Then stack configurations of $C^{(i+1)}$ are **linked** with those of $D^{(i+1)}$

→ sorting graph

Sorting graph for $\sigma^{(i)}$

$$\sigma^{(i)} = \ominus[B_1, B_2, \dots, B_s]$$



Links between compatibles stack configurations

→ a **path** gives a **valid** stack configuration of $\sigma^{(i)}$ which is a part of a **sorting procedure** of $\sigma_1 \dots \sigma_{k_i}$.

$\sigma = \dots \sigma_{k_1} \dots \sigma_{k_2} \dots \sigma_{k_\ell}$ (σ_{k_i} = right-to-left minima of σ)

At step i , the algorithm returns false if $\sigma_1 \dots \sigma_{k_i}$ is not 2-stack sortable.

Otherwise it computes the sorting graph of $\sigma^{(i)}$ describing all the possible stack configurations when σ_{k_i} enters the stacks in a sorting procedure of σ verifying some conditions.

Sorting graph of $\sigma^{(i)}$ computed from the one of $\sigma^{(i-1)}$ by checking compatibility between configurations.

Complexity ($n = |\sigma|$)

For each right-to-left minima (at most n times):

Complexity ($n = |\sigma|$)

For each right-to-left minima (at most n times):

- Compute the \ominus -decomposition of $\sigma^{(i)} = \ominus[B_1^{(i)}, B_2^{(i)}, \dots, B_s^{(i)}]$
→ linear

Complexity ($n = |\sigma|$)

For each right-to-left minima (at most n times):

- Compute the \ominus -decomposition of $\sigma^{(i)} = \ominus[B_1^{(i)}, B_2^{(i)}, \dots, B_s^{(i)}]$
→ linear
- For new $B_j^{(i)}$, compute valid stack configurations of $B_j^{(i)}$ → $n \times n^2$

Complexity ($n = |\sigma|$)

For each right-to-left minima (at most n times):

- Compute the \ominus -decomposition of $\sigma^{(i)} = \ominus[B_1^{(i)}, B_2^{(i)}, \dots, B_s^{(i)}]$
→ linear
- For new $B_j^{(i)}$, compute valid stack configurations of $B_j^{(i)}$ → $n \times n^2$
- Find indices p and q for which we have to check the compatibility of stack configurations of $B_p^{(i)}$ and $B_q^{(i-1)}$ → linear

Complexity ($n = |\sigma|$)

For each right-to-left minima (at most n times):

- Compute the \ominus -decomposition of $\sigma^{(i)} = \ominus[B_1^{(i)}, B_2^{(i)}, \dots, B_s^{(i)}]$
→ linear
- For new $B_j^{(i)}$, compute valid stack configurations of $B_j^{(i)}$ → $n \times n^2$
- Find indices p and q for which we have to check the compatibility of stack configurations of $B_p^{(i)}$ and $B_q^{(i-1)}$ → linear
- For all stack configuration c of $B_q^{(i-1)}$ and all stack configuration c' of $B_p^{(i)}$, check the compatibility of c and c' and add the right links between configurations of $B_q^{(i)}, \dots, B_p^{(i)}$ → $n \times n \times n$

Complexity ($n = |\sigma|$)

For each right-to-left minima (at most n times):

- Compute the \ominus -decomposition of $\sigma^{(i)} = \ominus[B_1^{(i)}, B_2^{(i)}, \dots, B_s^{(i)}]$
→ linear
- For new $B_j^{(i)}$, compute valid stack configurations of $B_j^{(i)}$ → $n \times n^2$
- Find indices p and q for which we have to check the compatibility of stack configurations of $B_p^{(i)}$ and $B_q^{(i-1)}$ → linear
- For all stack configuration c of $B_q^{(i-1)}$ and all stack configuration c' of $B_p^{(i)}$, check the compatibility of c and c' and add the right links between configurations of $B_q^{(i)}, \dots, B_p^{(i)}$ → $n \times n \times n$
- For indices $j > p$, for all configuration c of $B_j^{(i)}$ and all configuration c' of $B_{j+1}^{(i)}$, add a link between c and c' → $n \times n \times n$

Complexity ($n = |\sigma|$)

For each right-to-left minima (at most n times):

- Compute the \ominus -decomposition of $\sigma^{(i)} = \ominus[B_1^{(i)}, B_2^{(i)}, \dots, B_s^{(i)}]$
→ linear
- For new $B_j^{(i)}$, compute valid stack configurations of $B_j^{(i)}$ → $n \times n^2$
- Find indices p and q for which we have to check the compatibility of stack configurations of $B_p^{(i)}$ and $B_q^{(i-1)}$ → linear
- For all stack configuration c of $B_q^{(i-1)}$ and all stack configuration c' of $B_p^{(i)}$, check the compatibility of c and c' and add the right links between configurations of $B_q^{(i)}, \dots, B_p^{(i)}$ → $n \times n \times n$
- For indices $j > p$, for all configuration c of $B_j^{(i)}$ and all configuration c' of $B_{j+1}^{(i)}$, add a link between c and c' → $n \times n \times n$
- For indices $j < p, q$, do links between configurations of $B_j^{(i)}$ and configurations of $B_{j-1}^{(i)}$ by copying links between $B_j^{(i-1)}$ and $B_{j-1}^{(i-1)}$ → $n \times n \times n$

Complexity ($n = |\sigma|$)

For each right-to-left minima (at most n times):

- Compute the \ominus -decomposition of $\sigma^{(i)} = \ominus[B_1^{(i)}, B_2^{(i)}, \dots, B_s^{(i)}]$
→ linear
- For new $B_j^{(i)}$, compute valid stack configurations of $B_j^{(i)}$ → $n \times n^2$
- Find indices p and q for which we have to check the compatibility of stack configurations of $B_p^{(i)}$ and $B_q^{(i-1)}$ → linear
- For all stack configuration c of $B_q^{(i-1)}$ and all stack configuration c' of $B_p^{(i)}$, check the compatibility of c and c' and add the right links between configurations of $B_q^{(i)}, \dots, B_p^{(i)}$ → $n \times n \times n$
- For indices $j > p$, for all configuration c of $B_j^{(i)}$ and all configuration c' of $B_{j+1}^{(i)}$, add a link between c and c' → $n \times n \times n$
- For indices $j < p, q$, do links between configurations of $B_j^{(i)}$ and configurations of $B_{j-1}^{(i)}$ by copying links between $B_j^{(i-1)}$ and $B_{j-1}^{(i-1)}$
→ $n \times n \times n$

→ Total complexity n^4

- An $O(n^4)$ algorithm deciding if σ is 2-stack sortable

Conclusion

- An $O(n^4)$ algorithm deciding if σ is 2-stack sortable and giving a description of all its sorting verifying some conditions.

Conclusion

- An $O(n^4)$ algorithm deciding if σ is 2-stack sortable and giving a description of all its sorting verifying some conditions.
- No enumeration of 2-stack sortable permutations

Conclusion

- An $O(n^4)$ algorithm deciding if σ is 2-stack sortable and giving a description of all its sorting verifying some conditions.
- No enumeration of 2-stack sortable permutations
- Improve bounds about asymptotic?

Conclusion

- An $O(n^4)$ algorithm deciding if σ is 2-stack sortable and giving a description of all its sorting verifying some conditions.
- No enumeration of 2-stack sortable permutations
- Improve bounds about asymptotic?
- Generalization for $k > 2$ stacks?

Conclusion

- An $O(n^4)$ algorithm deciding if σ is 2-stack sortable and giving a description of all its sorting verifying some conditions.
- No enumeration of 2-stack sortable permutations
- Improve bounds about asymptotic?
- Generalization for $k > 2$ stacks?
- For k fixed, is the problem always polynomial?
Or is it some threshold?

Thank you for your attention