Stack-sorting:
A polynomial decision algorithm

Adeline Pierrot

LRI, Université Paris Sud

Séminaire de combinatoire Philippe Flajolet, october 2014

Joint work with Dominique Rossin, during my PHD at LIAFA

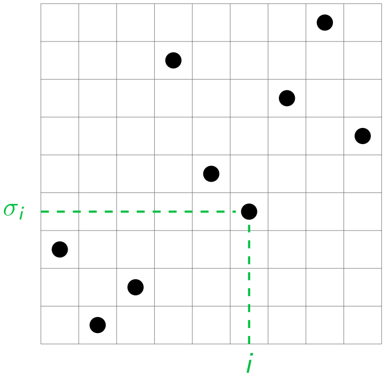
Outline

1. Introduction to stack sorting
2. Pushall sorting (tri par sas)
3. General sorting

Permutations and patterns

Permutation of size n : Order on $[1..n]$

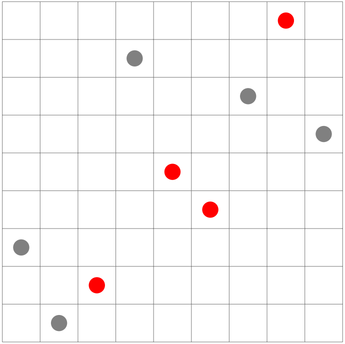
Example : $\sigma = 3\,1\,2\,8\,5\,4\,7\,9\,6$

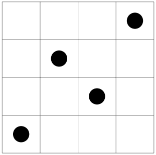


Permutations and patterns

Permutation of size n : Order on $[1..n]$
Example : $\sigma = 3\,1\,2\,8\,5\,4\,7\,9\,6$

Pattern : extracted sub-structure (cf subword)
Example : $1\,3\,2\,4 \preccurlyeq 3\,1\,2\,8\,5\,4\,7\,9\,6$ since $2\,5\,4\,9 \equiv 1\,3\,2\,4$.

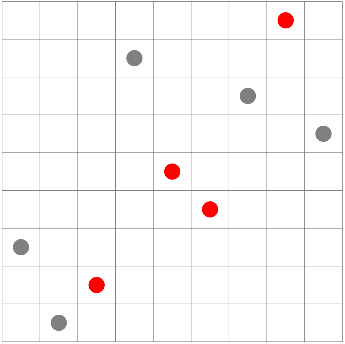


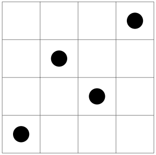


Permutations and patterns

Permutation of size n : Order on $[1..n]$
Example : $\sigma = 3\,1\,2\,8\,5\,4\,7\,9\,6$

Pattern : extracted sub-structure (cf subword)
Example : $1\,3\,2\,4 \preccurlyeq 3\,1\,2\,8\,5\,4\,7\,9\,6$ since $2\,5\,4\,9 \equiv 1\,3\,2\,4$.





Remark: σ, π as input, deciding whether $\pi \preccurlyeq \sigma$ is NP-complete.

Permutation Classes

Class of permutations: set downward closed for $\preccurlyeq$

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

Permutation Classes

Class of permutations: set downward closed for $\preccurlyeq$
Equivalently: $\sigma \in \mathcal{C}$ and $\pi \preccurlyeq \sigma \Rightarrow \pi \in \mathcal{C}$

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

Permutation Classes

Class of permutations: set downward closed for $\preccurlyeq$
Equivalently: $\sigma \in \mathcal{C}$ and $\pi \preccurlyeq \sigma \Rightarrow \pi \in \mathcal{C}$

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

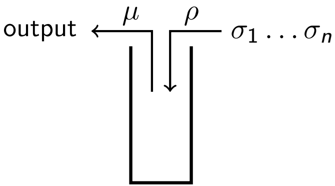
Prop.: Every class $\mathcal{C}$ is characterized by its basis:

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

Basis may be finite or infinite.

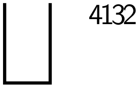
Stack sorting

Stack: last-in first-out device introduced by Knuth (1968).



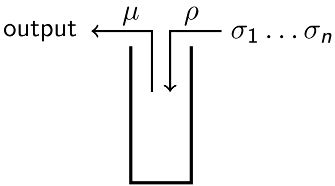
Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:



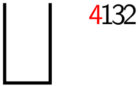
Stack sorting

Stack: last-in first-out device introduced by Knuth (1968).

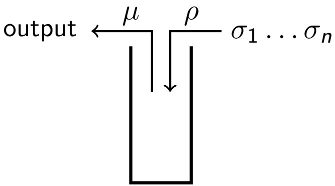


Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:



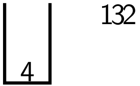
Stack sorting



Stack: last-in first-out device introduced by Knuth (1968).

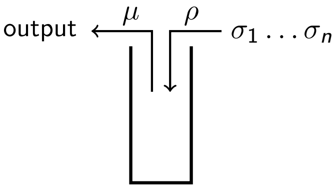
Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:



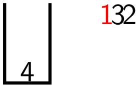
Stack sorting

Stack: last-in first-out device introduced by Knuth (1968).



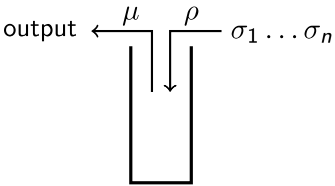
Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:



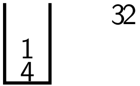
Stack sorting

Stack: last-in first-out device introduced by Knuth (1968).



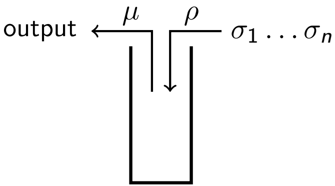
Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:



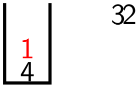
Stack sorting

Stack: last-in first-out device introduced by Knuth (1968).



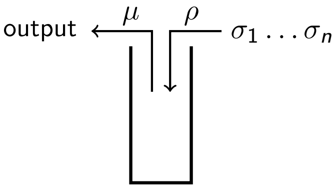
Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:



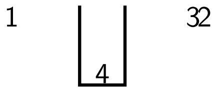
Stack sorting

Stack: last-in first-out device introduced by Knuth (1968).

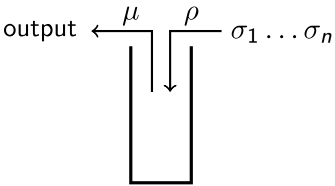


Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:



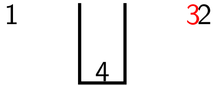
Stack sorting



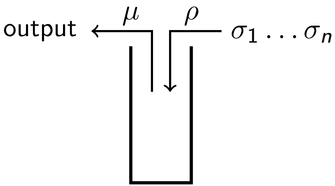
Stack: last-in first-out device introduced by Knuth (1968).

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

Example:



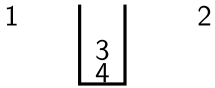
Stack sorting



Stack: last-in first-out device introduced by Knuth (1968).

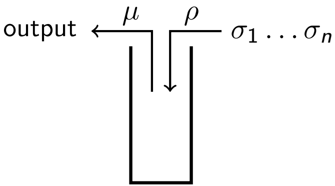
Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:



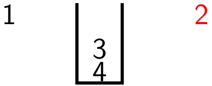
Stack sorting

Stack: last-in first-out device introduced by Knuth (1968).

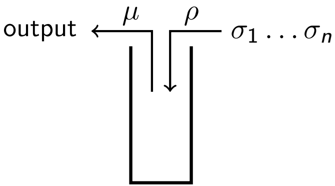


Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:



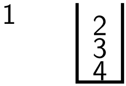
Stack sorting



Stack: last-in first-out device introduced by Knuth (1968).

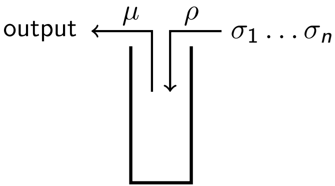
Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:



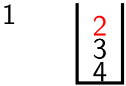
Stack sorting

Stack: last-in first-out device introduced by Knuth (1968).



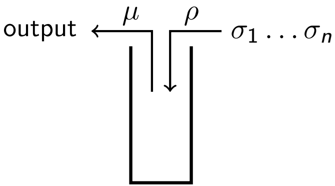
Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:



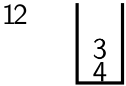
Stack sorting

Stack: last-in first-out device introduced by Knuth (1968).



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

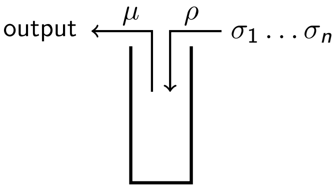
Example:



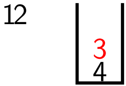
Stack sorting

Stack: last-in first-out device introduced by Knuth (1968).

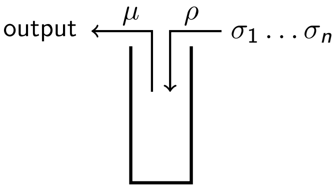
Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.



Example:



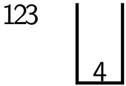
Stack sorting



Stack: last-in first-out device introduced by Knuth (1968).

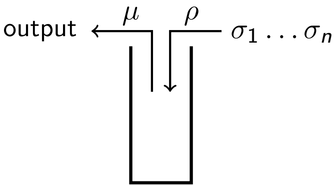
Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:



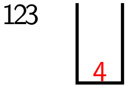
Stack sorting

Stack: last-in first-out device introduced by Knuth (1968).



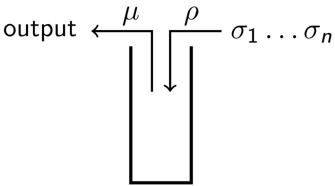
Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:



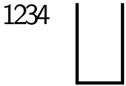
Stack sorting

Stack: last-in first-out device introduced by Knuth (1968).



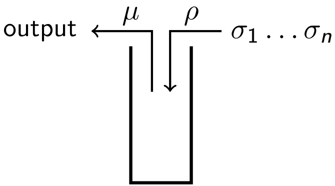
Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:



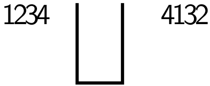
Stack sorting

Stack: last-in first-out device introduced by Knuth (1968).



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

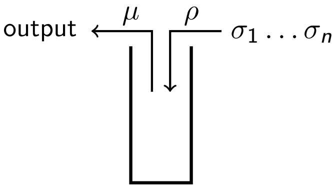
Example:



4132 is sortable

Stack sorting

Stack: last-in first-out device introduced by Knuth (1968).



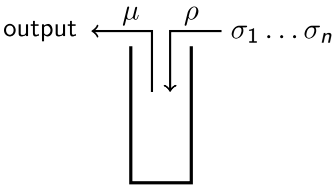
Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

4132 is sortable

Stack sorting

Stack: last-in first-out device introduced by Knuth (1968).



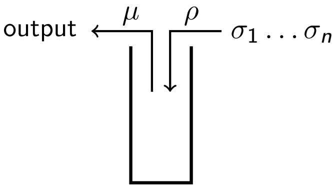
Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

4132 is sortable

Stack sorting

Stack: last-in first-out device introduced by Knuth (1968).



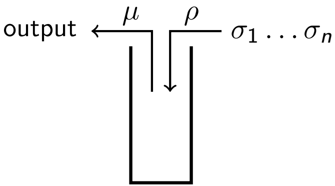
Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

4132 is sortable

Stack sorting

Stack: last-in first-out device introduced by Knuth (1968).

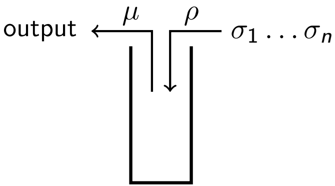


Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

4132 is sortable

Stack sorting



Stack: last-in first-out device introduced by Knuth (1968).

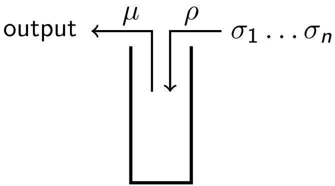
Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

4132 is sortable

Stack sorting

Stack: last-in first-out device introduced by Knuth (1968).



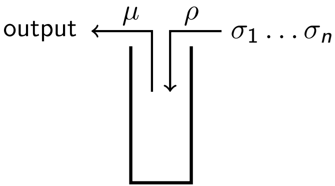
Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

4132 is sortable

Stack sorting

Stack: last-in first-out device introduced by Knuth (1968).



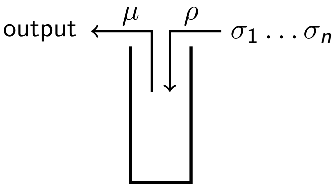
Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

4132 is sortable

Stack sorting

Stack: last-in first-out device introduced by Knuth (1968).



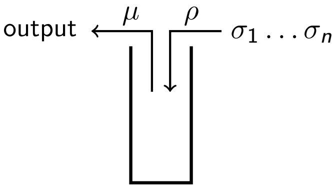
Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

4132 is sortable

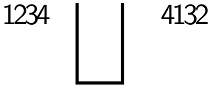
Stack sorting

Stack: last-in first-out device introduced by Knuth (1968).

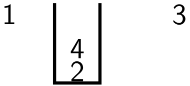


Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:

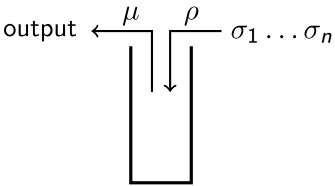


4132 is sortable



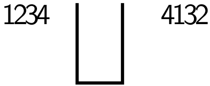
Stack sorting

Stack: last-in first-out device introduced by Knuth (1968).

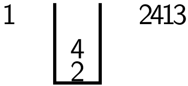


Definition: σ is sortable if $\exists$ a sequence of moves $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity.

Example:



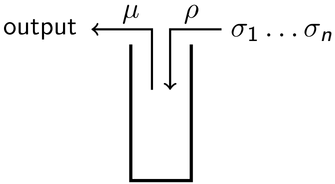
4132 is sortable



2413 is not sortable

Sorting with one stack: a linear algorithm

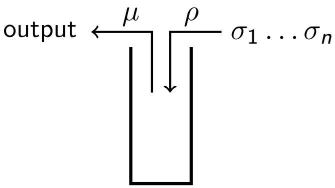
Question: How to decide if σ is sortable?



Sorting with one stack: a linear algorithm

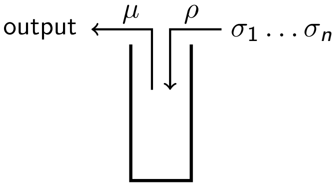
Question: How to decide if σ is sortable?

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?



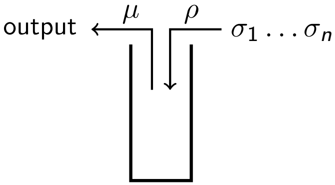
Find $m \in \{\rho, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity
$|\sigma| = n \;\Rightarrow\; |m(\sigma)|_\rho = |m(\sigma)|_\mu = n.$

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

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

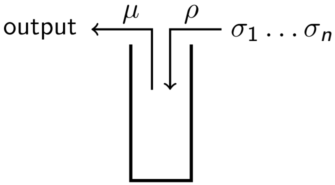
Naive algorithm: Check if $m(\sigma)$ is the identity $\forall\ m \in \{\rho, \mu\}^{2n}$

$\rightarrow$ 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 $\forall\ m \in \{\rho, \mu\}^{2n}$

$\rightarrow$ 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.

$\rightarrow$ 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

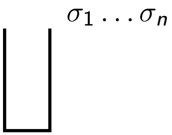
Question: How many sortable permutations of size n?

σ sortable $\Leftrightarrow$ σ avoids 231

Sorting with one stack: a mathematical characterization

Question: How many sortable permutations of size n?

σ sortable $\Leftrightarrow$ σ avoids 231

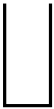


Sorting with one stack: a mathematical characterization

Question: How many sortable permutations of size n?

σ sortable $\Leftrightarrow$ σ avoids 231

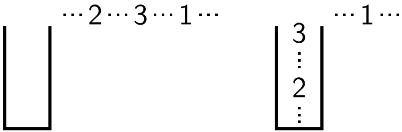
$\cdots 2 \cdots 3 \cdots 1 \cdots$



Sorting with one stack: a mathematical characterization

Question: How many sortable permutations of size n?

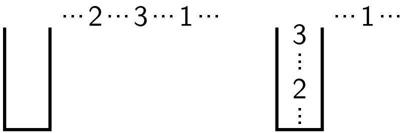
σ sortable $\Leftrightarrow$ σ avoids 231



Sorting with one stack: a mathematical characterization

Question: How many sortable permutations of size n?

σ sortable $\Leftrightarrow$ σ avoids 231

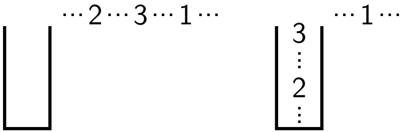


The set of permutations sortable with one stack: $Av(231)$
enumerated by Catalan numbers: $c_n = \frac{1}{n+1}\binom{n}{2n} \approx 4^n \;<<\; n! \approx n^n$

Sorting with one stack: a mathematical characterization

Question: How many sortable permutations of size n?

σ sortable $\Leftrightarrow$ σ avoids 231



The set of permutations sortable with one stack: $Av(231)$
enumerated by Catalan numbers: $c_n = \frac{1}{n+1}\binom{n}{2n} \approx 4^n \;<<\; n! \approx n^n$

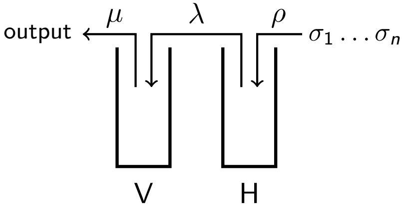
Generalized by Tarjan, Pratt...

Natural questions for sorting devices

- Decision: what is the complexity of the problem consisting of deciding whether a given permutation is sortable or not?
- Characterization: can one characterize permutations that are sortable?
- Counting: how many sortable permutations of size n?

Sorting with two stacks in serie

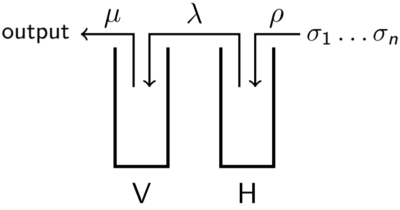
Definition: σ is sortable if $\exists$ $m \in \{\rho, \lambda, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity (Knuth 1973).



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

Sorting with two stacks in serie

Definition: σ is sortable if $\exists\ m \in \{\rho, \lambda, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity (Knuth 1973).



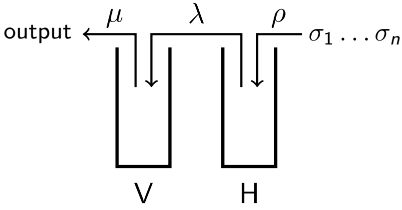
Question: σ a given permutation, is σ sortable with two stacks?

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

$\rightarrow$ exponential algorithm (3^{3n} tests).

Sorting with two stacks in serie

Definition: σ is sortable if $\exists$ $m \in \{\rho, \lambda, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity (Knuth 1973).



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

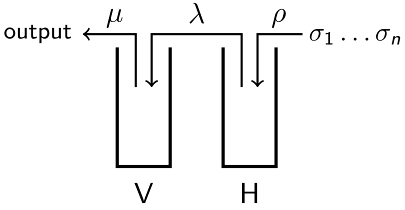
Naive algorithm: Check if $m(\sigma)$ is the identity $\forall$ $m \in \{\rho, \lambda, \mu\}^{3n}$ s.t. $|m(\sigma)|_\rho = |m(\sigma)|_\lambda = |m(\sigma)|_\mu = n$

$\rightarrow$ exponential algorithm (3^{3n} tests).

Is there a polynomial algorithm?

Sorting with two stacks in serie

Definition: σ is sortable if $\exists$ m $\in \{\rho, \lambda, \mu\}^*$ s.t. the output $m(\sigma)$ is the identity (Knuth 1973).



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

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

$\rightarrow$ exponential algorithm (3^{3n} tests).

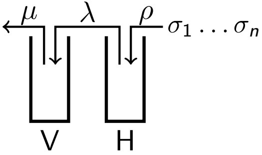
Is there a polynomial algorithm?

Conjectured NP-complete in the litterature

[Atkinson, Murphy, Ruskuc (2002)], [Bona (2003)], [Albert, Atkinson, Linton (2010)]

A canonical way to sort?

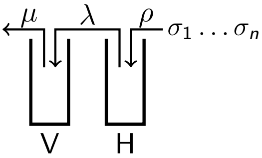
- Non unique way to sort.



A canonical way to sort?

- Non unique way to sort.

Example: moves μ and ρ commute.

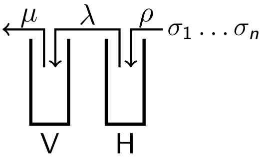


A canonical way to sort?

- Non unique way to sort.

Example: moves μ and ρ commute.

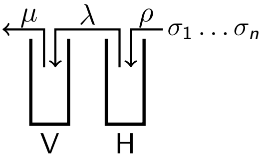
- Canonical sorting?



A canonical way to sort?

- Non unique way to sort.

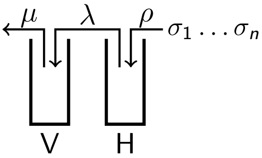
Example: moves μ and ρ commute.



- Canonical sorting?

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

A canonical way to sort?



- Non unique way to sort.

Example: moves μ and ρ commute.

- Canonical sorting?

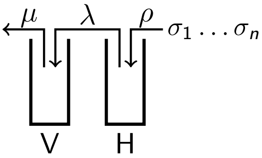
$\mu \Leftrightarrow$ top of V is the next element to be output.

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

A canonical way to sort?

- Non unique way to sort.

Example: moves μ and ρ commute.



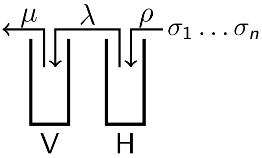
- Canonical sorting?

$\mu \Leftrightarrow$ 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 $2^{(n-1)}$ differents ways.

No way to choose between move λ and move ρ

A canonical way to sort?



- Non unique way to sort.

Example: moves μ and ρ commute.

- Canonical sorting?

$\mu \Leftrightarrow$ 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 $2^{(n-1)}$ differents ways.

No way to choose between move λ and move ρ

Several weaker variants have been studied:
Greedy algorithm (West 93), Increasing stacks (Murphy 02)...

A permutation class

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

sorting procedure for σ $\rightarrow$ sorting procedure for π

A permutation class

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

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

$\hookrightarrow \quad \sigma$ 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

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

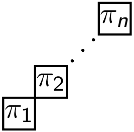
$\hookrightarrow$ sortable permutations form a class $Av(B)$

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

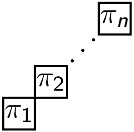
Decomposition

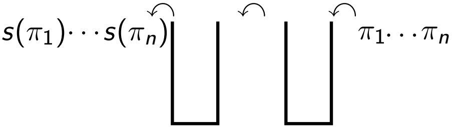
- $\sigma = \oplus[\pi_1, \ldots, \pi_n]$



Decomposition

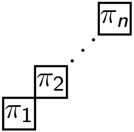
- $\sigma = \oplus[\pi_1, \ldots, \pi_n]$ is sortable $\Leftrightarrow$ $\forall i$, π_i is sortable.

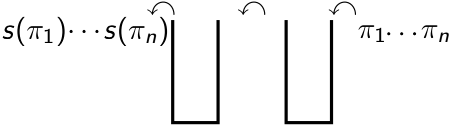




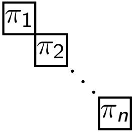
Decomposition

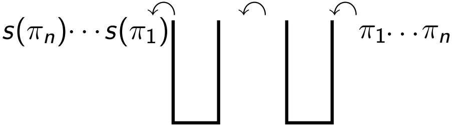
- $\sigma = \oplus[\pi_1, \ldots, \pi_n]$ is sortable $\Leftrightarrow \forall i$, π_i is sortable.





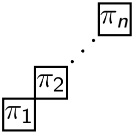
- $\sigma = \ominus[\pi_1, \ldots, \pi_n]$ is sortable $\Rightarrow \forall i$, π_i is sortable.

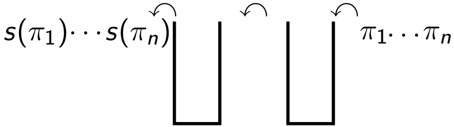




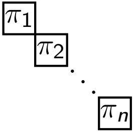
Decomposition

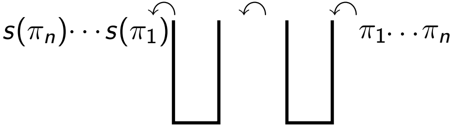
- $\sigma = \oplus[\pi_1, \ldots, \pi_n]$ is sortable $\Leftrightarrow \forall i$, π_i is sortable.





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

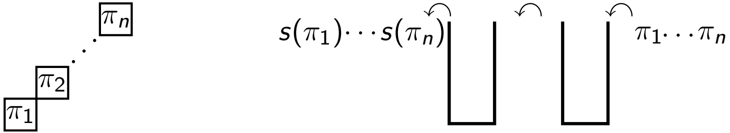




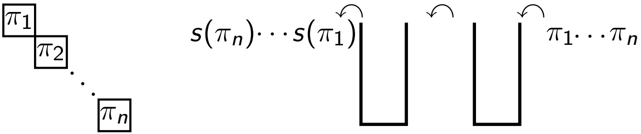
Converse not true: π_i has to admit a special sorting in 2 steps:

Decomposition

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



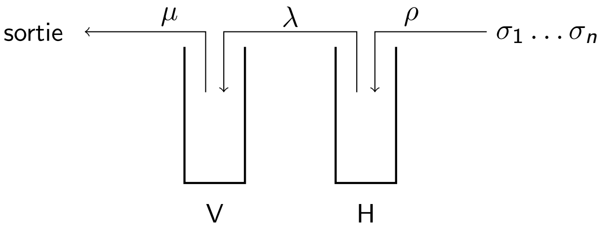
- $\sigma = \ominus[\pi_1, \ldots, \pi_n]$ is sortable $\Rightarrow \forall i$, π_i is sortable.



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

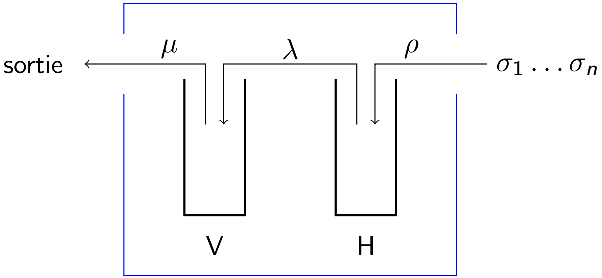
$\sigma = \ominus[\pi_1, \ldots, \pi_n]$ is sortable $\Leftrightarrow \forall i < n$, π_i is pushall sortable and π_n is sortable.

Pushall sorting



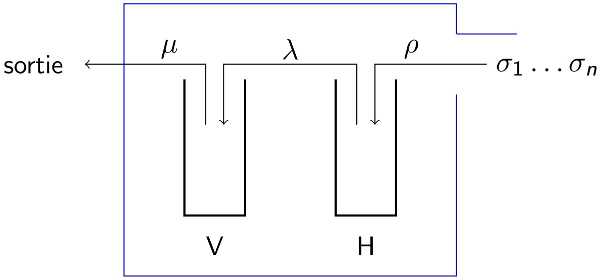
A sorting in 2 parts : first one $\in \{\rho, \lambda\}^*$, second one $\in \{\lambda, \mu\}^*$

Pushall sorting



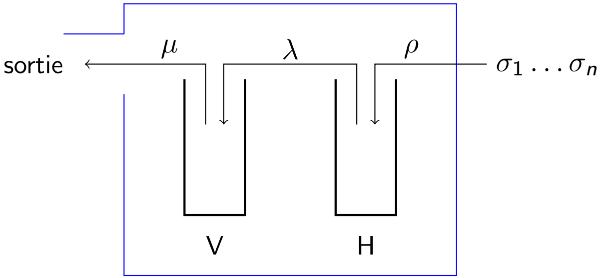
A sorting in 2 parts : first one $\in \{\rho, \lambda\}^*$, second one $\in \{\lambda, \mu\}^*$

Pushall sorting



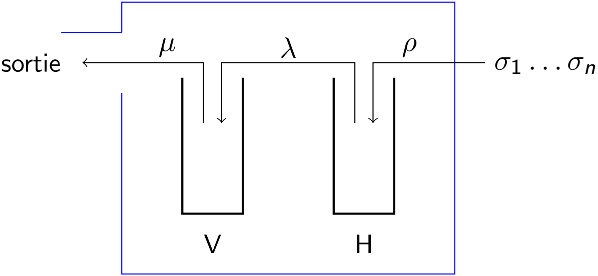
A sorting in 2 parts : first one $\in \{\rho, \lambda\}^*$, second one $\in \{\lambda, \mu\}^*$

Pushall sorting



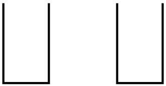
A sorting in 2 parts : first one $\in \{\rho, \lambda\}^*$, second one $\in \{\lambda, \mu\}^*$

Pushall sorting

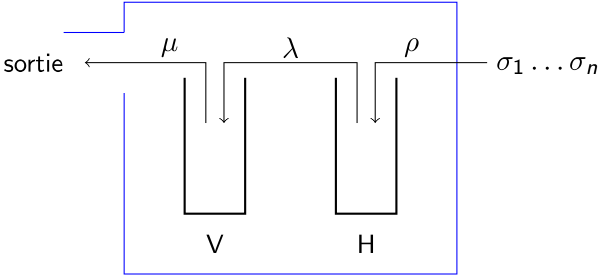


A sorting in 2 parts : first one $\in \{\rho, \lambda\}^*$, second one $\in \{\lambda, \mu\}^*$

Example : 2 4 1 3 is pushall sortable:

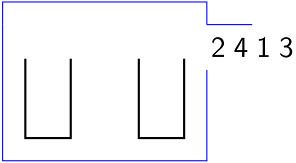


Pushall sorting

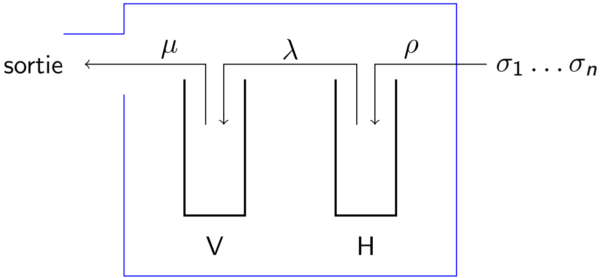


A sorting in 2 parts : first one $\in \{\rho, \lambda\}^*$, second one $\in \{\lambda, \mu\}^*$

Example : 2 4 1 3 is pushall sortable:

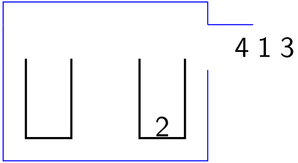


Pushall sorting

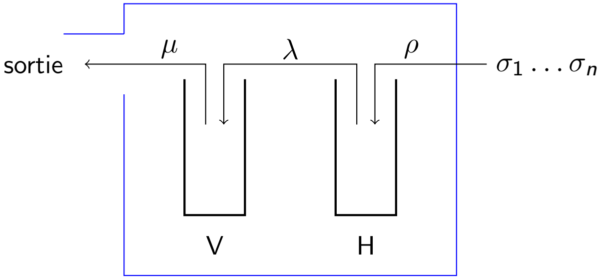


A sorting in 2 parts : first one $\in \{\rho, \lambda\}^*$, second one $\in \{\lambda, \mu\}^*$

Example : 2 4 1 3 is pushall sortable:

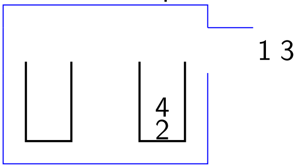


Pushall sorting

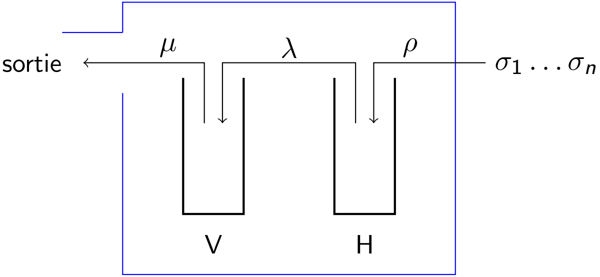


A sorting in 2 parts : first one $\in \{\rho, \lambda\}^*$, second one $\in \{\lambda, \mu\}^*$

Example : 2 4 1 3 is pushall sortable:

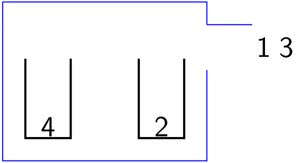


Pushall sorting

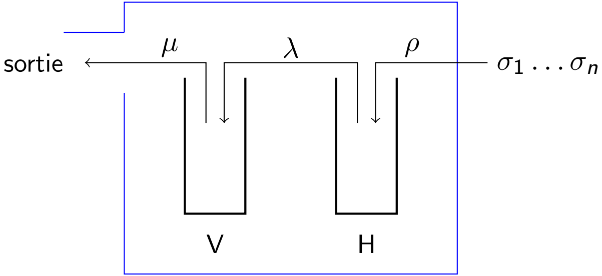


A sorting in 2 parts : first one $\in \{\rho, \lambda\}^*$, second one $\in \{\lambda, \mu\}^*$

Example : 2 4 1 3 is pushall sortable:

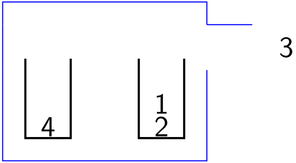


Pushall sorting

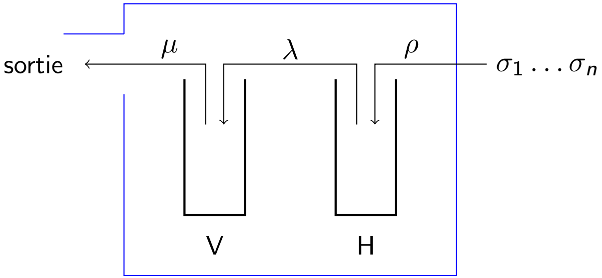


A sorting in 2 parts : first one $\in \{\rho, \lambda\}^*$, second one $\in \{\lambda, \mu\}^*$

Example : 2 4 1 3 is pushall sortable:

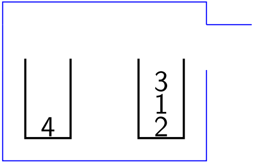


Pushall sorting

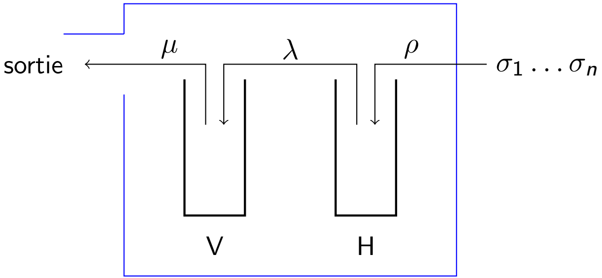


A sorting in 2 parts : first one $\in \{\rho, \lambda\}^*$, second one $\in \{\lambda, \mu\}^*$

Example : 2 4 1 3 is pushall sortable:

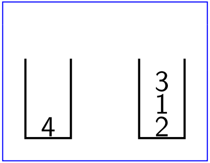


Pushall sorting

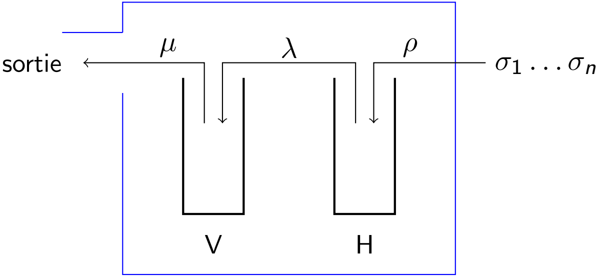


A sorting in 2 parts : first one $\in \{\rho, \lambda\}^*$, second one $\in \{\lambda, \mu\}^*$

Example : 2 4 1 3 is pushall sortable:

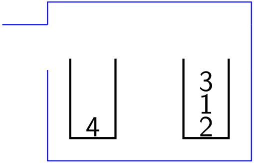


Pushall sorting

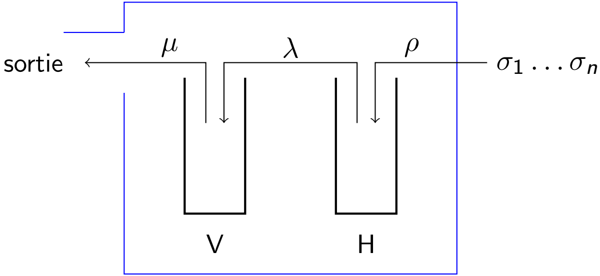


A sorting in 2 parts : first one $\in \{\rho, \lambda\}^*$, second one $\in \{\lambda, \mu\}^*$

Example : 2 4 1 3 is pushall sortable:

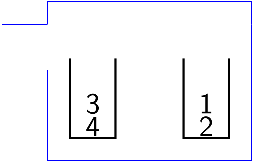


Pushall sorting

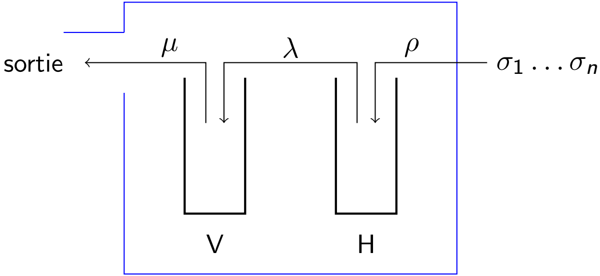


A sorting in 2 parts : first one $\in \{\rho, \lambda\}^*$, second one $\in \{\lambda, \mu\}^*$

Example : 2 4 1 3 is pushall sortable:

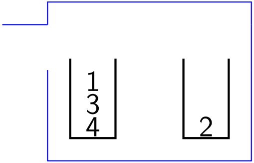


Pushall sorting

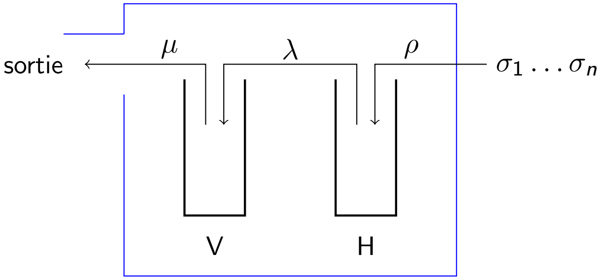


A sorting in 2 parts : first one $\in \{\rho, \lambda\}^*$, second one $\in \{\lambda, \mu\}^*$

Example : 2 4 1 3 is pushall sortable:

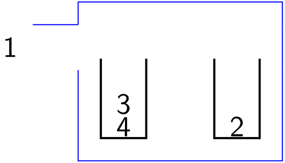


Pushall sorting

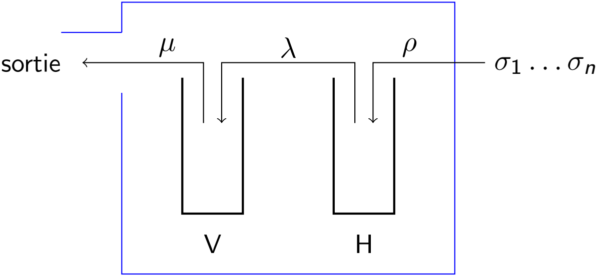


A sorting in 2 parts : first one $\in \{\rho, \lambda\}^*$, second one $\in \{\lambda, \mu\}^*$

Example : 2 4 1 3 is pushall sortable:

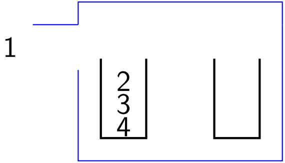


Pushall sorting

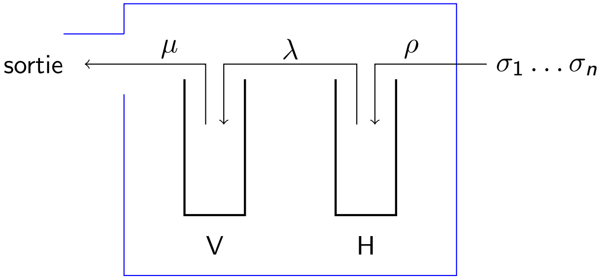


A sorting in 2 parts : first one $\in \{\rho, \lambda\}^*$, second one $\in \{\lambda, \mu\}^*$

Example : 2 4 1 3 is pushall sortable:

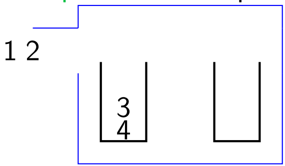


Pushall sorting

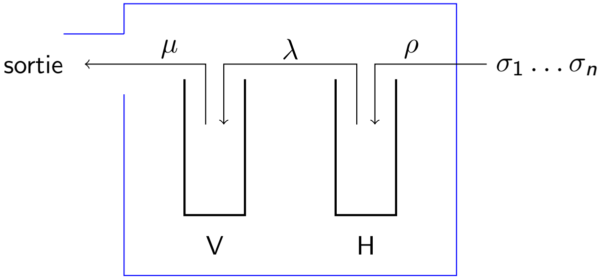


A sorting in 2 parts : first one $\in \{\rho, \lambda\}^*$, second one $\in \{\lambda, \mu\}^*$

Example : 2 4 1 3 is pushall sortable:

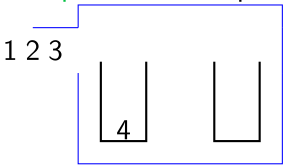


Pushall sorting

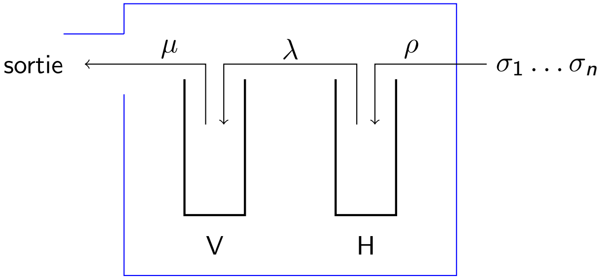


A sorting in 2 parts : first one $\in \{\rho, \lambda\}^*$, second one $\in \{\lambda, \mu\}^*$

Example : 2 4 1 3 is pushall sortable:

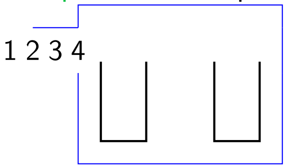


Pushall sorting

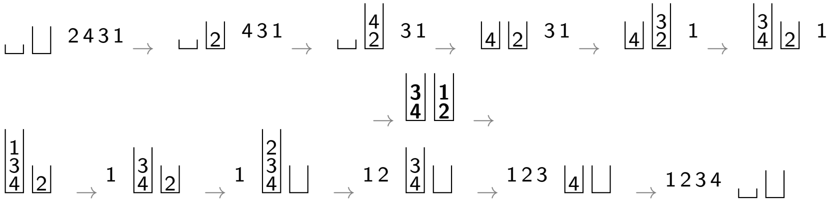


A sorting in 2 parts : first one $\in \{\rho, \lambda\}^*$, second one $\in \{\lambda, \mu\}^*$

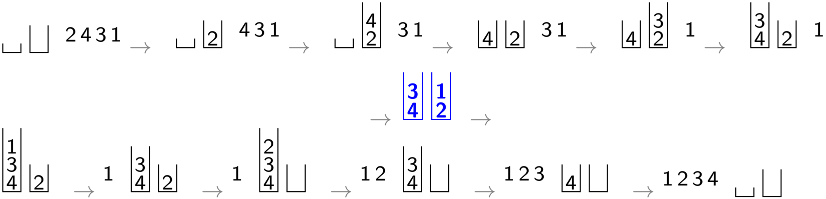
Example : 2 4 1 3 is pushall sortable:



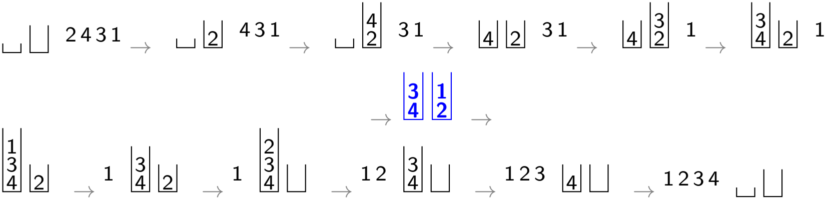
Encoding a pushall sorting



Encoding a pushall sorting

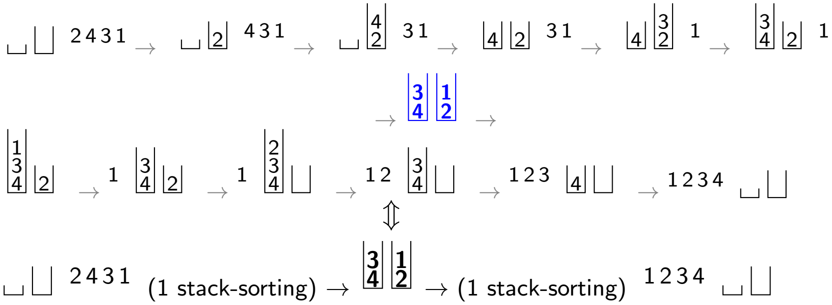


Encoding a pushall sorting



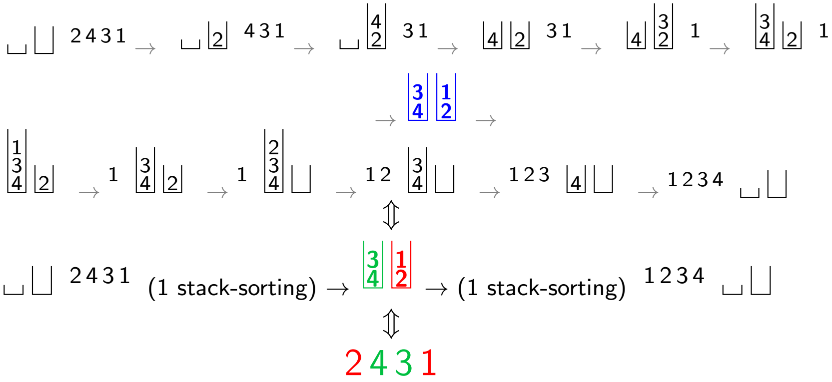
total configuration

Encoding a pushall sorting



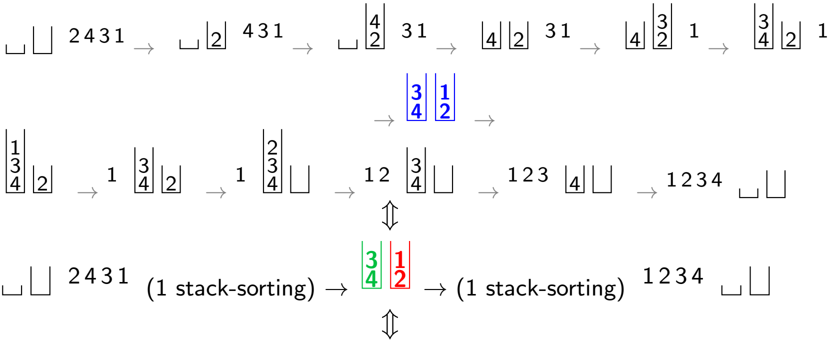
Pushall sorting process $\Leftrightarrow$ valid configuration

Encoding a pushall sorting



Pushall sorting process $\Leftrightarrow$ valid configuration

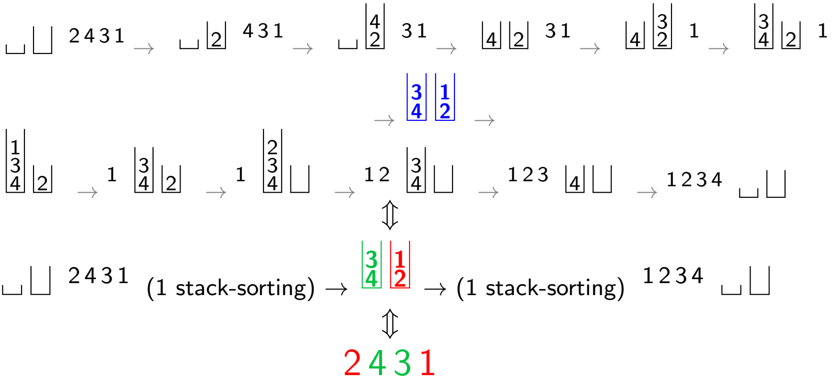
Encoding a pushall sorting



2 4 3 1

Pushall sorting process ⇔ valid configuration ⇔ valid coloring

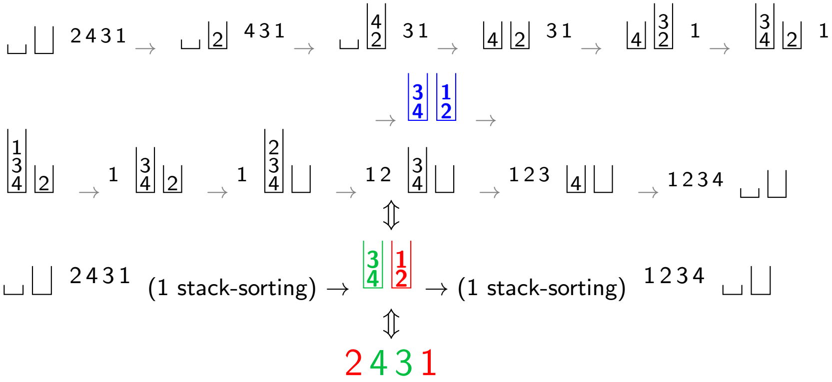
Encoding a pushall sorting



Pushall sorting process ⇔ valid configuration ⇔ valid coloring

→ Test in linear time whether a coloring is valid.

Encoding a pushall sorting



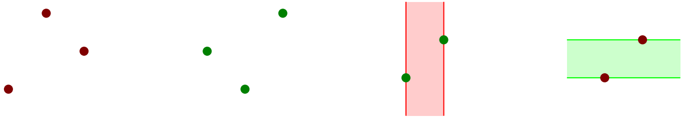
Pushall sorting process $\Leftrightarrow$ valid configuration $\Leftrightarrow$ valid coloring

$\rightarrow$ Test in linear time whether a coloring is valid.
2^n colorings to test $\rightarrow$ reduce this number.

Valid coloring: characterization

Valid 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

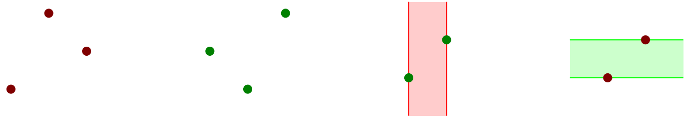


$\Rightarrow$ coloring with forbidden patterns 132, 213, 1X2 and 2/13

Valid coloring: characterization

Valid 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



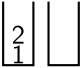
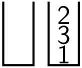
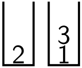
$\Rightarrow$ coloring with forbidden patterns 132, 213, 1X2 and 2/13

Proof: R = right stack and G = left stack $\Rightarrow$ bijection between these colorings and valid stack-configurations.

Proof

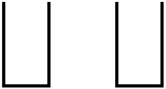
Sortable stack-configuration $\Leftrightarrow$ avoids $\begin{array}{|c|}2\\1\\\hline\end{array}\ \begin{array}{|c|}\quad\\\hline\end{array}$, $\begin{array}{|c|}\quad\\\hline\end{array}\ \begin{array}{|c|}2\\3\\1\\\hline\end{array}$ and $\begin{array}{|c|}2\\\hline\end{array}\ \begin{array}{|c|}3\\1\\\hline\end{array}$

Proof

Sortable stack-configuration $\Leftrightarrow$ avoids  ,  and 

Recall: Forbidden colored patterns = 132, 213, 1X2 and 2/13.
Correspondence between stack-patterns and colored patterns.

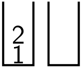
- If the coloring comes from a sorting process, then it avoids the colored patterns:

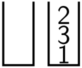


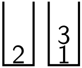
132

- If the coloring avoids the colored patterns, then we obtain a sorting process (no choice to put the elements of R in the right stack and thoses of G in the left stack in the right order).

Proof

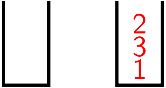
Sortable stack-configuration $\Leftrightarrow$ avoids 

, 

and 

Recall: Forbidden colored patterns = 132, 213, 1X2 and 2/13.
Correspondence between stack-patterns and colored patterns.

- If the coloring comes from a sorting process, then it avoids the colored patterns:



- If the coloring avoids the colored patterns, then we obtain a sorting process (no choice to put the elements of *R* in the right stack and thoses of *G* in the left stack in the right order).

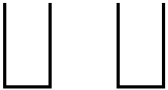
Proof

Sortable stack-configuration $\Leftrightarrow$ avoids $\left\lfloor\begin{matrix}2\\1\end{matrix}\right\rfloor \left\lfloor\quad\right\rfloor$, $\left\lfloor\quad\right\rfloor \left\lfloor\begin{matrix}2\\3\\1\end{matrix}\right\rfloor$ and $\left\lfloor 2\right\rfloor \left\lfloor\begin{matrix}3\\1\end{matrix}\right\rfloor$

Recall: Forbidden colored patterns = 132, 213, 1X2 and 2/13.
Correspondence between stack-patterns and colored patterns.

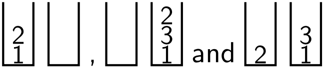
- If the coloring comes from a sorting process, then it avoids the colored patterns:

213



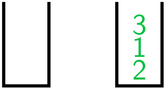
- If the coloring avoids the colored patterns, then we obtain a sorting process (no choice to put the elements of R in the right stack and thoses of G in the left stack in the right order).

Proof

Sortable stack-configuration $\Leftrightarrow$ avoids 

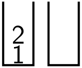
Recall: Forbidden colored patterns = 132, 213, 1X2 and 2/13.
Correspondence between stack-patterns and colored patterns.

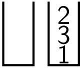
- If the coloring comes from a sorting process, then it avoids the colored patterns:

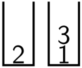


- If the coloring avoids the colored patterns, then we obtain a sorting process (no choice to put the elements of R in the right stack and thoses of G in the left stack in the right order).

Proof

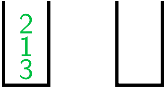
Sortable stack-configuration $\Leftrightarrow$ avoids 

, 

and 

Recall: Forbidden colored patterns = 132, 213, 1X2 and 2/13.
Correspondence between stack-patterns and colored patterns.

- If the coloring comes from a sorting process, then it avoids the colored patterns:



- If the coloring avoids the colored patterns, then we obtain a sorting process (no choice to put the elements of R in the right stack and thoses of G in the left stack in the right order).

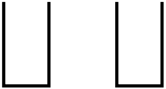
Proof

Sortable stack-configuration ⇔ avoids $\begin{array}{|c|}2\\1\\\hline\end{array}\;\begin{array}{|c|}\\\hline\end{array}$, $\begin{array}{|c|}\\\hline\end{array}\;\begin{array}{|c|}2\\3\\1\\\hline\end{array}$ and $\begin{array}{|c|}2\\\hline\end{array}\;\begin{array}{|c|}3\\1\\\hline\end{array}$

Recall: Forbidden colored patterns = 132, 213, 1X2 and 2/13.
Correspondence between stack-patterns and colored patterns.

- If the coloring comes from a sorting process, then it avoids the colored patterns:

1X2



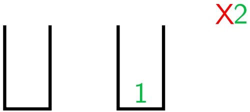
- If the coloring avoids the colored patterns, then we obtain a sorting process (no choice to put the elements of *R* in the right stack and thoses of *G* in the left stack in the right order).

Proof

Sortable stack-configuration $\Leftrightarrow$ avoids $\begin{array}{|c|}2\\1\\\hline\end{array}\ \begin{array}{|c|}\ \\\hline\end{array}$, $\begin{array}{|c|}\ \\\hline\end{array}\ \begin{array}{|c|}2\\3\\1\\\hline\end{array}$ and $\begin{array}{|c|}2\\\hline\end{array}\ \begin{array}{|c|}3\\1\\\hline\end{array}$

Recall: Forbidden colored patterns = 132, 213, 1X2 and 2/13.
Correspondence between stack-patterns and colored patterns.

- If the coloring comes from a sorting process, then it avoids the colored patterns:



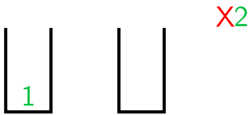
- If the coloring avoids the colored patterns, then we obtain a sorting process (no choice to put the elements of R in the right stack and thoses of G in the left stack in the right order).

Proof

Sortable stack-configuration $\Leftrightarrow$ avoids $\begin{array}{|c|}2\\1\\\hline\end{array}\ \begin{array}{|c|}\ \\\hline\end{array}$, $\begin{array}{|c|}\ \\\hline\end{array}\ \begin{array}{|c|}2\\3\\1\\\hline\end{array}$ and $\begin{array}{|c|}2\\\hline\end{array}\ \begin{array}{|c|}3\\1\\\hline\end{array}$

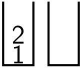
Recall: Forbidden colored patterns = 132, 213, 1X2 and 2/13.
Correspondence between stack-patterns and colored patterns.

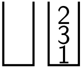
- If the coloring comes from a sorting process, then it avoids the colored patterns:

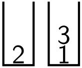


- If the coloring avoids the colored patterns, then we obtain a sorting process (no choice to put the elements of R in the right stack and thoses of G in the left stack in the right order).

Proof

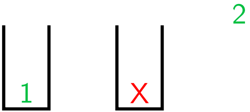
Sortable stack-configuration $\Leftrightarrow$ avoids 

, 

and 

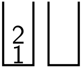
Recall: Forbidden colored patterns = 132, 213, 1X2 and 2/13.
Correspondence between stack-patterns and colored patterns.

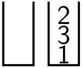
- If the coloring comes from a sorting process, then it avoids the colored patterns:

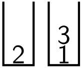


- If the coloring avoids the colored patterns, then we obtain a sorting process (no choice to put the elements of R in the right stack and thoses of G in the left stack in the right order).

Proof

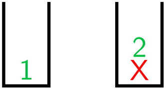
Sortable stack-configuration $\Leftrightarrow$ avoids 

, 

and 

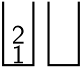
Recall: Forbidden colored patterns = 132, 213, 1X2 and 2/13.
Correspondence between stack-patterns and colored patterns.

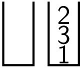
- If the coloring comes from a sorting process, then it avoids the colored patterns:

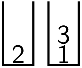


- If the coloring avoids the colored patterns, then we obtain a sorting process (no choice to put the elements of *R* in the right stack and thoses of *G* in the left stack in the right order).

Proof

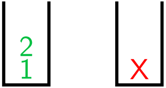
Sortable stack-configuration ⇔ avoids 

, 

and 

Recall: Forbidden colored patterns = 132, 213, 1X2 and 2/13.
Correspondence between stack-patterns and colored patterns.

- If the coloring comes from a sorting process, then it avoids the colored patterns:



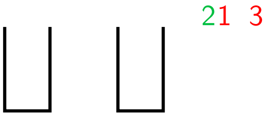
- If the coloring avoids the colored patterns, then we obtain a sorting process (no choice to put the elements of R in the right stack and thoses of G in the left stack in the right order).

Proof

Sortable stack-configuration $\Leftrightarrow$ avoids $\begin{array}{|c|}2\\1\\\hline\end{array}\ \begin{array}{|c|}\\\\\hline\end{array}$, $\begin{array}{|c|}\\\\\hline\end{array}\ \begin{array}{|c|}2\\3\\1\\\hline\end{array}$ and $\begin{array}{|c|}\\2\\\hline\end{array}\ \begin{array}{|c|}3\\1\\\hline\end{array}$

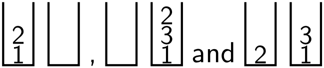
Recall: Forbidden colored patterns = 132, 213, 1X2 and 2/13.
Correspondence between stack-patterns and colored patterns.

- If the coloring comes from a sorting process, then it avoids the colored patterns:



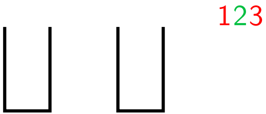
- If the coloring avoids the colored patterns, then we obtain a sorting process (no choice to put the elements of *R* in the right stack and thoses of *G* in the left stack in the right order).

Proof

Sortable stack-configuration $\Leftrightarrow$ avoids 

Recall: Forbidden colored patterns $=$ 132, 213, 1X2 and 2/13.
Correspondence between stack-patterns and colored patterns.

- If the coloring comes from a sorting process, then it avoids the colored patterns:



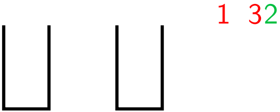
- If the coloring avoids the colored patterns, then we obtain a sorting process (no choice to put the elements of R in the right stack and thoses of G in the left stack in the right order).

Proof

Sortable stack-configuration $\Leftrightarrow$ avoids $\begin{array}{|c|}2\\1\\\hline\end{array}\ \begin{array}{|c|}\ \\\hline\end{array}$, $\begin{array}{|c|}\ \\\hline\end{array}\ \begin{array}{|c|}2\\3\\1\\\hline\end{array}$ and $\begin{array}{|c|}2\\\hline\end{array}\ \begin{array}{|c|}3\\1\\\hline\end{array}$

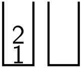
Recall: Forbidden colored patterns = 132, 213, 1X2 and 2/13.
Correspondence between stack-patterns and colored patterns.

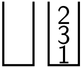
- If the coloring comes from a sorting process, then it avoids the colored patterns:

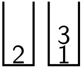


- If the coloring avoids the colored patterns, then we obtain a sorting process (no choice to put the elements of R in the right stack and thoses of G in the left stack in the right order).

Proof

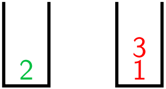
Sortable stack-configuration ⇔ avoids 

, 

and 

Recall: Forbidden colored patterns = 132, 213, 1X2 and 2/13.
Correspondence between stack-patterns and colored patterns.

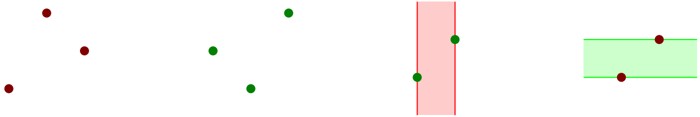
- If the coloring comes from a sorting process, then it avoids the colored patterns:



- If the coloring avoids the colored patterns, then we obtain a sorting process (no choice to put the elements of R in the right stack and thoses of G in the left stack in the right order).

Decomposition

Forbidden colored patterns:

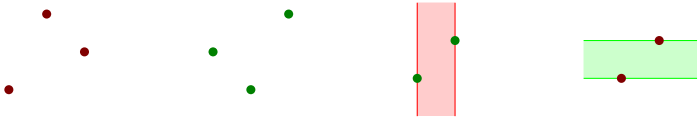


$Col(\sigma)$ = the set of valid colorings of σ

$\sharp Col(n\,(n{-}1)\,\ldots 1) = 2^n$

Decomposition

Forbidden colored patterns:



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

$\sharp Col(n\,(n-1)\,\ldots 1) = 2^n$

$$\ominus[\pi_1,\ldots,\pi_k] = \begin{matrix}\boxed{\pi_1}&&&\\&\boxed{\pi_2}&&\\&&\ddots&\\&&&\boxed{\pi_k}\end{matrix}$$

Example : $\ominus[1,\ldots,1] = n\,(n-1)\,\ldots 1$

Theorem

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

Restrict the number of bicolorings to test

Add hypothesis
to ensure a polynomial number of bicolorings to test.

Restrict the number of bicolorings to test

Add hypothesis
to ensure a polynomial number of bicolorings to test.

- Assume σ is $\ominus$-indecomposable.
  Otherwise $\sigma = \ominus[\pi_1 \ldots \pi_n]$ with π_i $\ominus$-indecomposable
  $\sigma = \ominus[\pi_1, \ldots, \pi_k] \Rightarrow Col(\sigma) \approx Col(\pi_1) \times \cdots \times Col(\pi_k)$
  So replace σ by the π_i.

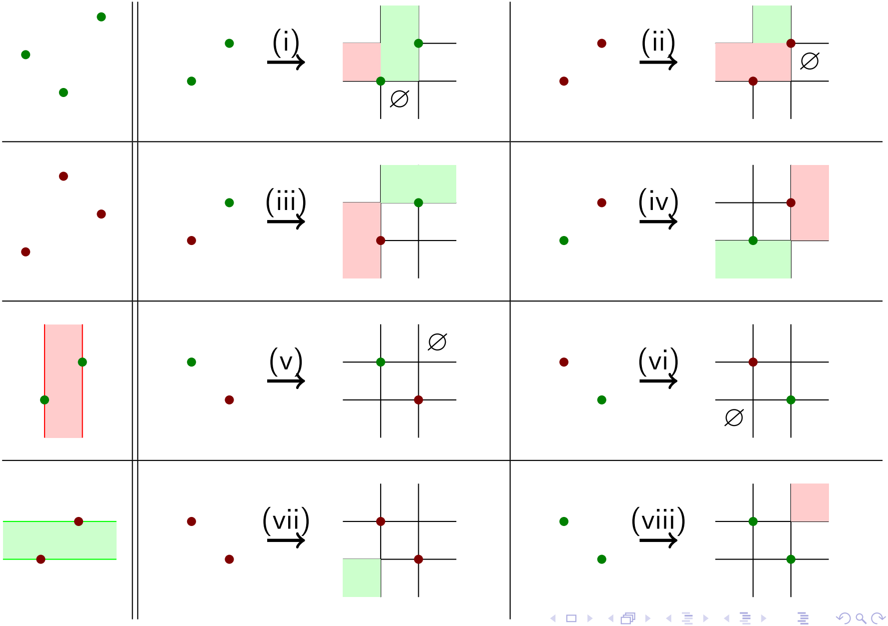
Restrict the number of bicolorings to test

Add hypothesis
to ensure a polynomial number of bicolorings 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.

Forbidden colored patterns ⇒ implication rules



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

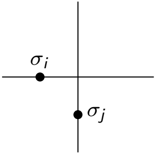
Proposition: σ $\ominus$-indecomposable and C a right coloring of σ where each pattern 12 is unicolor $\Rightarrow$ C is unicolor.

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.

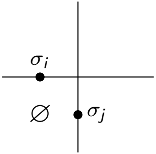


Let σ_i and σ_j consecutive left-to-right minima of σ.

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.

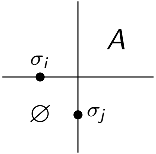


Let σ_i and σ_j consecutive left-to-right minima of σ.

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.



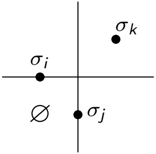
Let σ_i and σ_j consecutive left-to-right minima of σ.

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.



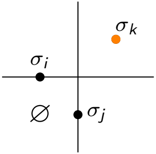
Let σ_i and σ_j consecutive left-to-right minima of σ.

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.

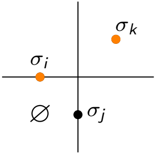
Zone A is non-empty as σ is $\ominus$-indecomposable.

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.

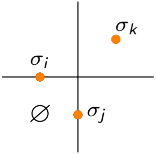
Zone A is non-empty as σ is $\ominus$-indecomposable.

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.

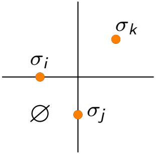
Zone A is non-empty as σ is $\ominus$-indecomposable.

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.

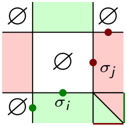
Zone A is non-empty as σ is $\ominus$-indecomposable.

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).

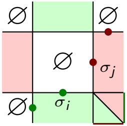
Other cases

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



Other cases

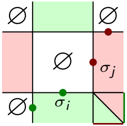
- There are patterns 12 but no pattern 12: Position of the down-rightmost pattern 12 determines all colors:



Moreover, knowing the position of σ_i is sufficient to recover σ_j and determine all colors.

Other cases

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

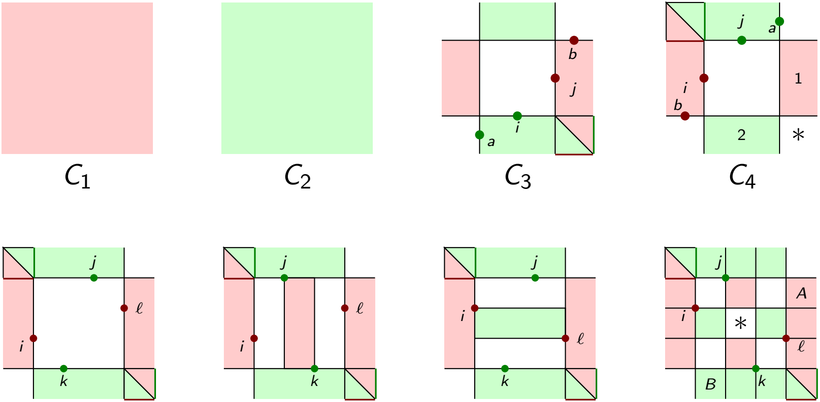


  Moreover, knowing the position of σ_i is sufficient to recover σ_j and determine all colors.

- Similar results for the other cases.

8 kinds of colorings for σ $\ominus$-indecomposable

Theorem : c valid coloring of $\sigma \Rightarrow \exists m, p$ s.t. $c = C_m(p)$.



C_1 C_2 C_3 C_4

C_5 C_6 C_7 C_8

Quadratic algorithm

Algorithm :

Input: σ $\ominus$-indecomposable.
Output: All valid colorings of σ:

For i from 1 to 8
    For p from 1 to $n = |\sigma|$
        Test if $C_i(p)$ is a valid coloring of σ

Quadratic algorithm

Algorithm :

Input: σ $\ominus$-indecomposable.
Output: All valid colorings of σ:

For i from 1 to 8
    For p from 1 to $n = |\sigma|$
        Test if $C_i(p)$ is a valid coloring of σ

Complexity :

Test if a coloring is valid $=$ linear

Quadratic algorithm

Algorithm :

Input: σ $\ominus$-indecomposable.
Output: All valid colorings of σ:

For i from 1 to 8
  For p from 1 to $n = |\sigma|$
    Test if $C_i(p)$ is a valid coloring of σ

Complexity :

Test if a coloring is valid = linear

σ $\ominus$-indecomposable $\Rightarrow |Col(\sigma)| \leq 8|\sigma|$ computed in $\mathcal{O}(|\sigma|^2)$

Quadratic algorithm

Algorithm :

Input: σ $\ominus$-indecomposable.
Output: All valid colorings of σ:

For i from 1 to 8
  For p from 1 to $n = |\sigma|$
    Test if $C_i(p)$ is a valid coloring of σ

Complexity :

Test if a coloring is valid $=$ linear

σ $\ominus$-indecomposable $\Rightarrow |Col(\sigma)| \leq 8|\sigma|$ computed in $\mathcal{O}(|\sigma|^2)$

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

$\rightarrow Col(\sigma)$ described by $\big(Col(\pi_1), \ldots, Col(\pi_k)\big)$

Quadratic algorithm

Algorithm :

Input: σ $\ominus$-indecomposable.
Output: All valid colorings of σ:

For i from 1 to 8
  For p from 1 to $n = |\sigma|$
    Test if $C_i(p)$ is a valid coloring of σ

Complexity :

Test if a coloring is valid $=$ linear

σ $\ominus$-indecomposable $\Rightarrow |Col(\sigma)| \leq 8|\sigma|$ computed in $\mathcal{O}(|\sigma|^2)$

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

$\rightarrow Col(\sigma)$ described by $\big(Col(\pi_1), \ldots, Col(\pi_k)\big)$

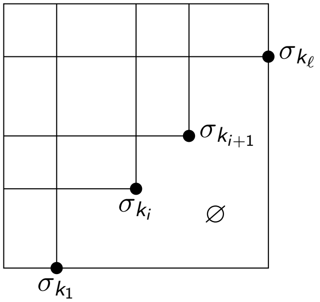
$\rightarrow$ computed in quadratic time: $8|\pi_1|^2 + \cdots + 8|\pi_k|^2 \leq 8|\sigma|^2$.

Outline

1. Introduction to stack sorting
2. Pushall sorting (tri par sas)
3. General sorting

From pushall sorting to general sorting

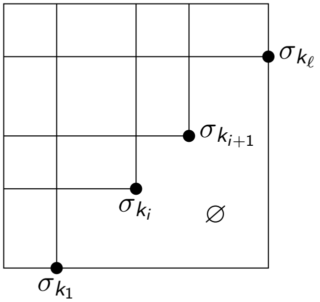
σ_{k_i} = right-left minima of σ



From pushall sorting to general sorting

σ_{k_i} = right-left minima of σ

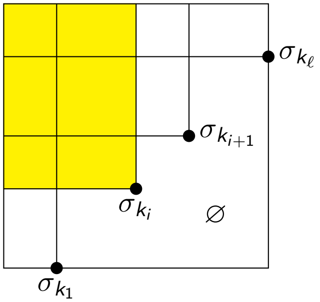
Configuration when σ_{k_i} enters the stacks



From pushall sorting to general sorting

σ_{k_i} = right-left minima of σ

Configuration when σ_{k_i} enters the stacks

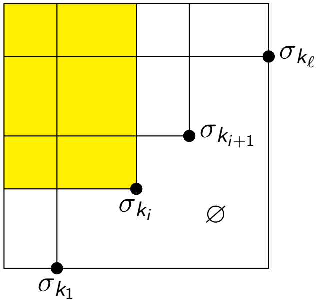


$$\sigma^{(i)} = \{\sigma_j \mid j < k_i \text{ et } \sigma_j > \sigma_{k_i}\}$$

From pushall sorting to general sorting

σ_{k_i} = right-left minima of σ

Configuration when σ_{k_i} enters the stacks = total for $\sigma^{(i)}$

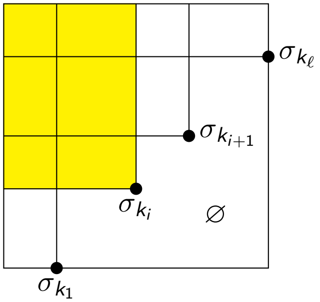


$$\sigma^{(i)} = \{\sigma_j \mid j < k_i \text{ et } \sigma_j > \sigma_{k_i}\}$$

From pushall sorting to general sorting

σ_{k_i} = right-left minima of σ

Configuration when σ_{k_i} enters the stacks = total for $\sigma^{(i)}$



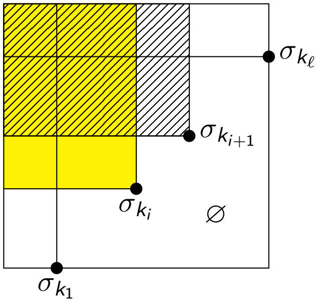
$$\sigma^{(i)} = \{\sigma_j \mid j < k_i \text{ et } \sigma_j > \sigma_{k_i}\}$$

σ sortable $\Rightarrow \sigma^{(i)}$ push-all sortable $\forall i$

From pushall sorting to general sorting

σ_{k_i} = right-left minima of σ

Configuration when σ_{k_i} enters the stacks = total for $\sigma^{(i)}$



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

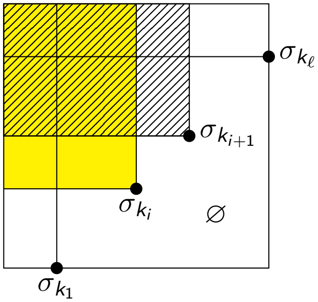
σ sortable $\Rightarrow$ $\sigma^{(i)}$ push-all sortable $\forall i$

The push-all sortings of the $\sigma^{(i)}$ must be compatibles

From pushall sorting to general sorting

σ_{k_i} = right-left minima of σ

Configuration when σ_{k_i} enters the stacks = total for $\sigma^{(i)}$



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

σ sortable $\Rightarrow \sigma^{(i)}$ push-all sortable $\forall i$

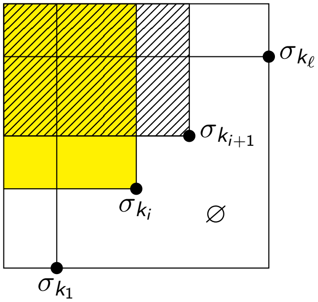
The push-all sortings of the $\sigma^{(i)}$ must be compatibles

Recursive algorithm

From pushall sorting to general sorting

σ_{k_i} = right-left minima of σ

Configuration when σ_{k_i} enters the stacks = total for $\sigma^{(i)}$



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

σ sortable $\Rightarrow$ $\sigma^{(i)}$ push-all sortable $\forall i$

The push-all sortings of the $\sigma^{(i)}$ must be compatibles

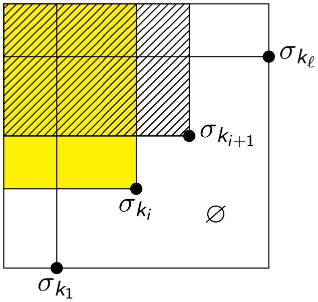
Recursive algorithm

Compatibility test = linear.

From pushall sorting to general sorting

σ_{k_i} = right-left minima of σ

Configuration when σ_{k_i} enters the stacks = total for $\sigma^{(i)}$



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

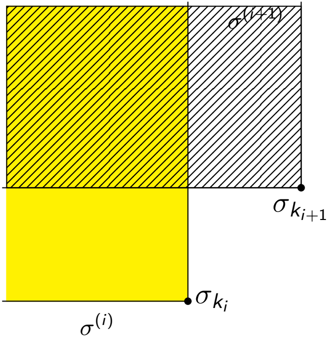
σ sortable $\Rightarrow$ $\sigma^{(i)}$ push-all sortable $\forall i$

The push-all sortings of the $\sigma^{(i)}$ must be compatibles

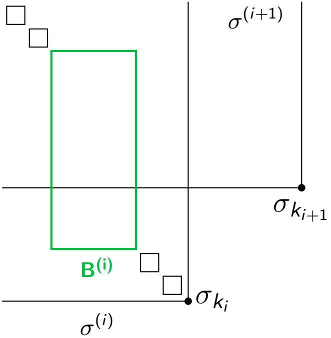
Recursive algorithm

Compatibility test = linear. Exponentiel number of tests?

Reduce the number of tests

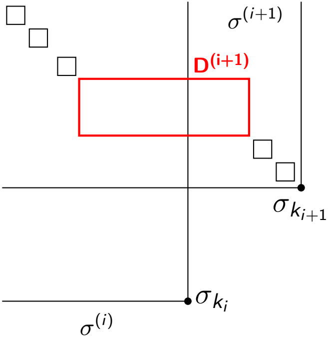


Reduce the number of tests



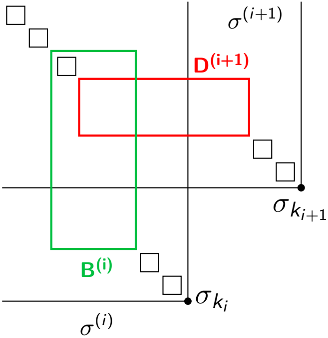
$$Col(\sigma^{(i)}) \approx$$
$$Col(B_1^{(i)}) \times \cdots \times Col(B_k^{(i)})$$

Reduce the number of tests



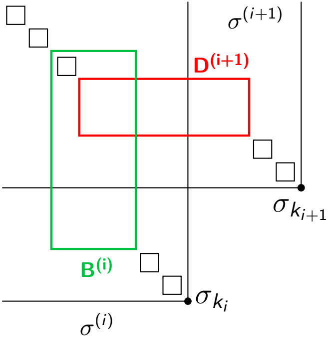
$$Col(\sigma^{(i)}) \approx Col(B_1^{(i)}) \times \cdots \times Col(B_k^{(i)})$$

Reduce the number of tests



$$Col(\sigma^{(i)}) \approx$$
$$Col(B_1^{(i)}) \times \cdots \times Col(B_k^{(i)})$$

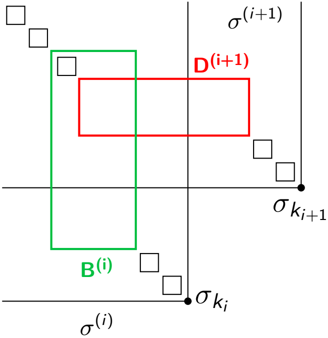
Reduce the number of tests



$$Col(\sigma^{(i)}) \approx Col(B_1^{(i)}) \times \cdots \times Col(B_k^{(i)})$$

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

Reduce the number of tests

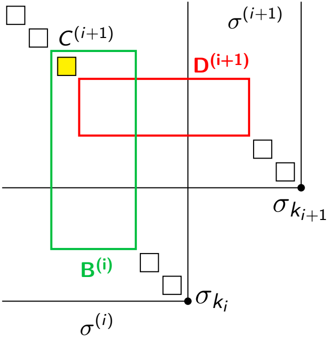


$Col(\sigma^{(i)}) \approx$
$Col(B_1^{(i)}) \times \cdots \times Col(B_k^{(i)})$

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

$\rightarrow$ linear number of tests

Reduce the number of tests



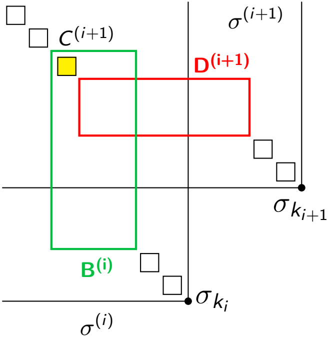
$Col(\sigma^{(i)}) \approx$
$Col(B_1^{(i)}) \times \cdots \times Col(B_k^{(i)})$

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

$\rightarrow$ linear number of tests

Configurations of $C^{(i+1)}$ linked to those of $D^{(i+1)}$

Reduce the number of tests



$Col(\sigma^{(i)}) \approx$
$Col(B_1^{(i)}) \times \cdots \times Col(B_k^{(i)})$

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

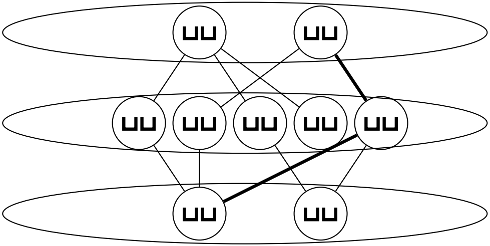
$\rightarrow$ linear number of tests

Configurations of $C^{(i+1)}$ linked to those of $D^{(i+1)}$

$\rightarrow$ sorting graph

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

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



Stack configurations of B_1

Stack configurations of B_2

Stack configurations of B_3

Links between compatibles stack configurations

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

Algorithm

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

At step i, the algorithm returns false if $\sigma_1 \ldots \sigma_{k_i}$ 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.

Conclusion

Polynomial decision algorithm for 2 stacks in series

- New notion: push-all sorting
- Characterization through bicolorings with excluded patterns
- Optimal quadratic algorithm to compute all push-all sortings
- Decomposition along right-left minima
- One gets all sortings satisfying a property P.

Perspectives

- Simplify the algorithm?

Perspectives

- Simplify the algorithm?
- Characterize the permutations sortable with 2 stacks in series? Enumeration?

Perspectives

- Simplify the algorithm?
- Characterize the permutations sortable with 2 stacks in series? Enumeration?
- Enumerate the push-all sortable permutations?

Perspectives

- Simplify the algorithm?
- Characterize the permutations sortable with 2 stacks in series? Enumeration?
- Enumerate the push-all sortable permutations?

- Complexity of the decision algorithm for k stacks in series:

Perspectives

- Simplify the algorithm?
- Characterize the permutations sortable with 2 stacks in series? Enumeration?
- Enumerate the push-all sortable permutations?

- Complexity of the decision algorithm for k stacks in series:
    - Generalize to more than 2 stacks?

Perspectives

- Simplify the algorithm?
- Characterize the permutations sortable with 2 stacks in series? Enumeration?
- Enumerate the push-all sortable permutations?

- Complexity of the decision algorithm for k stacks in series:
  - Generalize to more than 2 stacks?
  - For fixed k, is the problem still polynomial? Is there a threshold?

Perspectives

- Simplify the algorithm?
- Characterize the permutations sortable with 2 stacks in series? Enumeration?
- Enumerate the push-all sortable permutations?

- Complexity of the decision algorithm for k stacks in series:
  - Generalize to more than 2 stacks?
  - For fixed k, is the problem still polynomial? Is there a threshold?

Thank you for your attention