Algorithmic and Enumerative Combinatorics of Lattice Paths and Permutations

Sorting permutations with the DI-machine

Maher Mallem,
M1 Computer Science student at ENS Paris-Saclay,
supervised by Luca Ferrari at DIMAI, Florence

March 5th - July 20th 2018

Contents

1 Definitions and notations

Permutations

Generalities

Let $n \in \mathbb{N}^*$.
The set $\{1, 2, ..., n\}$ will be denoted $[1..n]$.
A permutation of length n is a bijection from $[1..n]$ to $[1..n]$.
Let $\mathfrak{S}_n$ be the group of the permutations of length n.
The identity permutation of length n is denoted Id_n.

Let π be a permutation of length n.
We write permutations in their single-line notation. For example Id_n is written $1...n$.

We define the order $\prec_\pi$ on $[1..n]$ according to the position of the elements in π:
if $a, b \in [1..n]$, then: $a \prec_\pi b$ iff a is at the left of b in the single-line notation of π.

If $i \in [1..n]$, we define: $max_i(\pi) \stackrel{def}{=} max(\{\pi(j), 1 \leq j \leq i\})$.
Let $b \in [1..n]$.
b is called an intermediate maximum iff $b = max_{\pi^{-1}(b)}(\pi)$.
In other words, if we read π from left to right, b is the biggest encountered value when we reach it.

Let $a, b \in [1..n], a < b$.
$\{a, b\}$ is called an inversion of π iff $b \prec_\pi a$.
Otherwise, it is called a non-inversion of π.

A peak is an element of π that has a predecessor and a successor and that is greater than both of them.
A valley is an element of π that has a predecessor and a successor and that is lower than both of them.
We define: $\alpha(\pi)$ the number of valleys that have a peak on their right (with no other valley in between).

Permutation patterns

If $\pi \in \mathfrak{S}_n$, a sub-permutation τ of π is a (non-necessarily continuous) subsequence of the elements of π:
$\tau = i_1...i_k$ such that $i_1 \prec_\pi ... \prec_\pi i_k$.
In τ if we replace the lowest element by 1, the second lowest element by 2 etc., we get $\overline{\tau}$ a permutation of $\mathfrak{S}_k$ which is equivalent to τ in terms of the relative ordering of the elements. We call $\overline{\tau}$ the reduction - or the type - of τ.

If $p \in \mathfrak{S}_k$, if $\pi \in \mathfrak{S}_n$ with $k \leq n$, we call p a pattern of π iff p is the reduction of one of the subpermutations of π.
For example 231 is a pattern of 3241 because it is the reduction of the subpermutation 341.

The relation of "being a pattern of" defines a partial order on the set of all the permutations: $\bigcup_{k=1}^{\infty} \mathfrak{S}_k$.
π_1 and π_2 are called independent iff $\pi 1$ is not a pattern of π_2 and π_2 is not a pattern of π_1.

For example 231 and 2134 are independent.

Stack-sorting devices

The sorting devices we consider are algorithms over a sequence of stacks $S = S_1...S_k$.
The input is denoted IN, the output is denoted OUT.

An increasing stack, or I-stack, is a stack that always guarantees the following property:
if a and b are in the stack and b is deeper than a, then $a < b$.
In other words, if any moment you go through the elements in the I-stack from top to bottom, you will always get an increasing sequence.

A decreasing stack, or D-stack, is defined in the same way.

For the remaining of this study, we will only consider machines that are combinations of I-stacks and D-stacks.

We put the following restrictions in the machines:
- Whatever goes out of IN goes into S_1.
- For every i in $[1..n]$, whatever goes out of S_i goes into S_{i+1}.
- Whatever goes out of S_k goes into OUT.

Thus for all i in $[1..k]$, every a in $[1..n]$ goes into S_k exactly once.
We denote $(S_i)_a$ the action of pushing a to S_i. Similarly, we denote O_a the action of pushing a to OUT. These are the elementary actions of the machine as the only thing it can do at every step is either one of these actions or a sequence of them.

If (S, Algo) is a stack-sorting machine, we define the order $\prec_{S,Algo,\pi}$ on the elementary actions according to the sequence of elementary actions done by (S, Algo) with π as the input:
$X \prec_{S,Algo,\pi} Y$ iff X is done before Y by (S, Algo) with π as the input.
For example for all i in $[1..k-1]$, for all a in $[1..n]$, we know that:
$(S_i)_a \prec_{S,Algo,\pi} (S_{i+1})_a$
from the restrictions we put.

At the beginning IN is a permutation π, which can be seen as a queue of numbers according to the single-line notation.
If the algorithm terminates, the sequence of numbers in OUT is the resulting permutation $S_{Algo}(\pi)$.
When there is no ambiguity about which algorithm is used, we simply denote it $S(\pi)$.
If it terminates after k iterations, the resulted permutation is denoted $S^k_{Algo}(\pi)$ (or $S^k(\pi)$).

For every i in $[1..n]$, the top of the stack S_i is denoted t_{S_i}.
The top of IN is denoted t_{IN}.

An empty stack is denoted $\perp$.

The I-machine

The I-machine is the procedure described in Algorithm 1 with the sequence of stacks S = I.

Algorithm 1 The I-machine

```
while IN is not empty do
  if I = ⊥ or t_IN < t_I then
    push t_IN to I
  else
    push t_I to OUT
  end if
end while
push everything from I to OUT
```

The DI-machine

The DI-machine is the procedure described in Algorithm 2 with the sequence of stacks S = DI. From section 3.1.3 to the remaining of the report, the DI-machine will use Algorithm 4 instead.

Algorithm 2 The initial DI-machine (as in [Smi14])

```
while IN is not empty do
  if I ≠ ⊥ and t_I is the lowest element not yet in OUT then
    (1) push t_I to OUT
  else if all the m entries in D make up the m lowest entries not yet in OUT in decreasing order
  then
    (2) push from D to I m times in a row
  else if (t_D < t_IN or D = ⊥) and (t_IN < t_I or I = ⊥) then
    (3) push t_IN to D
  else
    (4) push t_D to I
  end if
end while
push everything from D to I
push everything from I to OUT
```

Algorithm 3 The completed DI-machine

while IN is not empty **do**
  if $I \neq \perp$ and t_I is the lowest element not yet in OUT **then**
    (1) push t_I to OUT
  else if all the m entries in D make up the m lowest entries not yet in OUT in decreasing order **then**
    (2) push from D to I m times in a row
  else if ($t_D < t_{IN}$ or $D = \perp$) and ($t_{IN} < t_I$ or $I = \perp$) **then**
    (3) push t_{IN} to D
  else if $D \neq \perp$ **then**
    (4) push t_D to I
  else
    (5) push t_I to OUT
  end if
end while
push everything from D to I
push everything from I to OUT

Algorithm 4 The studied DI-machine

while IN is not empty **do**
  if ($t_D < t_{IN}$ or $D = \perp$) and ($t_{IN} < t_I$ or $I = \perp$) **then**
    (D) push t_{IN} to D
  else if $D \neq \perp$ **then**
    (I) push t_D to I
  else
    (O) push t_I to OUT
  end if
end while
push everything from D to I
push everything from I to OUT

2 Scientific context

The first work we know about stack-sorting was done by Knuth in [Knu68] in 1968. His goal was to sort a permutation with a single stack. He showed that the results are best if you use what we call an increasing stack with the algorithm given in Algorithm 1 (it will be referred as the I-machine). Of course such a device that is working in linear time (considering that comparisons are done in constant time) cannot sort every permutation. Knuth found an interesting way to characterize the sortable permutations using permutation patterns: a permutation is sortable by Knuth's machine if and only if it avoids the pattern 231.This was the starting point of many other works that tried to describe more complex devices, like in [Tar72] with networks of stacks in series or in parallel, or even using other structures like queues.

Julian West was the first one to focus on the case of several increasing stacks in series in his PhD thesis in [Wes90]. There are several ways to work with more than one stack. Either you let them freely communicate or you wait than one device is done before you use the next one. In the first case you have more flexibility and you could potentially sort permutations, but the study of such a device could be hard as a result. In [AMR02] Atkinson, Murphy and Ruškuc proved that there is an infinite family of mutually independent patterns that a sortable permutation with a two-increasing-stack device with no restriction must avoid. This is indeed way more complicated than the case of the I-machine where you only have to avoid one pattern. Instead West took the other approach and made the increasing stacks work independently. We call a permutation that is sorted by such a machine with k increasing stacks "West-k-stack-sortable". What is interesting is that each increasing stack can be seen as a function $f_I : \mathfrak{S}_n \to \mathfrak{S}_n$. Thus the whole machine is simply iterating the I-machine k times, and the problem has reduced to studying f_I, a.k.a. the algorithm for a single iteration of the I-machine. Of course this approach is likely to sort less permutations with the same number of permutations since we allow less moves, but it could simplify the study of the device.

It was the case because a couple years later West characterized the West-2-stack-sortable permutations and conjectured an enumeration for them in [Wes93]:

Theorem 2.0.1 (characterization of the West-2-stack-sortable permutations)**.** *A permutation $\pi \in \mathfrak{S}_n$ is not West-2-stack-sortable iff it has a subpermutation of type 2341, or it has a subpermutation of type 3241 which is not part of a subpermutation of type 35241.*

Conjecture 2.0.1 (West's conjecture)**.** *The number of West-2-stack-sortable permutations of length n is $\frac{2(3n)!}{(n+1)!(2n+1)!}$.*

The characterization is simpler then what we had in [AMR02]. Plus in general enumerating is a harder problem that characterizing with permutation patterns, and still West's conjecture was rapidly proven by Zeilberger in [Zei92]. The proof was computer-assisted - like the proof of the Four-Color theorem - and it took several more years before combinatorial proofs were found, first by Dulucq, Gire and Guibert in [DGG98], then a refined one by Goulden and West in [GW96].

Even though the case of the West-2-stack-sortable permutations was solved quite rapidly, it took nearly two more decades to characterize the West-3-stack-sortable permutations with Úlfarsson in [Úlf12]. The characterization is way more complicated and no enumeration result has been found yet. We can only imagine how intricate the characterization of West-k-stack-sortable permutations would be with $k \geq 4$.

Facing such difficulties people explored other stack-sorting devices. Rebecca Smith defined the DI-machine in [Smi14], which is a decreasing stack followed by an increasing stack. The issue with

more complicated machines is that unlike with a single increasing stack where you rapidly find the only working algorithm, you have multiple possible algorithms here. Smith showed that her algorithm (given by Algorithm 2) was optimal for sorting permutations with the DI-machine. She also characterized the DI-sortable permutations:

Theorem 2.0.2 (Smith, 2014). *A permutation $\pi \in \mathfrak{S}_n$ is DI-sortable iff it avoids the patterns 3142 and 3241.*

Thus like the I-machine we have a stack-sorting device with a simple characterization of the sortable permutations in terms of permutation patterns.

The main problem with Smith's algorithm is that it ends in a deadlock if the permutation is not DI-sortable while it would be interesting to output something that could be worked with later. The goal of this study is to adapt Smith's algorithm so that it could be iterable, and then study the iterations of the DI-machine as West did with Knuth's I-machine.

3 My contribution

The first goal will be to give a new algorithm (Algorithm 4) that would do essentially the same thing as the algorithm Smith proposed (Algorithm 2), except that it makes the DI-machine iterable. This will be done in section 3.1. Then we will investigate the behavior of the DI-machine, especially which inversions are not treated by a single iteration. Finally we will consider what happens with multiple iterations.

3.1 Preliminary work

3.1.1 Monotony of the DI-machine

This subsection will show something fundamental about the DI-machine: at any moment, the elements in the machine (e.g. in D or I) are so that it is always possible to push from D to I. It is essentially what Lemma 3.1.1 tells us.

Lemma 3.1.1. *At every moment when using the DI-machine with Algorithm 2/3/4, if D and I are not empty, then* $t_I > t_D$.

Proof. We prove this by induction on the sequence of operations. It is true at the beginning since D and I are both empty. If now we are in any situation such that D and I are not empty and $t_I > t_D$:

- if we push from IN to D (case (3) or (D)): we know from the conditions that we had $t_{IN} < t_I$. Now that the former t_{IN} is t_D, we indeed have $t_I > t_D$.
- if we push from D to I (case (2) or (4) or (I)): the former t_D is now t_I and either D is empty or the new t_D is lower than the former t_D since D is a decreasing stack.
- if we push from I to OUT (case (1) or (5) or (O)): the former t_I was greater than t_D and is now in OUT, and now either I is empty or t_I is even greater since I is an increasing stack.

Hence the proposition is also valid at the next state. Hence it is valid at any state of the machine. □

This lemma guarantees that we would never end in a deadlock because we pushed something from D to I that violated that I was an increasing stack. It also allows us to prove the following property:

Property 3.1.1 (Monotony of the DI-machine)**.** *At every moment when using the DI-machine with Algorithm 2/3/4, if* $a, b \in [1..n]$ *are in the machine (e.g. somewhere in D or I), then:*
$a < b$ *iff:*
- either a *and* b *are both in D with* b *deeper than* a*,*
- or a *is in D and* b *is in I,*
- or a *and* b *are both in D with* a *deeper than* b*.*

Proof. The only non-trivial cases are when a and b are not in the same stack. If $a < b$, then b in D and a in I would violate Lemma 3.1.1. Conversely if a is in D and b is in I, we deduce from Lemma 3.1.1 that $a < b$. □

Remark 3.1.1. *The DI-machine with any algorithm is equivalent to an increasing stack with a pointer (pointing at would be the top of I in the DI-machine, and at the "left extremity" if* $I = \perp$*) and the following operations:*

- push from IN to just the right of the pointer
(same as a D operation: push from IN to D)
- move the pointer to the element just at the right, if there is one
(same as an I operation: push from D to I if $D \neq \bot$*)*
- if not pointing at the left extremity: pop the pointed element to OUT and point at the element that was just at the left, or at the left extremity if there is none
(same as an O operation: push from I to OUT if $I \neq \bot$*).*

3.1.2 Refining the algorithm

The algorithm presented in [Smi14], which is more or less Algorithm 2, has a significant imperfection: when it fails to sort the permutation, it fails and it does not return anything. If we want to be able to iterate, we will need another algorithm.

Remark 3.1.2. *The algorithm given in [Smi14] is actually slightly different from Algorithm 2. Once IN becomes empty, instead of pushing everything from D to I then everything from I to OUT, it continues to try to apply the loop of the four cases. But it is worth noticing that as long as D is not empty, only cases (2) and (4) are possible: case (3) would require* t_{IN} *while there is none since* $IN = \bot$ *, and case (1) would mean that* t_I *is the lowest value not in OUT, hence D should be empty by the monotony of the DI-machine, which is not. A sequence of cases (2) and (4) do exactly what was said: pushing everything from D to I. Once D is empty, since I is an I-stack,* t_I *is indeed the lowest value not in OUT. Hence there is a sequence of cases (1) until I is empty, which is exactly pushing everything from I to OUT. Hence Algorithm 2 is indeed faithful to the algorithm in [Smi14].*

from Algorithm 2 to Algorithm 3

We want to ensure that the algorithm we use never ends in a deadlock, which is not the case of Algorithm 2. We prove that Algorithm 3 guarantees it.

Lemma 3.1.2. *The DI-machine with Algorithm 3 never ends in a deadlock.*

Proof. We know from Lemma 3.1.1. that a deadlock could not be caused by contradicting that I is an increasing stack. Furthermore the condition of case (3) ensures that D is indeed a decreasing stack (the part "$t_D < t_{IN}$ or $D = \bot$"), and cases (1) and (2) check if the corresponding stacks are not empty.

Hence the only problem that could happen with Algorithm 2 would be to be in case (4) and try to push from D to I while D is empty. And it does happen, for example with $\pi = 3241$. It actually happens with every non DI-sortable permutation. In order to avoid this, we check in Algorithm 3 if D is not empty before doing case (4), and otherwise we know that both cases (3) and (4) were not done, which means that:
$D = \bot$ and $[(t_D > t_{IN}$ and $D \neq \bot)$ or $(t_{IN} > t_I$ and $I \neq \bot)]$,
hence: $D = \bot$ and $t_{IN} > t_I$ and $I \neq \bot$.
Hence $I \neq \bot$, which means that we are sure that case (5), which pushes from I to OUT, will never create a deadlock. □

Now that we proved that Algorithm 3 always terminates, it is easy to establish that Algorithm 3 is a completed version of Algorithm 2.

Result 3.1.1. *Let* $n \in [1..n]$*. Let* $\pi \in \mathfrak{S}_n$*. If DI with Algorithm 2 terminates with* (π) *as the input (there has been no deadlock), then:* $DI_{Algorithm2}(\pi) = DI_{Algorithm3}(\pi)$

Proof. see Appendices □

from Algorithm 3 to Algorithm 4

We now have an iterable version of Algorithm 2 with Algorithm 3. However it is not entirely satisfying because it can be shown that cases (1) and (2) are not necessary. This is how we get to Algorithm 4: case (D) corresponds to case (3), case (I) corresponds to case (4) and case (O) corresponds to case (5).
We first show that Algorithm 4 also always terminates.

Lemma 3.1.3. *The DI-machine with Algorithm 4 never ends in a deadlock.*

Proof. It is essentially the same proof as in Lemma 3.1.2 since we did not use cases (1) and (2) to prove that whenever we reach case (5)/(O), $I \neq \bot$ and thus there is no deadlock from case (5)/(O). □

We now get rid of cases (1) and (2).

Result 3.1.2. *Let* $n \in [1..n]$. *Let* $\pi \in \mathfrak{S}_n$. *Then* $DI_{Algorithm3}(\pi) = DI_{Algorithm4}(\pi)$.
More precisely , if we remove the I_X *and* O_X *of the elements involved in cases (2) while using Algorithm 3, then the sequence of operations of Algorithm 3 and Algorithm 4 are the same.*

Proof. see Appendices □

Taking together Result 3.1.1 and Result 3.1.2, we get what we wanted:

Theorem 3.1.1. *Let* $n \in [1..n]$. *Let* $\pi \in \mathfrak{S}_n$. *If* $DI_{Algorithm2}(\pi)$ *terminates (without any deadlock), then:* $DI_{Algorithm2}(\pi) = DI_{Algorithm4}(\pi)$.

Before we say goodbye to Algorithm 2/3, there is one last interesting thing to notice.

Remark 3.1.3. *Whenever case (2) is done by the DI-machine with Algorithm 2/3,* $m = 1$.

3.1.3 Describing the series of operations

For the remaining of the study, we will consider that we use Algorithm 4 with the DI-machine.
According to the "Definitions" section, there are three elementary actions for each element a in $[1..n]$. We will denote them:
- D_a: push a from t_{IN} to t_D
- I_a: push a from t_D to t_I
- O_a: push a from t_I to t_O.

You might now understand why it was interesting to switch from Algorithm 3 to Algorithm 4. The three cases exactly correspond to the elementary actions $D_{t_{IN}}$, I_{t_D} and O_{t_I}. Consequently we can link the three cases of Algorithm 4 to the corresponding elementary actions.

Result 3.1.3. *From any working state of the DI-machine:*
- $D_{t_{IN}}$ *is next iff (*$D = \bot$ *or* $t_D < t_{IN}$*) and (*$I = \bot$ *or* $t_{IN} < t_I$*)*
- I_{t_D} *is next iff* $D \neq \bot$ *and [(*$t_D > t_{IN}$*) or (*$I \neq \bot$ *and* $t_{IN} > t_I$*)]*
- O_{t_I} *is next iff* $D = \bot$ *and* $I \neq \bot$ *and* $t_{IN} > t_I$

In the proofs of some following results we will need to refer to the state of t_{IN}, t_D and t_I at multiple different states of the process.

Definition 3.1.1. *Let* X *be an elementary action of the DI-machine with Algorithm 4 and* π *of length* n *as the input (*$X = Y_a, Y \in \{D, I, O\}, a \in [1..n]$*).*
We define $t_I(X)$ *the top of the stack* S_i *right before the machine does the action* X *(only defined if* I *is not empty).*
We define similarly $t_D(X)$ *and* $t_{IN}(X)$.

We are now ready to investigate the interesting stuff.

3.2 A fundamental lemma

The key to sorting is to understand what happens to the inversions of the initial permutation. This is the goal of the following result:

Lemma 3.2.1 (Fundamental lemma of the DI-machine). *Let* $\{a, b\}$ *be an inversion of* $\pi \in \mathfrak{S}_n$. *Then:* $\{a, b\}$ *is an inversion of* $DI(\pi)$ *iff there exists* c, d, e *in* $[1..n]$ *such that:*
- $b \preceq_\pi d \prec_\pi e \prec_\pi a$
- $c \prec_\pi d$
- $b \preceq_\pi c$ *or* $c = t_I(D_b)$
- $d < c < e$
- $b < e$

In order to understand why I call it "fundamental", here is what would be the equivalent lemma for the I-machine:

Lemma 3.2.2 (Fundamental lemma of the I-machine). *Let* $\{a, b\}$ *be an inversion of* $\pi \in \mathfrak{S}_n$. *Then:* $\{a, b\}$ *is an inversion of* $I_{Algorithm1}(\pi)$ *iff there exists* c *in* $[1..n]$ *such that:*
- $b \prec_\pi c \prec_\pi a$
- $b < c$

This is where the 231 pattern comes from: if $I(\pi) \neq Id_n$, then there is an inversion a, b of $I(\pi)$ (say $a < b$). It can be proved that non-inversions remain non-inversions, thus by contra-position $\{a, b\}$ is also an inversion of π. Hence according to Lemma 3.2.2, there is a 231 pattern in π.

3.2.1 Preliminary results

About the sequence of operations

Lemma 3.2.3. *Let* $a, b \in [1..n]$.
Then:
- $a \prec_\pi b$ *iff* $D_a \prec_{DI,\pi} D_b$.
- $a \prec_{DI(\pi)} b$ *iff* $O_a \prec_{DI,\pi} O_b$.

Proof. It is a direct consequence of how we remove/put elements in IN/OUT. □

Lemma 3.2.4. $\forall a \in [1..n], D_a \prec_{DI,\pi} I_a \prec_{DI,\pi} O_a$.

Proof. It is a direct consequence of the restrictions we put on the operations of the machines we consider. □

Lemma 3.2.5. *If* $a, b \in [1..n]$ *such that* $a < b$ *and* $I_a \prec_{DI,\pi} D_b$, *then:* $O_a \prec_{DI,\pi} D_b$.

Proof. see Appendices □

Result 3.2.1 (Monotony of the O_a operations). *If at any moment* a *and* b *are in the machine (e.g. somewhere in D or I) and* $a < b$, *then:* $O_a \prec_{DI,\pi} O_b$.

Proof. Using Property 3.1.1:
- if a and b are in I, b deeper than a:
I is a stack, hence: $O_a \prec_{DI,\pi} O_b$.
- If b is in I and a is in D:
O_b requires D to be empty.
Hence: $I_a \prec_{DI,\pi} O_b$.
Then right after I_a we are in the first case with both elements in I, b deeper than a.
- If b and a are in D, a deeper than b:

D is a stack, hence: $I_b \prec_{DI,\pi} I_a$.
Then right after I_b we are in the previous case where b is in I and a is in D.

□

Result 3.2.2 (Preservation of the non-inversions). *If* $\{a, b\}$ *is a non-inversion of* π*, then it is a non-inversion of* $DI(\pi)$.

Proof. We know that $a \prec_\pi b$ and $a < b$.
Hence: $D_a \prec_{DI,\pi} D_b$ (Lemma 3.2.3.).
Hence right after D_b is done, a is in D or I or OUT.
- If a is then in OUT:
Then $O_a \prec_{DI,\pi} O_b$.
- If a is then in D or I: Then a and b are both in the machine with $a < b$.
Hence, according to Result 3.2.1: $O_a \prec_{DI,\pi} O_b$
-> In both cases: $O_a \prec_{DI,\pi} O_b$.
Hence $a \prec_{DI(\pi)} b$ (Lemma 3.2.3.).
Hence $\{a, b\}$ is a non-inversion of $DI(\pi)$. □

Result 3.2.3. *Let* $\{a, b\}$ *be an inversion of* π.
Then: $\{a, b\}$ *is an inversion of* $DI(\pi)$ *iff* $O_b \prec_{DI,\pi} D_a$.

Proof. see Appendices □

Characterizing the stack I at D_a operations

Definition 3.2.1. *For all* b *in* $[1..n]$*:* $\overline{Inv}_{DI,\pi}(b) \stackrel{def}{=} \{c \in [1..n] | c > b \wedge c \prec_\pi b \wedge b \prec_{DI(\pi)} c\}$.

In other words, $\overline{Inv}_{DI,\pi}(b)$ is the set of the elements c at the left of b in π such that $\{b, c\}$ is an inversion in π and a non-inversion in $DI(\pi)$.

We can then capture all the inversions that have been "solved" by the DI-machine by taking the union of these sets. However, there is something else interesting about them.

Property 3.2.1. *Let* $b \in [1..n]$. *Right before the elementary action* D_b *is done by DI, the elements in I are exactly* $\overline{Inv}_{DI,\pi}(b)$.

Proof. ($\subseteq$) Right before D_b is done, let c be an element of I. - Then: $D_c \prec_{DI,\pi} D_b$.
Hence: $c \prec_\pi b$ (Lemma 3.2.3).
- Just after D_b: b is in D and c is in I.
Hence, according to Property 3.1.1: $c > b$.
- From the previous points, we know that $\{b, c\}$ is an inversion of π.
But: $D_b \prec_{DI,\pi} O_c$.
Hence, according to Result 3.2.3, $\{b, c\}$ is a non-inversion of $DI(\pi)$.

Hence $c \in \overline{Inv}_{DI,\pi}(b)$.

($\supseteq$) If $c \in \overline{Inv}_{DI,\pi}(b)$:
- $c \prec_\pi b$.
Hence $D_c \prec_{DI,\pi} D_b$ (Lemma 3.2.3). - We also know that $c > b$. Hence $\{b, c\}$ is an inversion of π.
But we know that $\{b, c\}$ is a non-inversion of $DI(\pi)$. (definition of $\overline{Inv}_{DI,\pi}(b)$).
Hence, according to Result 3.2.3: $D_b \prec_{DI,\pi} O_c$.
Hence: $D_c \prec_{DI,\pi} D_b \prec_{DI,\pi} O_c$.
Hence right before D_b is done, c is in D or I.
By contradiction: if c is in D right before D_b is done,

Then right after D_b is done: b and c are in I with c deeper than b while $c > b$.
Contradiction with Property 3.1.1.
Hence c is in I right before D_b is done. □

Corollary 3.2.1. *Let* $b \in [1..n]$. *If* $I \neq \perp$ *right before the elementary action* D_b *is done by DI, then:* $t_I(D_b) = \min(\overline{Inv}_{DI,\pi}(b))$.

Proof. Direct application of Property 3.2.1 and recalling that I is an increasing stack. □

Corollary 3.2.2. *If* b *is an intermediate maximum, then* $I = \perp$ *right before the elementary action* D_b *is done by DI.*

Proof. If b is an intermediate maximum, then there is no c at such that $c \prec_\pi b$ and $\{b, c\}$ is an inversion of π.
Hence $\overline{Inv}_{DI,\pi}(b) = \varnothing$.
Hence, according to Property 3.2.1, $I = \perp$ right before D_b is done. □

Corollary 3.2.3. *If* b *is NOT an intermediate maximum, then* $I \neq \perp$ *right before the elementary action* D_b *is done by DI.*

Proof. If b is not an intermediate maximum, then there exists c such that $c \prec_\pi b$ and $c > b$.
Let $c_b \stackrel{def}{=} \text{rightmost}(\{c | c \prec_\pi b \wedge c > b\})$.
By contradiction: if $\{b, c_b\}$ is an inversion of $DI(\pi)$:
Then, knowing that $\{b, c_b\}$ is an inversion of π, according to Result 3.2.3.: $O_{C_b} \prec_{DI,\pi} D_b$.
Hence, according to Result 3.1.3, when O_{C_b} was done, there was d as t_{IN} such that $c_b \prec_\pi d$ and $d > c_b$.
But $d > c_b$ implies $d > b$, which means that d cannot be b and thus $d \prec_{pi} b$.
So d is an element at the right of c such that $d \prec_\pi b$ and $d > b$.
Contradiction with c_b being the rightmost element with such properties.
Hence: $\{b, c_b\}$ is a non-inversion of $DI(\pi)$.
Hence : $c_b \in \overline{Inv}_{DI,\pi}(b)$ and $\overline{Inv}_{DI,\pi}(b)) \neq \varnothing$.
Hence, according to Property 3.2.1, $I \neq \perp$ right before D_b is done. □

3.2.2 The proof (see Appendices)

The proof is actually quite long and I wanted to save these several pages to present meaningful results instead. You can get the proof in Appendices.

3.2.3 Some implications

Like we did at the beginning of this section with the I-machine, it is possible to prove again the characterization of DI-sortable permutations with the fundamental lemma.

Corollary 3.2.4 (Another proof of Theorem 2.0.2)**.** π *is not DI-sortable iff there is a 3142 or a 3241 pattern in* π.

With the fundamental lemma, we can also infer another interesting result:

Corollary 3.2.5. *If* b *is not an intermediate maximum, then* $t_I(D_b)$ *exists and:*
$\forall c$ *s.t.* $t_I(D_b) \prec_\pi c \prec_\pi b$, $c > t_I(D_b)$ *or* $c < b$.

In other words: in between $t_I(D_b)$ and b, there is no c such that: $b < c < t_I(D_b)$.

3.3 Iterating the DI-machine

We call DI-k-sortable a permutation that is sorted after k iterations of the DI-machine.

3.3.1 Maximum number of iterations

A first result

We could once again take inspiration from the case of the I-machine, where if $\pi = L\pi R$, then $I(\pi) = I(L)I(R)n$, and thus a permutation of size n would require at most $n-1$ iterations, which is indeed the maximum (take for example, with $n \geq 3, \pi_n = 234...n1$).
We could actually prove the following result, which is very similar:

Result 3.3.1. *Let* $\pi \in \mathfrak{S}_n$, $\pi = a_1...a_n$.
Let k *be such that* $a_k = n$.
Let $k_0 \stackrel{def}{=} \min(\{i | \forall i \leq j \leq k, a_j$ *is an intermediate maximum of* $\pi\})$ *(from* n *we expand to the left the series of intermediate maxima as much as possible).*
Let $M \stackrel{def}{=} a_{k_0}...a_{k-1}$ *(M could be empty).*
Let $\pi = La_{k_0}...a_kR$, *so:* $\pi = LMnR$ *(L and R could be empty).*
Then: $DI(\pi) = DI(L)DI(MR)n$.

We have then an upper bound of the required number of iterations.

Corollary 3.3.1. *If* $\pi \in \mathfrak{S}_n$, *then sorting* π *with the DI-machine requires at most* $n-2$ *iterations.*

Proof. Let $n \geq 5$. According to Result 3.3.1. the greatest is guaranteed to be put at the very right of the output permutation. But with a second iteration, we can apply Result 3.3.1. to the permutation of the $(n-1)$ lowest elements to guarantee that $(n-1)$ will be put at the right of the $(n-2)$ lowest values. But since non-inversions remain non-inversions according to Result 3.2.2., we will have $(n-1)n$ at the end after two iterations. If we keep going, after $(n-4)$ iterations, we will have $5...(n-1n)$ at the end of the output permutation. What remains is a subpermutation of length 4 that is shown to be DI-2-sortable by Theorem 2.0.2. □

A better result

The previous result is not really satisfying because it is already non-optimal for $n = 5$: no permutation of length 5 is not DI-2-sortable. What is happening is actually more complicated than just putting elements at the extremities. We need another tool in order to get a better upper bound. One way is to use the α I defined that counts the number of successions (peak-valley) when we read the permutation from left to right. We only pay attention to the ascents and descents.

The first thing to notice is that a 3241 or 3142 pattern requires to have at least the "pattern" descent-ascent-descent. But it is exactly what counts α along the permutation.

Lemma 3.3.1. *If* $\alpha(DI(\pi)) = 0$, *then* π *is DI-sortable.*

Proof. By contradiction: if π is not DI-sortable,
Then π has a 3241 or 3142 pattern according to Theorem 2.0.2.
But it would mean that you have at least descent, then an ascent, then a descent. Thus it would mean that you have at least one valley followed by a peak, thus $\alpha(DI(\pi)) > 0$.
Contradiction.
Hence π is DI-sortable. □

We can also easily give an upper bound of $\alpha(\pi)$:

Lemma 3.3.2. $\forall \pi \in \mathfrak{S}_n, \alpha(\pi) \leq \left\lfloor \frac{n}{2} \right\rfloor - 1$.

Proof. Valleys and peaks cannot be the left/right extremities and since every peak is potentially followed by a valley but not another peak and conversely, every peak/valley cannot be part of two couples that $\alpha(\pi)$ counts.
Hence there are at most $\left\lfloor \frac{n-2}{2} \right\rfloor = \left\lfloor \frac{n}{2} \right\rfloor - 1$ couples (valley, peak) to be counted by $\alpha(\pi)$. □

The only missing ingredient is now the recurrence relation which would guarantee that $\alpha(\pi)$ strictly decreases every time we iterate the DI-machine on the permutation.

Lemma 3.3.3. *If* $\alpha(\pi) \geq 1$*, then:* $\alpha(DI(\pi)) \leq \alpha(\pi) - 1$.

Proof. There are two key points in this proof.
- The first one is that the DI-machine throws elements to OUT only when we are in an ascent of π (or when IN is empty). In a descent, every element is always lower than t_D which used to be their predecessor in π. From the point that the former predecessor is t_I, you can infer from Result 3.1.3 that there will be no (O) operation as long as the element is t_{IN}. It is easy to show that an ascent in π can only to $DI(\pi)$ one ascent: once some elements were first thrown to OUT since the beginning of the ascent, they were going in increasing order according to Result 3.2.1., and all the remaining elements in the machine and in the ascent are greater the output elements. The same could be then said to any sequence of elements thrown to OUT during this ascent. Hence once no element of the ascent remains in IN, the contribution of this ascent to π is at most one ascent.
- The second one is that the beginning of the permutation is either a V shape (descent- ascent) or an N shape (ascent-descent-ascent) and in both cases what you obtain from the DI-machine at that point is always an ascent. Hence at the beginning there is one less valley that was part of a couple (valley, peak) counted by $\alpha(\pi)$.
- One last detail: if π terminates in a descent, then there is one more ascent at the end of $DI(\pi)$ but as $DI(\pi)$ terminates then in an ascent, this new ascent does not give a new couple (peak, valley) to make $\alpha(DI(\pi))$ go up.

Hence we know that α decreases by at least one. □

As a consequence, we have the following general result:

Result 3.3.2. *If* $\pi \in \mathfrak{S}_n$*, then sorting* π *with the DI-machine requires at most* $\left\lfloor \frac{n}{2} \right\rfloor$ *iterations.*

We can even compute $\alpha(\pi)$ in linear time and potentially get a better upper bound.

Result 3.3.3. *If* $\pi \in \mathfrak{S}_n$*, then* $\alpha(\pi) + 1$ *is an upper bound of the number of iterations required to sort* π *with the DI-machine.*

Even if this upper bound is better than the previous one, it is still not perfect. For example if we consider $\sigma_{2k} = (2k-1)(2k)(2k-3(2k-2)...3412)$, then we have $\alpha(\sigma_{2k}) = k-1$ while σ_{2k} is DI-sortable. It shows that the relative positions of the peaks is also significant.

However the upper is still very interesting as we know permutations of length n that require $\left\lfloor \frac{n}{2} \right\rfloor$ iterations.

Result 3.3.4. *Let* $k \geq 2$.
- *If* $n = 2k$*:*

If $\pi = (k+1)a_1(k+2)a_2...(2k)a_k$ *with* $a_1a_2...a_k \in \mathfrak{S}_k$*, then* π *is not DI-(k-1)-sortable and is DI-k-sortable.*
- *If* $n = 2k+1$*:*

If $\pi = (k+1)a_1(k+2)a_2...(2k)a_k(2k+1)$ *with* $a_1a_2...a_k \in \mathfrak{S}_k$*, then* π *is not DI-(k-1)-sortable and is DI-k-sortable.*

What happens with these permutations is that we have a lot of occurrences of the 3142/3241 patterns and we placed the half of higher values strategically so that we will keep a significant number of occurrences of these patterns even after several iterations of the DI-machine.
For example if $n = 2k$, if $\pi = (k+1)a_1(k+2)a_2...(2k)a_k$ with $a_1a_2...a_k \in \mathfrak{S}_k$,
then: $DI(\pi) = a_1(k+1)a_2(k+2)...(2k-2)a_k(2k)$
Hence all the lower values changed their place with the higher value just at their left. The more we iterate, the more the lower values will accumulate to the left. But it takes k iterations for a_k to be at the left of $(k+1)$ since we managed to keep 3142/3241 patterns all along.

To conclude, we now have determined what was the maximum number of required iterations of the DI-machine for a permutation of length n.

Theorem 3.3.1. *The maximum number of iterations required to sort a permutation of length n with the DI-machine is $\lfloor \frac{n}{2} \rfloor$.*

It would also be interesting to characterize the permutations that require such a number of permutations. I think that the main idea is to keep the half of the higher values in the same fashion than in the previous example. (If $n = 2k+1$ is odd, my opinion is that the $(2k+1)$th value could put anywhere, it should not matter).

Conjecture 3.3.1. *Let $k \geq 2$.*
Let π be of length n which is DI-$\lfloor \frac{n}{2} \rfloor$-sortable but not DI-$(\lfloor \frac{n}{2} \rfloor - 1)$-sortable,
- If $n = 2k$, then $\pi = (k+1)a_1(k+2)a_2...(2k)a_k$ with $a_1a_2...a_k \in \mathfrak{S}_k$.
- If $n = 2k+1$, then π is of the form $\_\, a_1 \_\, (k+2) \_\, a_2 \_ ... \_\, (2k) \_\, a_k \_\, (2k+1) \_$ with $a_1a_2...a_k \in \mathfrak{S}_k$ and $(k+1)$ inserted anywhere where there is an underscore.

3.3.2 Conjecture with two iterations

Last but not least, the case of the DI-2-sortable permutations appear to be harder than the case of the maximum number of iterations. I tried to use the fundamental lemma as West did with the I-machine for the West-2-stack-sortable permutations. The problem is that the possibilities become too numerous very rapidly so that the chances to tackle the question in a exhaustive way without making ay mistakes are thin. For now all the examples I have found are about the following description in terms of permutation patterns.

Conjecture 3.3.2. *If π has a subpermutation of type $4a_15a_26a_3$ with $a_1a_2a_3 \in \mathfrak{S}_3$ that is not part of a subpermutation of type $4a_1658a_27a_3$, then π is not DI-2-sortable.*

4 Conclusion

We now understand better how the DI-machine works and what can be done with it. We gave an algorithm equivalent to Smith's one and that could be iterated. We proved a lemma that characterized the inversions of a permutation π which remained inversions of $DI(\pi)$. We proved that a permutation of length n required at most $\lfloor \frac{n}{2} \rfloor$ iterations, and that it was the maximum number. We conjectured the set of all permutations of length n that require the maximum number of iterations of the DI-machine. We conjectured a sufficient condition that a permutation would not be DI-2-sortable. The basic tools are now available to tackle the remaining conjectures, especially a characterization for the set of non DI-2-sortable permutations. Finally, the case of a DIDI machine with potentially other algorithms is still completely open.

Bibliography

[Alb12] Michael Albert. Permlab: Software for permutation patterns. http://www.cs.otago.ac.nz/staffpriv/malbert/permlab.php, 2012.

[AMR02] M.D. Atkinson, M.M. Murphy, and N. Ruškuc. Sorting with two ordered stacks in series. *Theoretical Computer Science*, 289(1):205–223, 2002.

[Bón03] Miklós Bóna. A survey of stack-sorting disciplines. *The Electronic Journal of Combinatorics*, 9(2), 2003.

[CF18] Giulio Cerbai and Luca Ferrari. Stack sorting with increasing and decreasing stacks. Dipartimento di Matematica e Informatica, University of Firenze, Italy, (unpublished) 2018.

[DGG98] S. Dulucq, S. Gire, and O. Guibert. A combinatorial proof of j. west's conjecture. *Discrete Mathematics*, 187:71–96, 1998.

[GW96] I. P. Goulden and J. West. Raney paths and a combinatorial relationship between rooted nonseparable planar maps and two-stack-sortable permutations. *Journal of Combinatorial Theory*, 75:220–242, 1996.

[Knu68] D.E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, Reading, MA, 1968.

[Smi14] Rebecca Smith. Two stacks in series: A decreasing stack followed by an increasing stack. *Springer Basel*, 18:359, 2014.

[Tar72] R. E. Tarjan. Sorting using networks of queues and stacks. *Journal of the ACM*, 19:341–346, 1972.

[Úlf12] Henning Úlfarsson. Describing west-3-stack-sortable permutations with permutation patterns. In *Séminaire Lotharingien de Combinatoire*, number 67, 2012.

[Wes90] Julian West. *Permutations with forbidden subsequences; and, Stack-sortable permutations*. PhD thesis, Massachusetts Institute of Technology, 63-66, 1990.

[Wes93] Julian West. Sorting twice through a stack. *Theoretical Computer Science*, 117(1-2):303–313, 1993.

[Zei92] Doron Zeilberger. A proof of julian west's conjecture that the number of two-stacksortable permutations of length n is 2(3n)!l((n + 1)!(2n + l)!). *Discrete Mathematics*, 102:85–93, 1992.

Appendices

Proof of Result 3.1.1.

Algorithm 2 and Algorithm 3 are literally the same thing for the first three cases. As we terminate with Algorithm 2, we always have D $\neq \perp$ whenever we are in case (4) with Algorithm 2. Hence Algorithm 3 is also in case (4) and will never be in case (5). Hence Algorithm 2 and Algorithm 3 are doing the same operations from start to end, so the results are the same.

Proof of Result 3.1.2.

- Case (1) is the easier to get rid of. We prove that whenever we are in the same machine state with both algorithms and we are in case (1) with Algorithm 3, we are in case (O) with Algorithm 4.
If we are in case (1) with Algorithm 3, then it means that t_I is the lowest element not yet in OUT.
Hence we have: I $\neq \perp$ and $t_I < t_{IN}$.
Hence we are not in case (D) with Algorithm 4.
Moreover, according to Property 3.1.1., D has to be empty.
Hence we are not in case (I) with Algorithm 4.
Hence we are in case (O) with Algorithm 4, and both algorithms will do the same thing next: push from I to OUT.

- Case (2) is trickier because we don't have the same sequence of elementary actions. However we show that if we ignore the "push from D to I" and "push from I to OUT" of the elements concerned by case (2) with Algorithm 3, then we get the same sequence of operations. In other words, these operations are not done at the same moment but it does not impact the other decisions.
We prove by induction on the sequence of actions that:
from the point that a case (2) happens with Algorithm 3, while IN is not empty or [I $= \perp$ or (I $\neq \perp$ and $t_I > t_{IN}$) or (t_D is not the top of the potential decreasing sequence of the lowest elements not in OUT with Algorithm 4)]],
the elementary actions are the same except for the "D to I" and 'I to OUT" ones of the elements concerned by case (2),
and that whenever the while condition is not fulfilled, Algorithm 4 will be eventually back to the exact same state as with Algorithm 4 - same IN, D, I, OUT.

Initially: as long as there is no case (2), both algorithms are doing the same thing (they have literally the same cases and conditions).

When there is a case (2) with Algorithm 3, we just ignore it with Algorithm 4 for now.
As long as the top of D is not the greatest element of the decreasing sequence of lowest elements not in OUT with Algorithm 4, all these elements at the bottom of D are isolated and do not impact the elementary action done by Algorithm 4 (only t_D matters). Moreover D is then not empty with Algorithm 3, thus no element other than the ones from the cases (2) are pushed to OUT.
If we get several cases (2) with Algorithm 3 without breaking the while condition, then they will directly go to OUT while with Algorithm 4 they will stack with the previous cases (2) and extend the decreasing sequence of lowest elements not yet in OUT (we know it will extend it because as soon as we have at least one case (2) with Algorithm 3, D is not empty with Algorithm 4 and thus there is no (O) case: nothing goes out of the machine with Algorithm 4.)
When the top of D is the top of the decreasing sequence of lowest elements not in OUT with Algorithm 4:
- If we have a case (2) with Algorithm 3:
then it would mean that D were not empty with Algorithm 3, which cannot be because D is empty with Algorithm 3 if and only if the top of D is the top of the decreasing sequence of lowest elements not in OUT with Algorithm 4.

- If we have a case (3) with Algorithm 3:
then $t_I > t_{IN}$ (same context in I with Algorithm 3 and Algorithm 4).
Plus $D \neq \perp$ with Algorithm 4 and we know that $t_D < t_{IN}$ since all the elements not yet in OUT lower than t_D are in D.
Hence we have a case (D) with Algorithm 4, which corresponds to the same elementary action.
(And the top of the decreasing sequence is no more the top of D with Algorithm 4.)
- If we have a case (4) with Algorithm 3:
impossible, same reason as with case (2).
- If we have a case (5) with Algorithm 3:
then it means that $t_I > t_{IN}$ with both algorithms (same content in I according the induction hypothesis).
We also know that then $I \neq \perp$ and thus the while condition is not fulfilled.

When the while condition is broken:
- if IN is empty:
then everything from D is pushed to I, and everything from I is pushed to OUT with both algorithms, but with Algorithm 4 we first push to OUT all the elements that were pushed in OUT earlier by the cases (2) with Algorithm 3.
- Otherwise: we know that I is not empty and $t_I > t_{IN}$ with both algorithms and t_D is the the top of the decreasing sequence of lowest elements not in OUT with Algorithm 4.
We know that t_I will eventually get pushed to OUT with both algorithms, but first Algorithm 4 pushes to I, then to OUT all the low elements that were pushed in OUT by cases (2) with Algorithm 3.
And we are finally back to the same global state (IN, D, I, OUT) right before pushing t_I to OUT!

Proof of Lemma 3.2.5

By contradiction: if we had $D_b \prec_{DI,\pi} O_a$,
We know that $I_a \prec_{DI,\pi} D_b$.
Hence: $I_a \prec_{DI,\pi} D_b \prec_{DI,\pi} O_a$.
Hence, just after D_b is done, a is in I and b is in D while $a < b$.
Contradiction with Property 3.1.1.

Proof of Result 3.2.3

Let $\{a, b\}$ be an inversion of π.
(=>) If $\{a, b\}$ is an inversion of $DI(\pi)$:
By contradiction: if $D_a \prec_{DI,\pi} O_b$,
Then we also know "$\{a, b\}$ be an inversion of π" that $b \prec_\pi a$, hence $D_b \prec_{DI,\pi} D_a$ (Lemma 3.2.3).
Hence: $D_b \prec_{DI,\pi} D_a \prec_{DI,\pi} O_b$.
Hence right after D_a is done, a and b are in the machine (with $b > a$).
Hence, according to Result 3.2.1: $O_a \prec_{DI,\pi} O_b$.
Hence $a \prec_{DI(\pi)} b$ (Lemma 3.2.3).
Contradiction with "$\{a, b\}$ is an inversion of $DI(\pi)$".
(<=) If $O_b \prec_{DI,\pi} D_a$:
We know from Lemma 3.2.4 that: $D_a \prec_{DI,\pi} O_a$.
Hence: $O_b \prec_{DI,\pi} O_a$.
Hence: $b \prec_{DI(\pi)} a$ (Lemma 3.2.3) while $a < b$.
Hence $\{a, b\}$ is an inversion of $DI(\pi)$.

Proof of Lemma 3.2.1. (Fundamental lemma)

The direct way

if $\{a, b\}$ is an inversion of π which is also an inversion of $DI(\pi)$:
We look at the operations done before O_b. We especially look for the rightmost D_χ action at the left of O_b (it exists since we have at least $D_b \prec_{DI,\pi} O_b$ according to Lemma 3.2.4).
Let α be the element such that D_α is the rightmost D_χ action at the left of O_b.
Let β be its successor in π.

*By contradiction: if we had $D_\beta \prec_{DI,\pi} O_b$:
Then we recall that $\alpha \prec_\pi \beta$, and thus $D_\alpha \prec_{DI,\pi} D_\beta$ (Lemma 3.2.3.).
Hence: $D_\alpha \prec_{DI,\pi} D_\beta \prec_{DI,\pi} O_b$.
Contradiction with the definition of α.
Hence: $O_b \prec_{DI,\pi} D_\beta$.
*We know that $O_b \prec_{DI,\pi} D_\beta$.
Hence: $D_\alpha \prec_{DI,\pi} O_b \prec_{DI,\pi} D_\beta$.
Hence: $\beta = t_{IN}(O_b)$.
Hence, according to Result 3.1.3.: $b < \beta$.

*By contradiction: if $D_a \prec_{DI,\pi} D_\beta$:
then, according to Lemma 3.2.3: $a \prec_\pi \beta$.
Hence: $a \preceq_\pi \alpha$.
Hence, according to Lemma 3.2.3: $D_a \preceq_{DI,\pi} D_\alpha$.
Hence a is in the machine or in OUT right after D_α is done.
However, by the definition of α, b is in the machine right after D_α is done.
Hence right after D_α is done:
- either b is the machine and a is in OUT: $O_a \prec_{DI,\pi} O_b$.
- or a and b are both in the machine with $a < b$ Contradiction by Lemma 3.2.3. that $\{a, b\}$ is an inversion of $DI(\pi)$.
Hence $D_\beta \prec_{DI,\pi} D_a$.
Hence $\beta \prec_\pi a$.
(Hence β is a good candidate for e in the lemma).

*By contradiction: if α were an intermediate maximum:
Then, according to Corollary 3.2.2.: $I = \perp$ right before D_α.
- If $\beta > \alpha$:
Then we would have D_b directly done.
Hence: $D_\beta \prec_{DI,\pi} O_b$.
- If $\beta < \alpha$:
Then α would go into I and become t_I, and then we know from Result 3.1.3 that there is no (O) operation until D_β is done.
-> Hence: $D_\beta \prec_{DI,\pi} O_b$ in both cases.
We already proved that it leads to a contradiction.
Hence α is not an intermediate maximum.
Hence, according to Corollary 3.2.3., $I \neq \perp$ right before D_α is done.
Let c_0 be the top of I just before D_α is done ($c_0 \stackrel{def}{=} t_I(D_\alpha)$).
Then c_0 is also the top of I just after D_α is done.
By contradiction: if $c_0 > \beta$:
Then as long as $t_{IN} = \beta$, (O) operations are not possible according to Result 3.1.3.
Hence β would get in D before b goes out of the machine:: $D_\beta \prec_{DI,\pi} O_b$.

We already proved that it leads to a contradiction.
Hence $c_0 < \beta$.

- If $b \preceq_\pi c_0$:
We propose $(c, d, e) = (c_0, \alpha, \beta)$.

- $c_0 = t_I(D_\alpha)$, hence $c_0 \prec_\pi \alpha$
- $b \preceq_\pi \alpha \prec_\pi \beta \prec_\pi a$
- $b \preceq_\pi c_0$
- We know that $c_0 < \beta$, and as $c_0 = t_I(D_\alpha)$, we know from Result 3.1.3 that $\alpha < c_0$
- $b < \beta$

- If $c_0 \prec_\pi b$:
— If $c_0 < b$:
We propose $(c, d, e) = (b, \alpha, \beta)$.

- by definition of α: $D_b \prec_{DI,\pi} D_\alpha$, hence: $b \prec_\pi \alpha$ (Lemma 3.2.3)
- $b \preceq_\pi \alpha \prec_\pi \beta \prec_\pi a$
- $b \preceq_\pi b$
- We know that $\alpha(< c_0) < b < \beta$
- $b < \beta$

— If $c_0 > b$:
We show that $c_0 = t_I(D_b)$.
*We first show that $c_0 \in \overline{Inv}_{DI,\pi}(b)$.
We already from the hypotheses that $c_0 \prec_\pi b$ and $c_0 > b$.
Hence we must only show that $\{b, c_0\}$ is a non-inversion of $DI(\pi)$.
Just after D_α was done, we know that c_0 is in I ($c_0 = t_I(D_\alpha)$) and that b is in D or I ($D_b \prec_{DI,\pi} D_\alpha \prec_{DI,\pi} O_b$).
We know that $c_0 > b$.
Then, according to Result 3.2.1., $O_b \prec_{DI,\pi} O_{c_0}$.
Hence: $b \prec_\pi c_0$ (Lemma 3.2.3).
Hence, because $b < c_0$, $\{b, c_0\}$ is a non-inversion of $DI(\pi)$.
Hence $c_0 \in \overline{Inv}_{DI,\pi}(b)$.
*By contradiction: if $c_0 \neq t_I(D_b)$:
$c_0 \in \overline{Inv}_{DI,\pi}(b)$, hence according to Property 3.2.1, just after D_b, c_0 is in I not at the top, and b is in D.
However $c_0 = t_I(D_\alpha)$, hence c_0 must be at the top of I right before D_α is done.
I being a stack, it means that: $O_{t_I(D_b)} \prec_{DI,\pi} D_\alpha$.
But right after D_b was done, b was in D, $t_I(D_b)$ was in I and we know from D_b and Result 3.1.3 that: $b < t_I(D_b)$.
Hence, according to Result 3.2.1.: $O_b \prec_\pi O_{t_I(D_b)}$.
Hence: $O_b \prec_{DI,\pi} D_\alpha$.
Contradiction with the definition of α.
Hence: $c_0 = t_I(D_b)$.
Hence we propose $(c, d, e) = (c_0, b, \beta)$.

- by definition of $t_I(D_b)$: $c_0 \prec_\pi b$

- $b \preceq_\pi b \prec_\pi \beta \prec_\pi a$
- $c_0 = t_I(D_b)$
- We proved that $c_0 < \beta$, and $b < c_0$ because of $c_0 = t_I(D_b)$ and Result 3.1.3. applied right before D_b
- $b < \beta$

Done.

The reciprocal

It is essentially about, for both cases $c = t_I(D_b)$ and $b \preceq_\pi c$, proving that $O_b \prec_{DI,\pi} D_e$ and that $D_e \prec_{DI,\pi} D_a$, then applying Result 3.2.3 to deduce that $\{a, b\}$ is an inversion of $DI(\pi)$.