Programming in python: permutations and trees

Mathilde Bouvel and Valentin Féray

Outline

1 Presentation of the project

2 Definitions of permutations and trees

3 `frec`: A recursive function on permutations

4 `stack`: Processing a permutation with a stack

5 `bintree`: Building a (binary) tree from a permutation

6 `inorder` and `postorder`: Building permutations from a tree

7 Testing the functions, and stating conjectures

8 Further questions

The purpose of the project is to :

- define in python some basic combinatorial objects: permutations, trees;
- implement some combinatorial maps between them;
- run some examples and discover experimentally some theorems.

Definition of permutations

A permutation is a list of positive integers with distinct entries.

Examples and counter-examples

- $[2, 4, 3, 7, 5]$ and $[1, 3, 4, 6, 2, 5]$ are permutations;
- $[3, -1, 2]$ and $[.5, 2, 1]$ are not permutations because one of their entry is not a positive integer;
- $[6, 1, 3, 4, 3, 5]$ is not a permutation because there is a repetition.

Definition of permutations

A permutation is a list of positive integers with distinct entries.

Examples and counter-examples

- $[2, 4, 3, 7, 5]$ and $[1, 3, 4, 6, 2, 5]$ are permutations;
- $[3, -1, 2]$ and $[.5, 2, 1]$ are not permutations because one of their entry is not a positive integer;
- $[6, 1, 3, 4, 3, 5]$ is not a permutation because there is a repetition.

Task 1

Implement a function `IsPermutation` which takes some object and tests whether it is a permutation or not.

Recursive definition of trees

A (binary) tree is either the empty tree (represented by the python object `None`) or a triple (a, T_1, T_2), where a is a positive integer and T_1 and T_2 are trees.

Examples and counter-examples

$(2, (3, (5, \texttt{None}, (7, \texttt{None}, \texttt{None}), \texttt{None}), (1, \texttt{None}, (6, \texttt{None}, \texttt{None})))$ is a tree;

But $(2, (3, 0, \texttt{None}), (1, \texttt{None}, (6, \texttt{None}, \texttt{None})))$ is not a tree because 0 is not a tree and hence $(3, 0, \texttt{None})$ is not a tree.

Recursive definition of trees

A (binary) tree is either the empty tree (represented by the python object `None`) or a triple (a, T_1, T_2), where a is a positive integer and T_1 and T_2 are trees.

Examples and counter-examples

$(2, (3, (5, \texttt{None}, (7, \texttt{None}, \texttt{None}), \texttt{None}), (1, \texttt{None}, (6, \texttt{None}, \texttt{None})))$ is a tree;

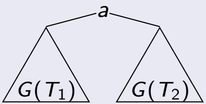
But $(2, (3, 0, \texttt{None}), (1, \texttt{None}, (6, \texttt{None}, \texttt{None})))$ is not a tree because 0 is not a tree and hence $(3, 0, \texttt{None})$ is not a tree.

Task 2

Implement a function `IsTree` which takes some object and tests whether it is a tree or not.

Graphical representation of a tree

The graphical representation $G(T)$ of a non-empty tree (a, T_1, T_2) is defined as follows



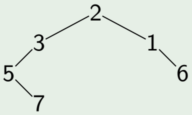
If one (or both) T_i is empty we erase the corresponding $G(T_i)$ and the corresponding edge.

Example of graphical representation of a tree

The graphical representation of

$$(2, (3, (5, \texttt{None}, (7, \texttt{None}, \texttt{None})), \texttt{None}), (1, \texttt{None}, (6, \texttt{None}, \texttt{None})))$$

is



Well-labelled trees

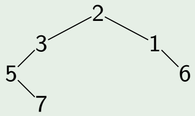
The label set $\mathrm{Lab}(T)$ of a tree is defined as:

- the empty set $\emptyset$ if T is the empty tree `None` ;
- the set $\{a\} \cup \mathrm{Lab}(T_1) \cup \mathrm{Lab}(T_2)$ if $T = (a, T_1, T_2)$.

A tree is called well-labelled if it is empty or is $T = (a, T_1, T_2)$ with T_1 and T_2 well-labelled and with $\{a\}$, $\mathrm{Lab}(T_1)$, $\mathrm{Lab}(T_2)$ pairwise disjoint.

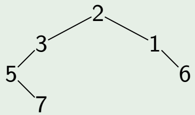
Intuitively, it means, that all labels on the graphical representation of the trees are distinct.

Example of a well-labelled tree



The tree here opposite is a well-labelled tree with label set $\{1, 2, 3, 5, 6, 7\}$;

Example of a well-labelled tree



The tree here opposite is a well-labelled tree with label set $\{1, 2, 3, 5, 6, 7\}$;

Task 3

Write a function `IsWellLabelled` which decides whether a tree given as input is well-labelled or not.

We will now define five combinatorial maps between permutations and trees. Your tasks will be to implement them.

- `frec` and `stack` take as input a permutation and return another permutation.
  The first one is recursive, while the second one is iterative.
- `bintree` is a recursive function, which takes as input a permutation and returns a well-labelled tree.
- `inorder` and `postorder` are also recursive functions, but take as input a well-labelled tree and return a permutation.

All these tasks are independent.

The function frec

frec is a recursive function on permutations defined by:

- $\mathtt{frec}([]) = []$
- if n is the maximum of a permutation P, writing $P = L \cdot [n] \cdot R$, we have: $\mathtt{frec}(L \cdot n \cdot R) = \mathtt{frec}(L) \cdot \mathtt{frec}(R) \cdot [n]$.

Here, $[]$ is the empty list and $\cdot$ is the concatenation of lists.

Example

$\mathtt{frec}([5,2,7,1]) = \mathtt{frec}([5,2]) \cdot \mathtt{frec}([1]) \cdot 7 = \cdots = [2,5,1,7]$.

The function frec

frec is a recursive function on permutations defined by:

- $\texttt{frec}([]) = []$
- if n is the maximum of a permutation P, writing $P = L \cdot [n] \cdot R$, we have: $\texttt{frec}(L \cdot n \cdot R) = \texttt{frec}(L) \cdot \texttt{frec}(R) \cdot [n]$.

Here, $[]$ is the empty list and $\cdot$ is the concatenation of lists.

Example

$\texttt{frec}([5, 2, 7, 1]) = \texttt{frec}([5, 2]) \cdot \texttt{frec}([1]) \cdot 7 = \cdots = [2, 5, 1, 7]$.

Task 4

Write a program that takes a permutation P as input, and returns as output $\texttt{frec}(P)$.

The function stack

stack is a function on permutations defined iteratively as follows:

(a) Take a permutation P as input.

(b) Create an empty stack S, and an empty list Q.

Comment: At the end of the procedure, Q will be the permutation stack(P).

(c) While P is not empty:

- (d) Consider the first element h of P.
- (e) While S is not empty and h is larger than the top element of S:
  - (f) Pop the top element of S to the end of Q.
- (g) Push h at the top of the stack S.

(h) While S is not empty:

- (i) Pop the top element of S to the end of Q.

(j) Output Q.

Step by step behavior of stack on the input $[6, 1, 3, 2, 7, 5, 4]$

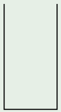
Step by step behavior of stack on the input $[6, 1, 3, 2, 7, 5, 4]$

$[6, 1, 3, 2, 7, 5, 4] = P$

This action correspond to step(s) (a) of the algorithm.

Step by step behavior of stack on the input $[6, 1, 3, 2, 7, 5, 4]$

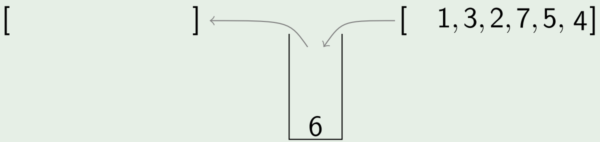
$Q = [\quad]$ $[6, 1, 3, 2, 7, 5, 4] = P$



S

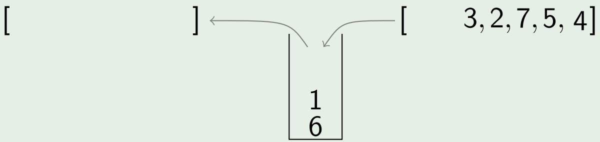
This action correspond to step(s) (b) of the algorithm.

Step by step behavior of stack on the input $[6, 1, 3, 2, 7, 5, 4]$



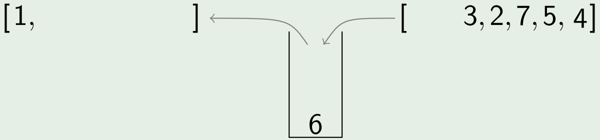
This action correspond to step(s) (d), (g) of the algorithm.

Step by step behavior of stack on the input $[6, 1, 3, 2, 7, 5, 4]$



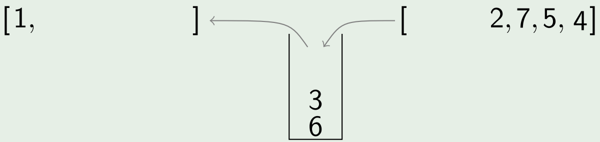
This action correspond to step(s) (d), (g) of the algorithm.

Step by step behavior of stack on the input [6, 1, 3, 2, 7, 5, 4]



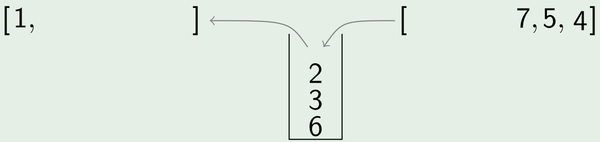
This action correspond to step(s) (d), (e), (f) of the algorithm.

Step by step behavior of stack on the input [6, 1, 3, 2, 7, 5, 4]



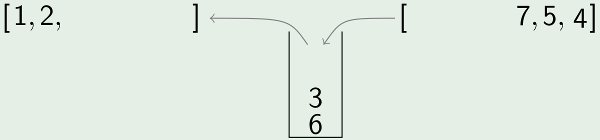
This action correspond to step(s) (g)
of the algorithm.

Step by step behavior of stack on the input [6, 1, 3, 2, 7, 5, 4]



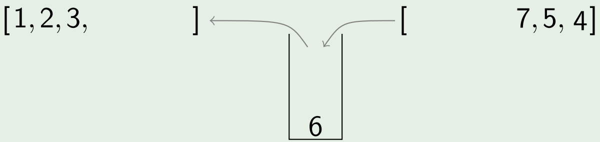
This action correspond to step(s) (d), (g) of the algorithm.

Step by step behavior of stack on the input $[6, 1, 3, 2, 7, 5, 4]$



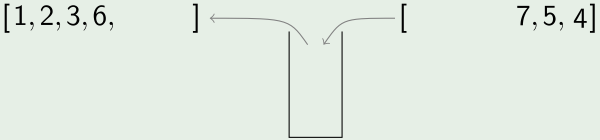
This action correspond to step(s) (d), (e), (f) of the algorithm.

Step by step behavior of stack on the input $[6,1,3,2,7,5,4]$



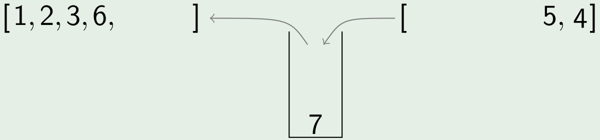
This action correspond to step(s) (e), (f) of the algorithm.

Step by step behavior of stack on the input $[6, 1, 3, 2, 7, 5, 4]$



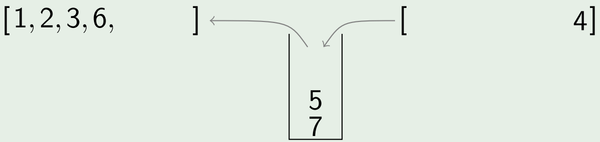
This action correspond to step(s) (e), (f) of the algorithm.

Step by step behavior of stack on the input $[6,1,3,2,7,5,4]$



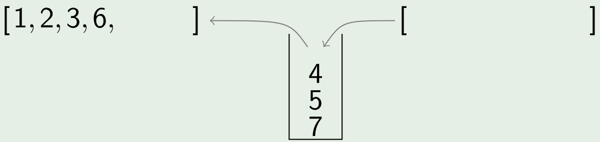
This action correspond to step(s) (g) of the algorithm.

Step by step behavior of stack on the input [6, 1, 3, 2, 7, 5, 4]



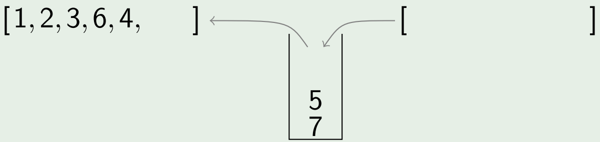
This action correspond to step(s) (d), (g) of the algorithm.

Step by step behavior of stack on the input $[6, 1, 3, 2, 7, 5, 4]$



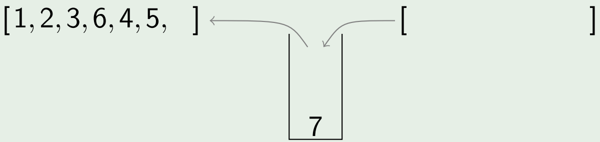
This action correspond to step(s) (d), (g)
of the algorithm.

Step by step behavior of stack on the input $[6, 1, 3, 2, 7, 5, 4]$



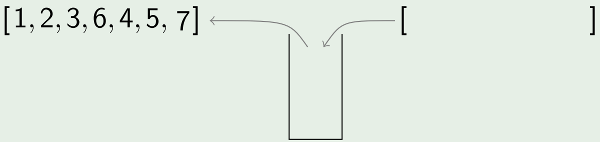
This action correspond to step(s) (h), (i) of the algorithm.

Step by step behavior of stack on the input $[6, 1, 3, 2, 7, 5, 4]$



This action correspond to step(s) (i)
of the algorithm.

Step by step behavior of stack on the input $[6, 1, 3, 2, 7, 5, 4]$



This action correspond to step(s) (i)
of the algorithm.

Step by step behavior of stack on the input $[6, 1, 3, 2, 7, 5, 4]$

Output $[1, 2, 3, 6, 4, 5, 7]$

This action correspond to step(s) (j)
of the algorithm.

Step by step behavior of stack on the input $[6, 1, 3, 2, 7, 5, 4]$

Output $[1, 2, 3, 6, 4, 5, 7]$

This action correspond to step(s) (j)
of the algorithm.

Task 5

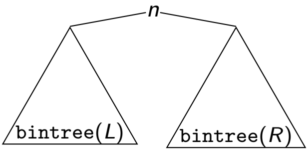
Write a program that takes a permutation P as input, and returns as output $\texttt{stack}(P)$.

The function bintree

bintree (standing for *binary tree*) is a function which associate a tree to a permutation. It is recursively defined by:

- $\texttt{bintree}([]) = \texttt{None}$
- if n is the maximum of a permutation P, writing $P = L \cdot n \cdot R$, we have: $\texttt{bintree}(L \cdot n \cdot R) = (n, \texttt{bintree}(L), \texttt{bintree}(R))$.

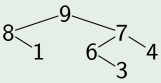
With the graphical representation of trees explained earlier, this means that

$\texttt{bintree}(L \cdot n \cdot R) =$ 

Example

For the permutation $P = [8, 1, 9, 6, 3, 7, 4]$, we have

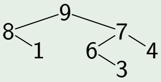
$$\texttt{bintree}(P) = (9, \texttt{bintree}([8,1]), \texttt{bintree}([6,3,7,4])) = \dots$$
$$= (9, (8, \texttt{None}, (1, \texttt{None}, \texttt{None})),$$
$$(7, (6, \texttt{None}, (3, \texttt{None}, \texttt{None})), (4, \texttt{None}, \texttt{None}))).$$

Graphically, this means that $\texttt{bintree}(P) =$  .

Example

For the permutation $P = [8, 1, 9, 6, 3, 7, 4]$, we have

$$\texttt{bintree}(P) = (9, \texttt{bintree}([8, 1]), \texttt{bintree}([6, 3, 7, 4])) = \dots$$
$$= (9, (8, \texttt{None}, (1, \texttt{None}, \texttt{None})), (7, (6, \texttt{None}, (3, \texttt{None}, \texttt{None})), (4, \texttt{None}, \texttt{None}))).$$

Graphically, this means that bintree(P) = 

.

Task 6

Write a program that takes a permutation P as input, and returns the tree bintree(P) as output.

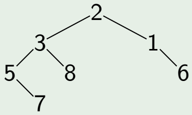
Definition of inorder and postorder

inorder and postorder are recursive functions, which associate a permutation to a well-labelled tree. They are defined as follows:

- $\texttt{inorder}(\texttt{None}) = \texttt{postorder}(\texttt{None}) = []$
- $\texttt{inorder}\big((a, T_1, T_2)\big) = \texttt{inorder}(T_1) \cdot [a] \cdot \texttt{inorder}(T_2)$
- $\texttt{postorder}\big((a, T_1, T_2)\big) = \texttt{postorder}(T_1) \cdot \texttt{postorder}(T_2) \cdot [a]$

Example of inorder and postorder

Let T be the tree below.

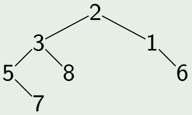


Then

$$\texttt{inorder}(T) = [5,7,3,8,2,1,6]$$
$$\texttt{postorder}(T) = [7,5,8,3,6,1,2]$$

Example of inorder and postorder

Let T be the tree below.



Then

$$\texttt{inorder}(T) = [5, 7, 3, 8, 2, 1, 6]$$
$$\texttt{postorder}(T) = [7, 5, 8, 3, 6, 1, 2]$$

Task 7

Write two programs that both take a well-labelled tree T as input, and return respectively the permutations $\texttt{inorder}(T)$ and $\texttt{postorder}(T)$ as output.

Task 8

Take several examples of permutations P (neither too short nor too easy) and do:

- compute $\texttt{inorder}\big(\texttt{bintree}(P)\big)$ on these examples. What do you observe? Formulate a conjecture.
- compute $\texttt{frec}(P)$, $\texttt{stack}(P)$ and $\texttt{postorder}\big(\texttt{bintree}(P)\big)$ on these examples. What do you observe? Formulate a second conjecture.

- So far, you have only tested your conjectures on a few examples chosen arbitrarily. Write a program which takes n as input and generates all permutations with entries $1, 2, \ldots, n$ and test your conjectures on all permutations for n as big as possible ($n = 10$ can already take a long time on a standard computer!).
- Experimenting is good to get an intuition, but we have not proven anything so far. Can you prove your conjectures?
- Explain how `inorder`(T) and `postorder`(T) can be read directly on the graphical representation of the tree T.
- Try and draw the trees using sage (algebra software based on python). Run sage on www.sagenb.org or install it on your computer and look at the documentation of DiGraph by typing `sage: DiGraph?`