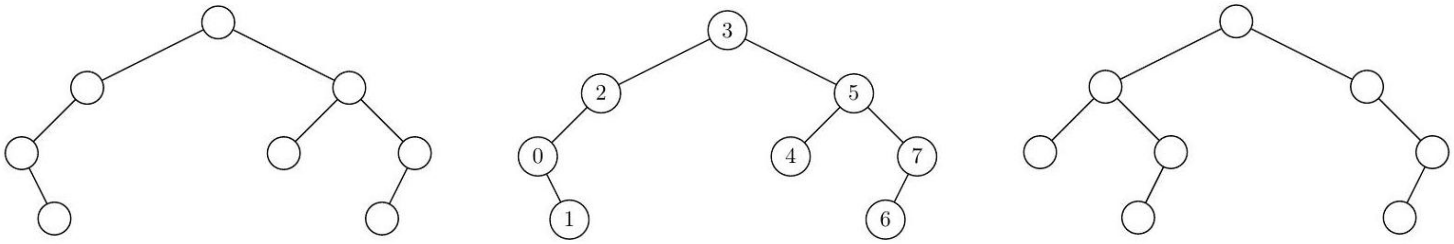
Exercice 1. Permutation triable avec une pile

Dans cet exercice on s'interdit d'utiliser les traits impératifs du langage OCaml (références, tableaux, champs mutables, etc.).

On représente en OCaml une permutation σ de $[\![0, n-1]\!]$ par la liste d'entier $[\sigma_0; \sigma_1; \ldots; \sigma_{n-1}]$. Un arbre binaire étiqueté est soit un arbre vide, soit un nœud formé d'un sous-arbre gauche, d'une étiquette et d'un sous-arbre droit :

```
type arbre = V | N of arbre * int * arbre
```

On représente un arbre binaire non étiqueté par un arbre binaire étiqueté en ignorant simplement les étiquettes. On étiquette un arbre binaire non étiqueté à n nœuds par $[\![0; n-1]\!]$ en suivant l'ordre infixe de son parcours en profondeur. La permutation associée à cet arbre est donnée par le parcours en profondeur par ordre préfixe. La figure 1 propose un exemple (on ne dessine pas les arbres vides).



(a) (b) (c)

Figure 1: (a) un arbre binaire non étiqueté; (b) son étiquetage en suivant un ordre infixe, la permutation associée est $[3; 2; 0; 1; 5; 4; 7; 6]$; (c) un autre arbre binaire non étiqueté.

1. Étiqueter l'arbre (c) de la figure 1 et donner la permutation associée.
2. Écrire une fonction `parcours_prefixe : arbre -> int list` qui renvoie la liste des étiquettes d'un arbre dans l'ordre préfixe de son parcours en profondeur.
3. Écrire une fonction `etiquette : arbre -> arbre` qui prend en paramètre un arbre dont on ignore les étiquettes et qui renvoie un arbre identique mais étiqueté par les entiers de $[\![0, n-1]\!]$ en suivant l'ordre infixe d'un parcours en profondeur.

Une permutation σ de $[\![0, n-1]\!]$ est **triable avec une pile** s'il est possible de trier la liste $[\sigma_0; \sigma_1; \ldots; \sigma_{n-1}]$ en utilisant uniquement une structure de pile comme espace de stockage interne. On considère l'algorithme suivant, énoncé ici dans un style impératif :

- Initialiser une pile vide;
- Pour chaque élément en entrée :
  - Tant que l'élément est plus grand que le sommet de la pile, dépiler le sommet de la pile vers la sortie;
  - Empiler l'élément en entrée dans la pile;
- Dépiler tous les éléments restant dans la pile vers la sortie.

Par exemple, pour la permutation $[3; 2; 0; 1; 5; 4; 7; 6]$, on empile $3, 2, 0$, on dépile 0 , on empile 1 , on dépile $1, 2, 3$, on empile 5,4 , on dépile 4,5 , on empile 7,6 , on dépile 6 , 7 . On obtient la liste triée $[7; 6; 5; 4; 3; 2; 1; 0]$ en supposant avoir ajouté en sortie les éléments dans une liste. On admet qu'une permutation est triable par pile si et seulement cet algorithme permet de la trier correctement.

5. Dérouler l'exécution de cet algorithme sur la permutation associée à l'arbre (c) de la figure1 1 et vérifier qu'elle est bien triable par pile.
6. Écrire une fonction `trier : int list -> int list` qui implémente cet algorithme dans un style fonctionnel. Par exemple, trier $[3; 2; 0; 1; 5; 4; 7; 6]$ doit s'évaluer en la liste $[7; 6; 5; 4; 3; 2; 1; 0]$. On utilisera directement une liste pour implémenter une pile.
7. Montrer que s'il existe $0 \leq i < j < k \leq n-1$ tels que $\sigma_k < \sigma_i < \sigma_j$, alors σ n'est pas triable par une pile.
8. On se propose de montrer que les permutations de $[\![0, n-1]\!]$ triables par une pile sont en bijection avec les arbres binaires non étiquetés à n noeuds.

(a) Montrer que la permutation associée à un arbre binaire est triable par pile. On pourra remarquer le lien entre le parcours préfixe et l'opération empiler d'une part et le parcours infixe et l'opération dépiler d'autre part.

(b) Montrer qu'une permutation triable par pile est une permutation associée à un arbre binaire.

Exercice 2. Minimum sur les fenêtres de taille k

Soit `a` un tableau de n entiers et $k \in \mathbb{N}$. L'objectif de cet exercice est de créer un tableau `a_min` de taille $n - k$ tel que `a_min[i]` soit le minimum des `a[j]` pour $j \in [\![i, i + k]\!]$.

1. Quelle serait la complexité de la méthode naïve ?
2. Écrire un meilleur algorithme en pseudo-code, en utilisant une file de priorité. Quelle serait la complexité ?

Dans la suite, on va implémenter un algorithme utilisant une file à deux bouts `q` (*deque* ou *double-ended queue* en anglais) qui va contenir des éléments de `a`. Une file à deux bouts possède les opérations suivantes :

- `add_right(q, x)` : ajoute l'élément x à la fin de `q`.
- `add_left(q, x)` : ajoute l'élément x au début de `q`.
- `peek_right(q)` : renvoie l'élément à la fin de `q`.
- `peek_left(q)` : renvoie l'élément au début de `q`.
- `pop_right(q)` : supprime l'élément à la fin de `q`.
- `pop_left(q)` : supprime l'élément au début de `q`.
- `is_empty(q)` : détermine si `q` est vide.

3. Donner des implémentations possibles pour une file à deux bouts et discuter de leurs complexités. Implémenter une file à deux bouts en C (on écrira seulement `add_right`, `peek_right`, `pop_right`).

À l'itération i, l'algorithme va considérer `a[i]` avec l'invariant de boucle suivant : `q` contient, dans l'ordre, des indices $i_1, ..., i_p$ de `a` (où i_1 est l'élément au début de `q` et i_p l'élément de fin) tels que :

- $i_1 \leq i_2 \leq ... \leq i_p$.
- `a[i_1]` $\leq$ `a[i_2]` $\leq ... \leq$ `a[i_p]`.
- $i_1 > i - k$ (on considère seulement les k derniers éléments).

4. Avec ces notations, que vaut `a_min[i]` ?
5. Comment mettre à jour `q` à l'itération $i + 1$?
6. En déduire une fonction `int* minimum_sliding(int* a, int n, int k)` renvoyant `a_min`.
7. Quelle est la complexité de `minimum_sliding` ?

Exercice 3. Tableau autoréférent

1. Écrire une fonction `somme : int array -> int -> int` telle que l'appel `somme t i` calcule la somme partielle $\sum_{k=0}^{i} t.(k)$ des valeurs du tableau t entre les indices 0 et i inclus.

Un tableau t de $n > 0$ éléments de $[\![0, n-1]\!]$ est dit **autoréférent** si pour tout indice $0 \leq i < n$, $t.(i)$ est exactement le nombre d'occurrences de i dans t, c'est-à-dire que

$$\forall i \in [\![0, n-1]\!], \quad t.(i) = \text{card}(\{k \in [\![0, n-1]\!] \mid t.(k) = i\})$$

Ainsi, par exemple, pour $n = 4$, le tableau suivant est autoréférent :

i	0	1	2	3
$t.(i)$	1	2	1	0

En effet, la valeur 0 existe en une occurrence, la valeur 1 en deux occurrences, la valeur 2 en une occurrence et la valeur 3 n'apparaît pas dans t.

3. Justifier rapidement qu'il n'existe aucun tableau autoréférent pour $n \in [\![1; 3]\!]$ et trouver un autre tableau autoréférent pour $n = 4$.
4. Écrire une fonction `est_auto : int array -> bool` qui vérifie si un tableau de taille $n > 0$ est autoréférent. On attend une complexité en $O(n)$.
5. Écrire une fonction `gen_auto : int -> unit` affichant tous les tableaux autoréferents de taille donnée. On pourra supposer qu'il existe une fonction `affiche` permettant d'afficher un tableau.
6. Quelle est la complexité de `gen_auto` ?

Pour accélérer la recherche, on peut élaguer l'arbre de recherche (repérer le plus rapidement possible qu'on se trouve dans une branche ne pouvant pas donner de solution).

7. Que peut-on dire de la somme des éléments d'un tableau autoréférent? En déduire une stratégie d'élagage pour accélérer la recherche.
8. Que peut-on dire si juste après avoir affecté la case $t.(i)$, il y a déjà strictement plus d'occurrences d'une valeur $0 \leq k \leq i$ que la valeur de $t.(k)$? En déduire une stratégie d'élagage supplémentaire et la mettre en œuvre. Comparer expérimentalement (sur ordinateur) le temps d'exécution avec la fonction de la question 5.
9. Après avoir affecté la case $t.(i)$, combien de cases reste-t-il à remplir? Combien de ces cases seront complétées par une valeur non nulle? À quelle condition est-on alors certain que la somme dépassera la valeur maximale possible à la fin? En déduire une stratégie d'élagage supplémentaire et la mettre en œuvre. Combien de temps faut-il pour résoudre le problème pour $n = 30$?
10. Montrer qu'il existe un tableau autoréférent pour tout $n \geq 7$. On pourra conjecturer la forme de ce tableau en testant empiriquement pour différentes valeurs de $n \geq 7$. On ne demande pas de montrer que cette solution est unique.

Exercice 4. Arbre de segment

Soit `a` un tableau d'entiers.
On souhaite concevoir, à partir de `a`, une structure de donnée `t` permettant de réaliser efficacement les opérations suivantes :

- `min_range i j t` : renvoie le minimum des éléments de `a` entre les indices `i` et `j`.
- `set i e t` : met à jour la structure pour que `a.(i)` soit remplacé par `e`.

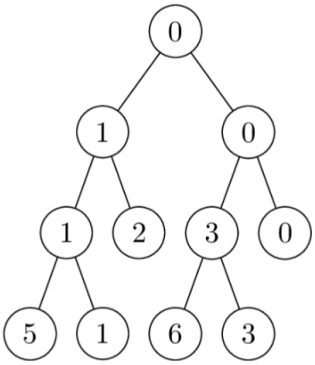
1. Proposer une solution naïve et donner sa complexité.

Dans la suite, on va utiliser un arbre de segments :

Définition : Arbre de segments

Un arbre de segments (pour un tableau `a`) est un arbre binaire dont :
- Les feuilles sont les éléments de `a`
- Chaque noeud est étiqueté par un triplet `(m, i, j)` tel que son sous-arbre contienne les feuilles `a.(i), ... , a.(j)` et `m` est le minimum de ces valeurs.

Par exemple, voici un arbre de segments obtenu à partir du tableau `[|5; 1; 2; 6; 3; 0|]`, où on a représenté seulement les minimums (premiers éléments de chaque triplet) :



Ainsi, les feuilles sont bien les éléments du tableau `[|5; 1; 2; 6; 3; 0|]` et chaque noeud correspond à un minimum sur une certaine plage du tableau.
Remarque : Il y a d'autres arbres de segments possibles pour le même tableau.
On utilisera le type suivant :

```
type tree = E | N of int * int * int * tree * tree
```

Ainsi, un sous-arbre `N(m, i, j, g, d)` possède `a.(i)`, `a.(i + 1)`, ..., `a.(j)` comme feuilles, de minimum `m` et de sous-arbres `g`, `d`.

2. Écrire une fonction `make : int array -> tree` qui construit un arbre de segments à partir d'un tableau d'entiers. On fera en sorte que l'arbre construit soit de hauteur logarithmique en la taille du tableau.
3. Écrire une fonction `set i e t` qui met à jour `t` en remplaçant `a.(i)` par `e`.
   Quelle est la complexité de cette fonction?
4. Écrire une fonction `min_range i j t` renvoyant le minimum des éléments de `a` entre les indices `i` et `j`.
5. Montrer que la complexité de `min_range i j t` est $O(\log(n))$, où n est la taille de `a`.
6. On s'intéresse à un autre problème : calculer efficacement une somme d'éléments entre les indices `i` et `j`, dans un tableau. Adapter les fonctions précédentes pour y parvenir.

Exercice 5. Arbretas (treap)

On peut montrer qu'un arbre binaire de recherche (ABR) construit en ajoutant un à un n entiers choisis « uniformément au hasard » a une hauteur moyenne $O(\log(n))$. Si les éléments à rajouter ne sont pas générés aléatoirement, mais sont tous connus à l'avance, on peut commencer par les mélanger aléatoirement puis les rajouter dans cet ordre aléatoire pour obtenir à nouveau une hauteur moyenne $O(\log(n))$. Pour cela, on considère l'algorithme suivant (mélange de Knuth), où `Random.int (i+1)` renvoie un entier uniformément au hasard entre 0 et `i` :

```
let shuffle a =
    for i = 1 to Array.length a - 1 do
        swap a i (Random.int (i + 1))
    done
```

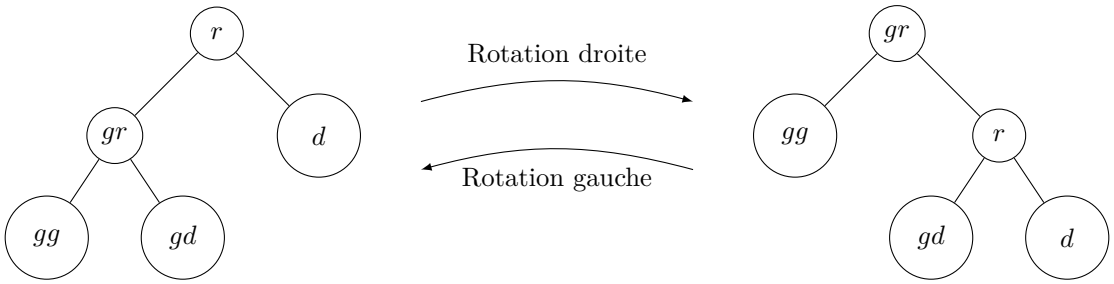
1. Écrire la fonction `swap : 'a array -> int -> int` utilisée par `shuffle`, telle que `swap t i j` échange `t.(i)` et `t.(j)`.
2. (à faire à la fin s'il reste du temps) Montrer que `shuffle t` applique une permutation choisie uniformément au hasard sur le tableau `t`.

Lorsque la totalité des éléments à rajouter n'est pas connue à l'avance, on peut utiliser une structure appelée **arbretas** qui est un arbre binaire (`type 'a arb = V | N of 'a * 'a arb * 'a arb`) dont les noeuds sont étiquetés par des couples (élément, priorité), où la priorité est un nombre entier choisi uniformément au hasard au moment de l'ajout de l'élément. De plus :

- les éléments doivent vérifier la propriété d'ABR.
- la priorité d'un sommet doit être inférieure à la priorité de ses éventuels fils (propriété de tas min sur les priorités).

3. Dessiner un arbretas dont les couples (élément, priorité) sont : (1, 4), (5, 6), (3, 8), (2, 2), (0, 7).
4. Étant donnés des éléments et priorités tous distincts, montrer qu'il existe un unique arbretas les contenants.

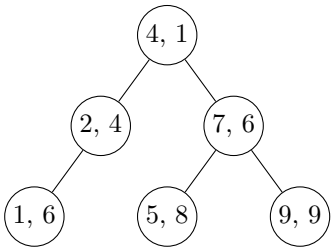
Nous allons utiliser des opérations de rotation sur un arbretas `N(r, N(gr, gg, gd), d)` :



5. Écrire une fonction `rotd` effectuant une rotation droite sur un arbre `N(r, N(gr, gg, gd), d)`.

On supposera définie dans la suite une fonction `rotg` pour effectuer « l'inverse » d'une rotation droite. On remarquera que, si `a` est un ABR, `rotg a` et `rotd a` sont aussi des ABR.
Pour ajouter un sommet s dans un arbretas (en conservant la structure d'arbretas), on l'ajoute comme dans un ABR classique (en ignorant les priorités) puis, si sa priorité est inférieure à celle de son père, on applique une rotation sur son père pour faire remonter s et on continue jusqu'à rétablir la structure d'arbretas.

6. Dessiner l'arbretas obtenu en rajoutant (6, 0) à l'arbretas suivant :



7. Écrire une fonction utilitaire `prio` renvoyant la priorité de la racine d'un arbre (on renverra `max_int`, c'est à dire le plus grand entier représentable en base 2 sur le processeur, si cet arbre est vide).
8. Écrire une fonction `add a e` ajoutant `e` (qui est un couple (élément, priorité)) à un arbretas `a`, en conservant la structure d'arbretas.

Pour supprimer un élément d'un arbretas, on commence par le chercher comme dans un ABR classique (en ignorant les priorités) puis on le fait descendre avec des rotations jusqu'à ce qu'il devienne une feuille qu'on peut alors supprimer librement.

9. Écrire une fonction `del` supprimant un élément dans un arbretas, en conservant la structure d'arbretas.