Exercice 1. Happy array

Soit t un tableau de n entiers. On note $s(i,j) = t[i] + ... + t[j-1]$ et $p(j) = s(0,j)$ (appelé somme préfixe).
Un tableau est heureux (*happy*) si toutes ses sommes préfixes sont positives ou nulles.
On note $S(t)$ la somme des sous-tableaux heureux de t ($S(t) = \sum_{\substack{t' \text{ sous-tableau} \\ \text{heureux de } t}}$ somme des éléments de t).

1. Que vaut $S(t)$ si $t = [1, -2, 3, -2, 4]$?
2. Écrire un algorithme en $O(n^2)$ pour trouver $S(t)$.

Pour trouver $S(t)$ plus efficacement, nous allons nous intéresser à un autre problème (*all nearest smaller values*) consistant à calculer, pour chaque position i dans un tableau t, le plus grand indice $l[i] < i$ tel que $t[l[i]] \leq t[i]$. Si un tel indice n'existe pas, on pose $l[i] = -1$.

3. Calculer l si $t = [1, 4, 2, 3, 5]$
4. En utilisant une pile, écrire un algorithme pour obtenir l à partir de t, en $O(n)$.
5. On peut en fait calculer l en $O(n)$ sans utiliser de pile. Dans le code ci-dessous, on suppose pour simplifier que $t[0]$ est le minimum de t.

```
for i = 0 to n - 1:
    j = i - 1
    while t[j] >= t[i]:
        j = l[j]
    l[i] = j
```

   Montrer que le code ci-dessus calcule l correctement, et que sa complexité est linéaire.

Revenons maintenant sur le problème du calcul de $S(t)$. On suppose avoir précalculé toutes les valeurs de $p(j)$ (dans un tableau p).

6. Soit i un indice de t. Expliquer comment obtenir le plus grand indice j tel que $t[i:j]$ (sous-tableau de t de l'indice i à j) est un tableau heureux à l'aide du problème *all nearest smaller values* et en utilisant p.
7. Aurait-on pu utiliser une recherche par dichotomie pour la question précédente ?
8. En déduire un algorithme en $O(n)$ pour trouver $S(t)$.

Exercice 2. 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
```

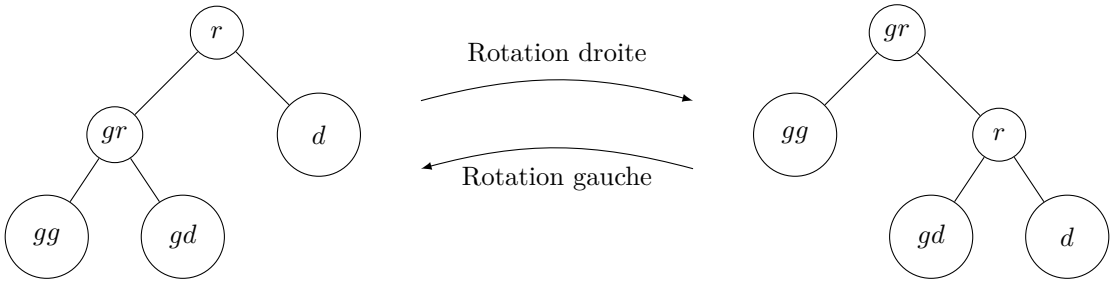
1. Écrire la fonction `swap : 'a array -> int -> int` utilisée par `shuffle`, telle que `swap t i j` échange `t.(i)` et `t.(j)`.
2. Montrer que `shuffle t` applique une permutation choisie uniformément au hasard sur le tableau `t`.

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.

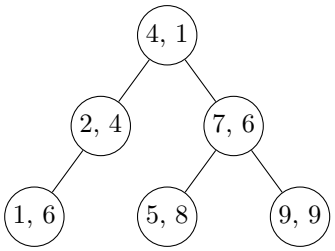
Nous allons utiliser des opérations de rotation sur un arbretas `N(r, N(gr, gg, gd), d)` :



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.

Exercice 3. Arbre de segments

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`.

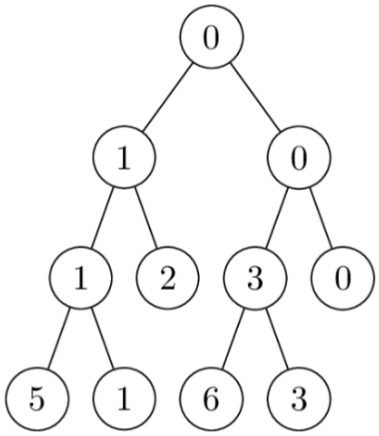
1. Proposer une solution naïve et donner sa complexité.

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.