On considère dans ce TP que l'on a des clefs notée $a_1, \dots, a_n$ appartenant à un ensemble possédant un ordre total (typiquement des entiers). On veut placer ces clefs dans un arbre binaire de recherche. On sait qu'il existe de nombreuses manières de construire un arbre binaire de recherche avec un ensemble de clefs donné.

On suppose que l'on connaisse de plus la fréquence avec laquelle on va chercher les différentes clefs [1]. Précisément pour $i \in [\![1, n]\!]$, on note $f_i \in [0, 1]$ la fréquence avec laquelle on va chercher la clefs a_i dans l'arbre binaire de recherche.

On sait que, l'algorithme de recherche d'un mot dans un arbre binaire de recherche nécessite un temps proportionnel à sa profondeur dans l'arbre. Si bien que, si pour tout entier $i \in [\![1, n]\!]$ on note p_i la profondeur de a_i dans l'arbre binaire de recherche, on considère que le temps pour trouver a_i dans l'arbre est $t_i = 1 + p_i$. On appelle alors *coût* d'un arbre, et on note $c(T)$ le temps moyen pour trouver une clef :

$$c(T) = \sum_{i=1}^{n} f_i \times (1 + p_i).$$

On veut donc construire un arbre binaire de recherche qui minimise cette valeur.

Dans tout ce TP on utilise des arbres binaires étiquetés par des entiers définis par le type

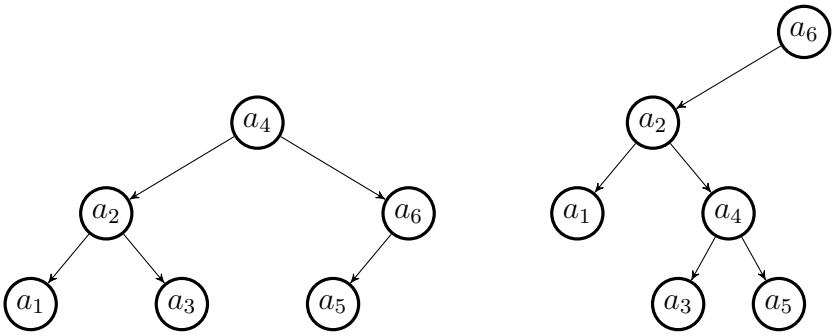
```
type arbre = Vide | N of arbre * int * arbre;;
```

De ce fait, on identifie a_i et i pour tout entier i.

1) On suppose que l'on a six clefs $a_1 < a_2 < a_3 < a_4 < a_5 < a_6$ et que

$$f_1 = 0,1; f_2 = 0,4; f_3 = 0,2; f_4 = 0,15; f_5 = 0,1; f_6 = 0,05.$$

On considère deux arbres binaires de recherche



Calculer (à la main) le coût de ces deux arbres.

2) On suppose que l'on a n clefs et que l'on dispose d'un tableau `f` de type `float array` tel que la case i contienne la fréquence f_i. On n'utilise pas la case d'indice 0 du tableau.

Écrire une fonction `cout : float array-> 'a arbre -> float` telle que si `t` est un `'a arbre`, `cout f t` renvoie le coût de l'arbre `t`.

On garantira une complexité linéaire en la taille de l'arbre.

1. On peut imaginer que les clefs sont en fait les mots d'un dictionnaire et on sait que certains mots sont plus souvent cherchés que d'autres

3) Pour tout i et j compris entre 0 et n, on note :
   - $T_{i,j}$ un arbre binaire de recherche optimal pour les clefs $a_{i+1}, a_{i+2}, \dots, a_j$.
   - $f_{i,j} = f_{i+1} + f_{i+2} + \cdots + f_j$ la somme des fréquences des éléments de l'arbre $T_{i,j}$.
   - $r_{i,j}$ l'étiquette (parmi les clefs $a_{i+1}, a_{i+2}, \dots, a_j$) de la racine de l'arbre $T_{i,j}$.
   - $c_{i,j}$ le coût de l'arbre $T_{i,j}$.

   On conviendra que pour tout i, $T_{i,i}$ est un arbre vide.

   Si on considère l'arbre $T_{i,j}$ et que l'on note $k \in [\![i+1, j]\!]$ tel que $r_{i,j} = a_k$. Le sous-arbre de gauche contient les clefs $a_{i+1}, \dots, a_{k-1}$ et le sous-arbre de droite les clefs $a_{k+1}, \dots, a_j$. Il est clair que si $T_{i,j}$ est optimal alors les sous-arbres de droite et de gauche sont aussi des sous-arbres optimaux pour les clefs qu'ils contiennent.

   a) Justifier qu'avec les notations précédentes, le coût de l'arbre ayant pour racine a_k, dont le sous arbre de gauche est $T_{i,k-1}$ et le sous arbre de droite est $T_{k,j}$ est

   $$c_{i,j}(k) = f_{i,j} + c_{i,k-1} + c_{k,j}.$$

   b) En déduire que

   $$c_{i,j} = f_{i,j} + \min_{i+1 \leq k \leq j} (c_{i,k-1} + c_{k,j}).$$

   c) On va avoir besoin de calculer les valeurs $f_{i,j}$. Écrire une fonction

   `tableauF : float array -> float array array`

   qui prend en entrée le tableau `f` des fréquences (on rappelle que la case numérotée 0 n'est pas utilisée) et renvoie `t` de type `float array array` tel que `t.(i).(j)` contient $f_{i,j}$. On garantira une complexité en $O(n^2)$ où n est la taille de `f` à l'aide d'un algorithme de programmation dynamique.

   d) En déduire une fonction

   `tableauxAbrOpt : float array -> (int array array)*(float array array)`

   tel que si `f` est un tableau comme ci-dessus alors `tableauxAbrOpt f` renvoie un couple de matrices (r, c) où r est le tableau des racines et c le tableau des coûts qui correspondent aux arbres optimaux. C'est-à-dire que `c.(i).(j)` stocke $c_{i,j}$ et `r.(i).(j)` stocke $r_{i,j}$.

   e) Tester la fonction pour `f = [|0. ; 0.1; 0.4; 0.2; 0.15 ; 0.1 ; 0.05|]`.

   f) Écrire une fonction `abrOpt : float array -> arbre` qui renvoie l'arbre optimal.