I Tri fusion

Le tri fusion permet de trier une liste l récursivement, de la façon suivante :

- Diviser l en deux listes l_1 et l_2.
- Trier récursivement l_1 et l_2 pour obtenir l_1' et l_2'.
- Fusionner l_1' et l_2' pour obtenir l', qui est une liste triée des éléments de l.

1. Écrire une fonction `divise : 'a list -> 'a list * 'a list` qui sépare une liste en deux (à $\pm$ 1 près).
   Exemple : `divise [1;2;3;4;5]` peut renvoyer `([1;3;5], [2;4])`.

> Solution : On peut mettre un élément sur deux dans chaque liste, en regardant les éléments deux par deux (ce qui implique de distinguer le cas d'un singleton dans le `match`).

```
let rec divise l = match l with
    | [] -> [], []
    | [x] -> [x], []
    | x::y::l -> let l1, l2 = divise l in x::l1, y::l2
```

2. Écrire une fonction `fusion : 'a list -> 'a list -> 'a list` qui fusionne deux listes triées en une seule liste triée.
   Exemple : `fusion [1;3;5] [2;4]` doit renvoyer `[1;2;3;4;5]`.

> Solution :

```
let rec fusion l1 l2 = match l1, l2 with
    | [], l2 -> l2
    | l1, [] -> l1
    | x::l1, y::l2 -> if x < y then x::fusion l1 l2 else y::fusion l1 l2
```

3. Écrire une fonction `tri : 'a list -> 'a list` qui trie une liste par le tri fusion.
   Exemple : `tri [5;4;3;2;1]` doit renvoyer `[1;2;3;4;5]`.

> Solution :

```
let rec tri l = match l with
    | [] -> []
    | [x] -> [x]
    | _ -> let l1, l2 = divise l in fusion (tri l1) (tri l2)
```

4. Montrer que la complexité de `tri` est $\mathrm{O}(n \log n)$ sur une liste de taille n.

> Solution : Soit $C(n)$ la complexité de `tri l` pour `l` de taille n.

$$
\begin{aligned}
C(n) &= \underbrace{O(n)}_{divise} + \underbrace{O(n)}_{fusion} + 2C(n/2) \leq Kn + 2C(n/2) \\
&\leq Kn + 2K\frac{n}{2} + 4C(n/4) = 2Kn + 4C(n/4) \\
&\leq ... \leq pKn + 2^p C(n/2^p) \underset{p=\log_2(n)}{=} \boxed{O(n\log_2(n))}
\end{aligned}
$$

> où Kn est un majorant de la complexité de `divise` et `fusion`.

II Calcul de rang

Soit `t` un tableau de n entiers. Si `e` est un élément de `t`, on appelle **rang** de `e` le nombre d'éléments de `t` inférieurs strictement à `e`.
Étant donné un entier k, on souhaite obtenir l'élément de rang k dans `t`.
Par exemple, l'élément de rang 3 dans `[0; 42; 7; 4; 1]` est 7.

II.1 Avec prétraitement

1. Comment effectuer un prétraitement en $\mathrm{O}(n\log(n))$ sur le tableau, pour ensuite être capable d'obtenir l'élément de rang k en $\mathrm{O}(1)$?

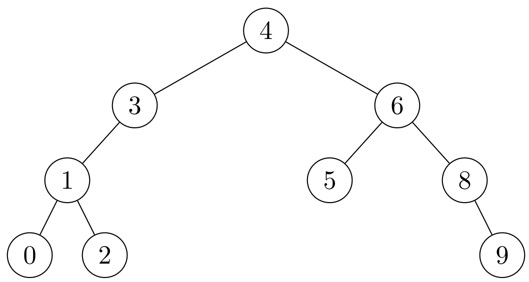
> Solution : Utiliser un algorithme de tri.

II.2 Avec une file de priorité

1. Rappeler les opérations requises pour une file de priorité.
2. Rappeler une implémentation possible d'une file de priorité.
3. Quelle serait alors la complexité pour obtenir l'élément de rang k dans une file de priorité de taille n ?

II.3 Avec un ABR

Un **arbre binaire de recherche** (ABR) est un arbre binaire tel que, pour chaque noeud d'étiquette r et de sous-arbres g et d, r est supérieur à toutes les étiquettes de g et inférieur à toutes les étiquettes de d.



Exemple d'ABR

On utilise le type suivant : `type 'a arb = V | N of 'a * 'a arb * 'a arb`

4. Écrire une fonction `abr_min : 'a arb -> 'a` qui renvoie la plus petite valeur d'un ABR (c'est à dire l'élément de rang 0). Quelle est sa complexité ?

> Solution : Le minimum est tout à gauche dans un arbre binaire de recherche. On se déplace suivant la branche tout à gauche, en $\mathrm{O}(h)$ avec h la hauteur de l'arbre.
>
> ```
> let rec abr_min = function
>     | V -> failwith "abr_min"
>     | N (e, V, _) -> e
>     | N (_, g, _) -> abr_min g
> ```

Pour récupérer l'élément de rang k quelconque dans un ABR, on ajoute une information à chaque sommet s : le nombre de noeuds du sous-arbre enraciné en s.
On change donc le type : `type 'a arb_rang = V | N of 'a * 'a arb_rang * 'a arb_rang * int`

Pour ajouter un élément e en conservant la propriété d'ABR, on se déplace dans l'arbre en comparant l'étiquette r du noeud actuel à e : si $e < r$ on se déplace dans le sous-arbre gauche, sinon dans le sous-arbre droit.
Lorsqu'on arrive à un emplacement vide, on peut ajouter l'élément en créant un nouveau noeud (l'élément ajouté est donc une feuille).

5. Dessiner l'ABR obtenu en ajoutant 7 à l'ABR donné en exemple.
6. Écrire une fonction `add : 'a -> 'a arb_rang -> 'a arb_rang` qui ajoute un élément dans un `arb_rang` tout en conservant la propriété d'ABR et en mettant à jour le nombre de noeuds. Complexité ?

> Solution : On parcourt une branche de l'arbre jusqu'à trouver une position libre pour rajouter l'élément. La complexité est donc linéaire en la hauteur de l'arbre.

```
let rec add e = function
    | V -> N (e, V, V, 1)
    | N (r, g, d, n) ->
       if e < r then N (r, add e g, d, n + 1)
       else N (r, g, add e d, n + 1)
```

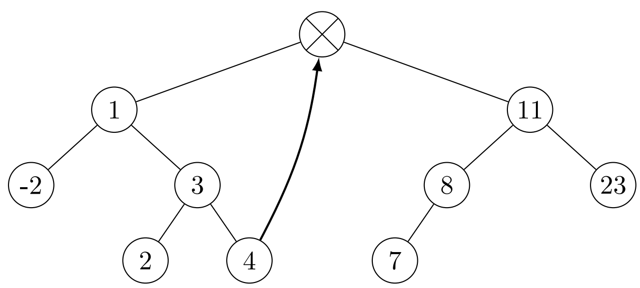
7. Écrire une fonction `get : int -> 'a arb_rang -> 'a` pour obtenir l'élément de rang k dans un `arb_rang`.

   Solution : Si le sous-arbre gauche contient $k-1$ éléments, on renvoie la racine. Sinon, on s'appelle récursivement sur l'un des sous-arbres. Chaque appel récursif augmente la profondeur du sommet visité : il y a donc $O(h)$ tels appels, chacun en $O(1)$.

```
let taille = function
    | V -> 0
    | N (_, _, _, n) -> n

let rec get k = function
    | V -> failwith "get"
    | N (r, g, d, n) ->
       let n_g = taille g in
       if k = n_g then r
       else if k < n_g then get k g
       else get (k - n_g - 1) d
```

Pour supprimer un élément en conservant la propriété d'ABR, on commence par chercher l'élément à supprimer. Puis on le remplace par le maximum de son sous-arbre gauche (ou par le minimum de son sous-arbre droit).



8. Écrire une fonction `del_max : 'a arb_rang -> 'a * 'a arb_rang` qui supprime le maximum dans un `arb_rang` et renvoie cet élément ainsi que l'arbre modifié (en diminuant de 1 son nombre de sommets).

   Solution :

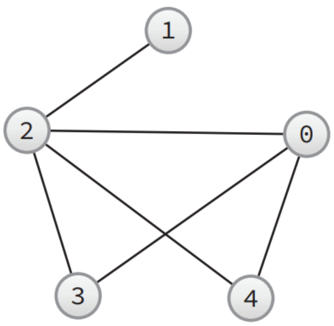
```
let rec del_max = function
    | V -> failwith "del_max"
    | N (r, g, V, n) -> (r, g)
    | N (r, g, d, n) ->
        let (r', d') = del_max d in
        (r', N (r, g, d', n - 1))
```

9. Écrire une fonction `del : 'a -> 'a arb_rang -> 'a arb_rang` qui supprime un élément dans un `arb_rang` tout en conservant la propriété d'ABR et en mettant à jour le nombre de noeuds.

   Solution :

```
let rec del e = function
    | V -> V (* e n'a pas été trouvé *)
    | N (r, g, d, n) ->
        if e = r then
            let (r', g') = del_max g in
            N (r', g', d, n - 1)
        else if e < r then N (r, del e g, d, n - 1)
        else N (r, g, del e d, n - 1)
```

III Exercice E3A 2018 sur les graphes



1. Écrire la matrice d'adjacence du graphe ci-dessus.
2. Écrire en Caml une fonction `chemin : int array array -> int list -> bool` qui prend en entrée la matrice d'adjacence d'un graphe et un chemin (une liste de sommets du graphe) et qui vérifie si ce chemin est possible dans le graphe. Par exemple, sur le graphe ci-dessus, avec le chemin `[2;1;0;4]` la fonction `chemin` doit renvoyer `false` car les sommets `1` et `0` ne sont pas connectés. Avec le chemin `[1;2;3]` la fonction `chemin` doit renvoyer `true` car les sommets `1` et `2` sont connectés, ainsi que les sommets `2` et `3`.

3. Écrire une fonction Caml `atteignable : int list array -> int -> int -> bool` telle que, si ##1 est un graphe orienté représenté par liste d'adjacence et ##1, ##1 deux sommets de ##1, `atteignable g r u` renvoie `true` si et seulement si il existe un chemin de ##1 à ##1 dans ##1.

Solution :

1. $$\begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

2.

```
let rec chemin g c = match c with
    | [] | [_] -> true
    | u::v::q -> g.(u).(v) = 1 && chemin g (v::q)
```

3. On peut utiliser un parcours en profondeur avec une référence `res` pour savoir si `u` a été trouvé :

```
let atteignable g u v =
    let n = Array.length g in
    let visited = Array.make n false in
    let rec aux w =
        if w = v then true
        else if visited.(w) then false
        else List.exists aux g.(w) in
    aux u
```