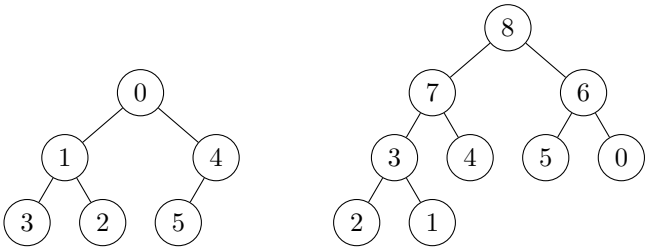
TD structures de données 2: corrigé partiel

I Un tas de questions

1. Dessiner le tas min (sous forme d'arbre et de tableau) obtenu à partir d'un tableau vide en ajoutant 3, 1, 2, 0 puis en supprimant deux fois le minimum puis en ajoutant 5, 0, 1, 4.

2. Dessiner le tas max (sous forme d'arbre et de tableau) obtenu en convertissant le tableau `[|3; 1; 0; 8; 4; 5; 6; 2; 7|]` en tas, en utilisant l'algorithme linéaire du cours.
   ▶ Vérifiez que vous obtenez le tas de gauche pour la question 1 et celui de droite pour la question 2.



3. Dans un tas, obtient-on un tri en parcourant les éléments dans l'ordre infixe? Préfixe? En largeur?
   ▶ Non: les tas précédents sont des contre-exemples.

4. Écrire une fonction pour vérifier qu'un tableau représente bien un tas max. Complexité?
   ▶ `aux i` détermine si les éléments de `tas` d'indices $\geq i$ forment un tas. aux étant appelé une fois par élément du tas, et chaque appel effectuant un nombre constant d'opérations, `is_tas` est linéaire en la taille du tas.

```
let is_tas tas =
  let rec aux i =
    if i > tas.n then true
    else (fd i >= tas.n || tas.t.(fd i) < tas.t.(i))
      && (fg i >= tas.n || tas.t.(fg i) < tas.t.(i))
      && aux (fd i) && aux (fg i) in
  aux 0;;
```

5. Écrire une fonction `tas_to_arb : 'a tas -> 'a arb` pour convertir d'un tas (sous forme de tableau) en arbre, avec `type 'a tas = { t : 'a array; mutable n : int }`.
   ▶

```
let tas_to_arb tas =
  let rec aux i = (* renvoie l'arbre enraciné en tas.t.(i) *)
    if i >= tas.n then V
    else N(tas.t.(i), aux (fg i), aux (fd i)) in
  aux 0;;
```

6. Écrire un algorithme en $O(n \log(k))$ pour fusionner k listes triées en une unique liste triée de taille n, en utilisant une FP min et le fait que l'on puisse comparer des listes avec `<` (dans l'ordre lexicographique). On pourra utiliser le type suivant de FP min:

```
type 'a fp =
  { add : 'a -> unit;
    is_empty : unit -> bool;
    take_min : unit -> 'a };;
```

   ▶ On insère d'abord toutes les listes dans la FP. Puis, tant que possible, extraire la tête de liste minimum de la FP et l'ajouter à la liste de sortie. Il y a k listes dans la FP donc les opérations `add` et `take_min` sont en $O(\log(k))$, en utilisant une FP implémentée par tas. On effectue k `add` et n `take_min`, d'où la complexité $O(k \log(k)) + O(n \log(k)) = O(n \log(k))$ (en supposant que les listes sont non vides, on a $n \geq k$).

```
let kfusion fp ll = (* renvoie la fusion triée des listes triées de ll *)
  let rec init = function
    | [] -> ()
    | l::q -> fp.add l; init q in
  init ll;
  let rec aux () =
    if fp.is_empty () then [] (* fusion terminée *)
    else match fp.take_min () with
    | [] -> aux ()
    | e::q -> e::aux () in
  aux ();;
```

2ème méthode (sans utiliser de FP...): utiliser la même idée que le tri fusion. On sépare les listes à fusionner en deux, qu'on fusionne récursivement. On obtient ainsi deux listes triées qu'on fusionne ensuite avec la même fonction de fusion de liste (`list_fusion`) que le tri fusion.

```
let rec list_fusion l1 l2 = match l1, l2 with (* fusion de 2 listes triées *)
  | [], _ -> l2
  | _, [] -> l1
  | e1::q1, e2::q2 -> if e1 < e2 then e1::list_fusion q1 l2
                      else e2::list_fusion q2 l1;;

let rec divise l = match l with
  | [] -> [], []
  | [e] -> [e], []
  | e1::e2::q -> let l1, l2 = divise q in
                 e1::l1, e2::l2;;

let rec klist_fusion ll = match ll with (* fusionne une liste de listes triées *)
  | [l] -> l
  | _ -> let ll1, ll2 = divise ll in
         list_fusion (klist_fusion ll1) (klist_fusion ll2);;
```

Autre possibilité, en fusionnant les listes 2 à 2 à chaque étape, jusqu'à obtenir une seule liste triée:

```
let rec klist_fusion = function (* fusionne une liste de listes triées *)
  | [l] -> l
  | _ ->
     let rec fusion_paire = function (* fusionne les listes par paires *)
       | [] -> []
       | [l] -> [l]
       | l1::l2::q -> (list_fusion l1 l2)::fusion_paire q in
     klist_fusion (fusion_paire ll);;
```

7. Qu'obtient-on dans la question précédente si toutes les listes sont de taille 1?
   ▶ Cela revient alors à trier n éléments. On obtient donc un algorithme de tri en $O(n\log(n))$.

8. Expliquer comment implémenter une « file de priorité à deux bouts » qui permette d'extraire aussi bien le minimum que le maximum en $O(\log(n))$, et d'ajouter un élément en $O(\log(n))$ également.
   ▶ Avec un ABR équilibré (exemple: AVL).

9. Est-il possible d'implémenter une file de priorité à n éléments avec extraction et ajout en $o(\log(n))$?
   ▶ Non: ceci donnerait un algorithme de tri d'un tableau de taille n en $o(n\log(n))$, ce qui est impossible (Preuve: un algorithme de tri peut être représenté par son arbre de décision, qui doit posséder $\geq n!$ feuilles. Sa hauteur (le nombre maximum de comparaisons effectuées par l'algorithme) h vérifie alors $f \leq 2^h$, i.e $h \geq \log_2(n!) \sim n\log(n)$).

10. Décrire un algorithme en $O(n\log(k))$ pour trier un tableau « presque trié » `t` de taille n, où chaque élément est à au plus k indices de sa bonne position dans le tableau trié.
    ▶ On ajoute les k premiers éléments dans une FP min. Puis, tant que possible: on extrait le minimum qu'on rajoute au tableau de sortie et on ajoute le prochain élément de `t` (s'il existe) à la FP.

II Compression de Huffman

Comme un arbre de Huffman contient de l'information seulement aux feuilles, on utilise le type:

```
type 'a arb_huffman = F of 'a | N of 'a arb_huffman * 'a arb_huffman;;
```

1. Écrire une fonction `lire arb l` qui renvoie un couple constitué:

   - du premier caractère obtenu en se déplaçant, dans `arb`, suivant les éléments (0 ou 1) de la liste `l`
   - de la partie de `l` non lue

   On supposera que `l` est une liste valide (qui correspond bien à une suite de caractères).

```
let rec read arb l = match arb with
  | F(e) -> e, l
  | N(g, d) -> read (if (List.hd l) = 0 then g else d) (List.tl l);;
```

2. Écrire une fonction `decode : 'a arb_huffman -> int list -> 'a list` qui décode une liste de 0 et 1 avec un arbre de Huffman.

```
let rec decode arb l =
  if l = [] then []
  else let e, next = read arb l in
       e::(decode arb next);;
```

On veut maintenant construire l'arbre de Huffman. Pour cela on utilise une file de priorité min `fp` contenant des couples (fréquence, arbre), ordonné par fréquence croissante:

- Initialement, ajouter à `fp` tous les caractères (en tant que feuille) avec leur fréquence.
- Tant que possible: retirer les deux plus petits couples `(f1, a1)` et `(f2, a2)` de `fp` et y rajouter `(f1+f2, N(a1, a2))`.

On peut montrer que le dernier arbre obtenu est alors celui de Huffman (il optimise la longueur moyenne du codage d'un texte).
On utilisera le type abstrait de file de priorité suivant:

```
type 'a fp =
  { add : 'a -> unit;
    is_empty : unit -> bool;
    take_min : unit -> 'a };;
```

3. Quel arbre obtient t-on avec les fréquences (20, 'a'), (15, 'b'), (7, 'c'), (14, 'd'), (44, 'e')?
   ▶ `N(F('e'), N(N(F('c'), F('d')), N(F('b'), F('a'))))`

4. Écrire une fonction `to_huffman` construisant un arbre de Huffman à partir d'une FP vide et d'une liste de fréquences.
   ▶ En traduisant l'algorithme ci-dessus:

```
let to_huffman fp freq =
  let rec init = function
    | [] -> ()
    | (f, c)::q -> fp.add (f, F(c)); init q in
  init freq;
  let rec build () =
    let f1, a1 = fp.take_min () in
    if fp.is_empty () then a1
    else let f2, a2 = fp.take_min () in
         fp.add (f1 + f2, N(a1, a2));
         build ()
    in build ();;
```

   On peut faire un peu plus simple en remarquant qu'à chaque itération de l'algorithme, la taille de la FP diminue de 1, donc qu'il faut un nombre d'itérations égal à `Array.length freq - 1` pour obtenir un seul arbre:

```
let to_huffman fp freq =
  let rec init = function
    | [] -> ()
    | (f, c)::q -> fp.add (f, F(c)); init q in
  init freq;
  for i = 0 to List.length freq - 2 do
    let (f1, a1), (f2, a2) = fp.take_min (), fp.take_min () in
    fp.add (f1 + f2, N(a1, a2))
  done;
  snd (fp.take_min ());;
```

5. Écrire une fonction `to_dico arb di` stockant le code de chaque caractère dans un dictionnaire `di`, à partir d'un arbre de Huffman `arb`. On utilisera le type de dictionnaire suivant:

```
type ('key, 'value) dico =
  { add : 'key -> 'value -> unit;
    get : 'key -> 'value list;
    del : 'key -> unit };;
```

   ▶ On peut utiliser un accumulateur (initialement vide) pour conserver le chemin (inversé) depuis la racine:

```
let rec to_dico arb di chemin = match arb with
  | F(e) -> di.add e (List.rev chemin)
  | N(g, d) -> to_dico g di (0::chemin); to_dico d di (1::chemin);;
```

6. En déduire une fonction pour coder un texte (sous forme d'une liste de lettres), à partir d'un dictionnaire (rempli par la fonction précédente).
   ▶

```
let rec code di = function
  | [] -> []
  | c::q -> (List.hd (di.get c)) @ code di q;;
```

III Calcul de rang (order statistics)

Étant donné un tableau `t` de taille n, on souhaite sélectionner le $k$ème plus petit (que l'on appelle élément de rang k). Nous allons voir plusieurs approches, en effectuant éventuellement un prétraitement.

Méthode simple

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

Avec un tas min

On pourra utiliser les fonctions de tas définies en cours.

2. Quel est la complexité de prétraitement pour transformer `t` en tas?
   ▶ O(taille de `t`) avec l'algorithme du cours, en construisant le tas de bas en haut (en commençant par les feuilles).

3. Écrire une fonction `get_rank : 'a tas -> int -> 'a` renvoyant l'élément de rang donné dans un tas (tout en conservant les même éléments dans le tas). Quelle est sa complexité?
   ▶ On extrait k fois le minimum, on les remet et on renvoie la valeur du $k$ème. Complexité $O(k\log(n))$.

```
let rec get_rank tas k =
  if k = 1 then (let res = tas.take_min () in
                 tas.add res;
                 res)
  else (let mini = tas.take_min () in
        let res = get_rank tas (k - 1) in
        tas.add mini;
        res);;
```

Avec un ABR

4. Comment obtenir l'élément de rang 1 d'un ABR? En quelle complexité?
   ▶ En regardant tout à gauche de l'arbre (complexité $O(h)$):

```
let rec abr_min = function
  | V -> failwith "arbre vide"
  | N(r, V, d) -> r
  | N(r, g, d) -> 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 sommets du sous-arbre enraciné en s.
On utilise donc le type: `type 'a arb_rang = V | N of 'a * 'a arb_rang * 'a arb_rang * int;;`

5. Écrire une fonction pour ajouter un élément dans un `arb_rang`. Complexité?
   ▶ Comme dans le cours mais en augmentant la taille de 1 là où on rajoute l'élément:

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

6. Écrire une fonction pour supprimer un élément dans un `arb_rang`. Complexité?
   ▶ On peut utiliser la méthode par fusion vue dans le TD précédent:

```
let sz = function
  | V -> 0
  | N(_, _, _, n) -> n;;
let rec fusion g d = match d with
  | V -> g
  | N(rd, gd, dd, nd) -> N(rd, fusion g gd, dd, nd + sz g);;

let rec del a e = match a with
  | N(r, g, d, n) when e = r -> fusion g d
  | N(r, g, d, n) when e < r -> N(r, del g e, d, n-1)
  | N(r, g, d, n) -> N(r, g, del d e, n-1);;
```

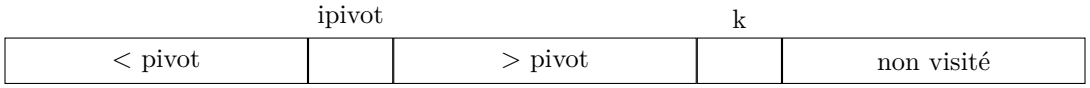
7. Écrire une fonction pour récupérer l'élément de rang k dans un `arb_rang` en temps linéaire en sa hauteur (et indépendant de k).
   ▶ Si le sous-arbre gauche contient $k-1$ éléments, on renvoie la racine. Sinon, on s'appelle récursivement sur l'un des sous-arbres. Chaque appel récursif augmente la profondeur du sommet visité: il y a donc $O(h)$ tels appels, chacun en $O(1)$.

```
let sz = function
  | V -> 0
  | N(_, _, _, n) -> n;;
let rec get_kth a k = match a with
  | N(r, g, d, _) when k = sz g + 1 -> r
  | N(r, g, d, _) when k < sz g + 1 -> get_kth g k
  | N(r, g, d, _) -> get_kth d (k - sz g - 1);;
```

Avec un algorithme de partition similaire au tri rapide

Cette méthode consiste à chercher l'élément de rang k en choisissant un pivot p (normalement aléatoirement, mais ici on prendra le premier élément possible pour simplifier) puis en partitionnant le reste du tableau en deux: les éléments inférieurs à p et ceux supérieurs. Enfin on cherche récursivement dans l'un des deux tableaux.

8. Écrire une fonction `partition` telle que `partition t i j` modifie le tableau `t` de sorte que, entre les indices i et j, il contienne d'abord les éléments inférieurs au pivot, puis le pivot, puis les éléments supérieurs. `partition` doit être en temps $O(j-i)$ et, si possible, en complexité mémoire $O(1)$ (c'est à dire sans création de tableau intermédiaire). `partition` devra renvoyer le nouvel indice du pivot.
   ▶ `ipivot` est l'indice où il faut rajouter le prochain élément inférieur au pivot. On met le pivot à sa bonne place seulement à la fin de l'algorithme.




```
let partition t i j =
  let pivot = t.(i) and ipivot = ref i in
  for k = i + 1 to j do
    if t.(k) < pivot
    then (t.(!ipivot) <- t.(k);
          incr ipivot;
          t.(k) <- t.(!ipivot))
  done;
  t.(!ipivot) <- pivot;
  !ipivot;;
```

9. En déduire un algorithme pour obtenir l'élément de rang k dans un tableau de taille n. Quelle est sa complexité dans le pire des cas? Dans le meilleur?
   ▶ On partitionne le tableau puis on regarde si le $k$ème est dans la partie gauche ou droite. Dans le pire cas, l'intervalle de recherche est diminué de 1 à chaque itération donc on effectue n appels à `partition`, d'où une complexité $\Theta(n^2)$. Dans le meilleur cas, on trouve l'élément de rang k dès la première partition, en $\Theta(n)$.

```
let rec get_kth t k i j = (* renvoie l'élément de rang k de t en sachant qu'il est
                             entre les indices i et j *)
  let ipivot = partition t i j in
  if k = ipivot + 1 then t.(ipivot)
  else if k < ipivot + 1 then get_kth t k i (ipivot - 1)
  else get_kth t k (ipivot + 1) j;;
```

On peut montrer que cet algorithme est en complexité moyenne linéaire en la taille n du tableau. Pour une démonstration (utilisant linéarité de l'espérance, calcul de sommes...), voir le livre Introduction à l'algorithmique, chapitre 9: Médians et rangs.
Dans ce même chapitre est décrit un algorithme « médian des médians » remarquable (mais un peu compliqué) de calcul de l'élément de rang k en temps $O(n)$ dans le pire cas.

Bilan

Faire un tableau avec la complexité de prétraitement et de recherche de l'élément de rang k, pour chacune des méthodes ci-dessus.

IV Union-Find (sera réutilisé pour les graphes et les automates)

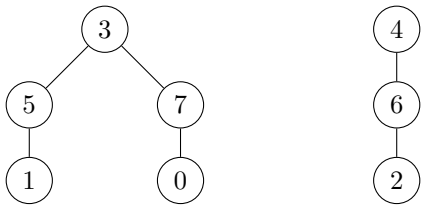
La structure Union-Find sert à représenter des classes d'équivalences (ou: une partition) d'un ensemble E. Pour simplifier, on suppose $E = \{0, ..., n-1\}$. Une structure Union-Find `uf` possède 3 opérations:

- `create n`: crée une partition de $\{0, ..., n-1\}$ où chaque élément est seul dans sa partition.
- `find uf i`: renvoie un *représentant* de la classe de `i`.
- `union uf i j`: réunie les classes de `i` et `j`.

On implémente chaque classe comme un arbre (non binaire a priori) dont les sommets sont les éléments de la classe et la racine son représentant. On représente ces arbres par un tableau `pere`, où `pere.(i)` est le père de `i` et `pere.(i) = i` ssi `i` est représentant de sa classe. `union uf i j` branche alors le représentant de i sur celui de j (le représentant de i devient un fils du représentant de j).

1. Pourquoi ne représente t-on pas ces arbres par le type persistant habituel?
   ▶ On a besoin d'accéder au père d'un sommet, ce qui est impossible avec le type `V | N of ...`

2. Dessiner les arbres représentés par `[|7; 5; 6; 3; 4; 3; 4; 3|]`
   ▶ On commence par dessiner les racines, puis leurs fils...



`find uf i` doit trouver la racine de l'arbre contenant `i`, ce qui demande une complexité linéaire en la hauteur de l'arbre. Deux optimisations permettent de diminuer cette hauteur:

- *Compression de chemin*: lors d'un appel `find uf i`, le représentant de i devient le père de i. Ainsi un appel ultérieur à `find uf i` sera plus rapide.
- *Union par rang*: associe à chaque sommet un rang (initialement 0) qui majore la hauteur de l'arbre. On branche la racine de l'arbre de moindre rang vers la racine de l'arbre de plus fort rang (en cas d'égalité, on branche arbitrairement). On stockera dans un tableau `rang` le rang de chaque sommet.

3. Écrire des fonctions `create`, `find`, `union` en utilisant la compression de chemin et l'union par rang. On utilisera le type suivant: `type union_find = { pere : int array; rang : int array };;`
   ▶

```
let create n =
  let t = Array.make n 0 and rg = Array.make n 0 in
  for i = 1 to n - 1 do
    t.(i) <- i
  done;
  { pere = t; rang = rg };;
```

```
let rec find uf i =
  if uf.pere.(i) = i then i
  else let rep = find uf uf.pere.(i) in
       uf.pere.(i) <- rep;
       rep;;
```

```
let union uf i j =
  let repi = find uf i and repj = find uf j in
  if uf.rang.(repi) < uf.rang.(repj) then uf.pere.(repi) <- repj
  else uf.pere.(repj) <- repi;
  if uf.rang.(repi) = uf.rang.(repj) then uf.rang.(repj) <- uf.rang.(repj) + 1;;
```

On peut montrer que la complexité *amortie* de `find` et `union` est $O(\alpha(n))$ où α est une fonction à croissance très lente[1]: $\alpha(n) \leq 4$, $\forall n \leq 16^{512}$. `find` et `union` sont donc «quasiment en temps constant».

4. Comment pourrait-on adapter la structure pour pouvoir ajouter des éléments?
   ▶ On pourrait utiliser un tableau dynamique.

[1] son inverse, la fonction de Ackermann, est célèbre en informatique