MP2I – DS4 2023-2024 – 4h

Pour chaque question, le langage imposé par l'énoncé doit être respecté. La rédaction doit être soignée.
Les fonctions doivent être documentées et commentées à bon escient. Le code doit être indenté et d'un seul tenant (pas de flèche pour dire qu'un bout de code se situe ailleurs par exemple). Le code illisible ne sera pas corrigé.
Sauf précision contraire, les réponses doivent être justifiées.

Tout document interdit. Calculatrice interdite. Téléphone éteint et rangé dans le sac. Pas de montre connectée.

Ce qui est illisible ou trop sale ne sera pas corrigé.
Soulignez ou encadrez vos résultats.

Correction :
Cet énoncé est entièrement repris de l'épreuve d'option informatique du Concours Centrale-Supélec 2016, mais ce n'en est qu'un extrait. À noter que l'épreuve complète comptait 13 questions supplémentaires avec des tableaux (épreuve originale à faire en 4h).

Correction :
Rappel : toute fonction non explicitement demandée dans l'énoncé doit être documentée. En `OCaml` il s'agit de donner explicitement son type et d'expliquer succintement ce qu'elle fait. C'est d'autant plus important en cas d'erreur dans le code de la fonction ou de fonction illisible : ça permet au correcteur de tenir compte de son utilisation comme si elle était correcte pour le reste. Ce n'est pas possible si la fonction n'est pas documentée.

Tout le sujet est à faire en `OCaml`.

1 Présentation

1.1 Motivation

Trier des données est un problème récurrent dans tous les systèmes d'information. Dans un système travaillant en temps réel (par exemple un système de freinage d'une voiture) ou un système pouvant être soumis à des attaques (par exemple un serveur web), on s'intéresse à la complexité dans le pire des cas.
Or, dans une application réelle, les données que l'on veut trier ne sont pas quelconques mais suivent une certaine distribution aléatoire, qui est loin d'être uniforme : ainsi, il est fréquent que les données soient déjà presque triées. De plus, de nombreux algorithmes de tris (même les plus performants) atteignent leur complexité maximale lorsque les données sont déjà triées.
Le but de ce problème est d'étudier un algorithme de tri, proche du tri par tas, mais présentant avec lui quelques différences significatives et notamment des performances intéressantes lorsque les données qu'il reçoit sont presque triées.
Dans tout le problème, on triera, par ordre croissant, des valeurs entières.
Dans toutes les questions de complexité en temps, la mesure de complexité à considérer est le nombre de comparaisons par la relation d'ordre $\leq$ entre entiers.

1.2 Notations et préliminaires

Dans la suite, si un objet mathématique est noté t, on notera `t` l'objet `OCaml` qui l'implante et, si t appartient à un ensemble T implanté en `OCaml` par le type `tt`, on écrira de manière équivalente $t \in T$ ou `t : tt`. Par exemple, pour signifier que l désigne une liste d'entiers, on notera `l : int list`.
On tiendra compte dans la suite que la structure à trier n'est pas un ensemble, car le même élément peut être répété plusieurs fois. Ainsi on sera amené à manipuler en `OCaml` des listes dans lesquelles un même élément peut apparaître plusieurs fois, et des arbres dans lesquels la même valeur peut étiqueter plusieurs nœuds différents.
Dans la suite, lorsqu'il s'agira de déterminer le minimum d'une telle structure, il pourra être atteint plusieurs fois ; de même, lorsqu'on triera ou "réunira" deux telles structures, ce sera toujours en tenant compte des répétitions, c'est-à-dire sans perte d'éléments. Ainsi, par exemple, pour les listes $l_1 = (7,4,2,8,2,7,3)$ et $l_2 = (5,2,3,9)$, le minimum de l_1 est 2, trier l_1 consiste à renvoyer la liste `[2;2;3;4;7;7;8]` et "réunir" l_1 et l_2 consiste à renvoyer la liste `[7;4;2;8;2;7;3;5;2;3;9]`.
De manière générale, lorsqu'on dira que deux structures de données contiennent les mêmes éléments, ce sera toujours en tenant compte des répétitions. Par exemple les listes `[1;2;2]` et `[2;1;2]` contiennent les mêmes éléments mais pas les listes `[3;4;4]` et `[4;3;3]`.

2 Algorithme sur des arbres

2.1 Tri par insertion

Question 1 : Écrire la fonction `insere: int -> int list -> int list` insérant, en temps linéaire sur la longueur de la liste (sans avoir à justifier cette complexité), un élément dans une liste supposée triée, c'est-à-dire telle que pour toute liste `u` supposée triée et tout élément `x`, `insere x u` est une liste `v` telle que

- `v` contient les mêmes éléments que `x::u` et
- `v` est triée.

Question 2 : Écrire la fonction `tri_insertion : int list -> int list` triant la liste reçue en argument en utilisant la fonction précédente.

Correction :

```
1 (** insere e lst insere la valeur e dans la liste triee lst de sorte que
      le résultat soit trié *)
2 let rec insere e = function
3   | [] -> [e]
4   | hd :: tl -> if hd < e then hd :: (insere e tl) else e :: hd :: tl
5
6 (** tri_insertion lst trie la liste lst *)
7 let rec tri_insertion = function
8   | [] -> []
9   | hd :: tl -> insere hd (tri_insertion tl)
```

Question 3 : Pour $n \in \mathbb{N}$, on note $P_I(n)$ le nombre de comparaisons effectuées par l'appel `tri_insertion l` dans le cas le pire pour une liste `l` de longueur n. On note de même $M_I(n)$ le nombre de comparaisons effectuées dans le cas le meilleur. Déterminer l'ordre de grandeur exact de $P_I(n)$ et $M_I(n)$.

Correction :
La fonction `tri_insertion` effectue un appel à `insere` pour chacun de ses éléments. La complexité de la fonction `insere` dépend de la place finale de l'élément à insérer : s'il s'insère au début, elle s'exécute en temps constant, s'il s'insère à la fin, la liste est parcourue élément par élément et la complexité est donc proportionnelle à la longueur de cette liste.

La complexité de `tri_insertion` est donc en $\Omega(n)$ et $\mathcal{O}(n^2)$ car $\sum_{k=0}^{n-1} k = \frac{n(n-1)}{2} = \Theta(n^2)$.

Si la liste de départ est dans l'ordre décroissant, alors `tri_insertion` s'exécute en $\Theta(n^2)$ opérations, donc $P_I(n) = \Theta(n^2)$.

Si la liste de départ est dans l'ordre croissant, alors `tri_insertion` s'exécute en $\Theta(n)$ opérations, donc $M_I(n) = \Theta(n)$.

Notez, pour la n-ème fois que je ne prétends jamais *que l'ordre décroissant est le pire des cas ou que l'ordre croissant est le meilleur des cas. Si je le disais, il faudrait que je le prouve. Ce n'est pas la stratégie que j'utilise (on pourrait, mais c'est difficile dans plein de cas de prouver que telle entrée est le pire des cas pour une taille donnée). Ce que je fais pour le pire des cas, c'est :*

- *donner une majoration qui est vraie pour* toutes *les entrées d'une certaine taille (donc en particulier pour le pire de cas qui est une des entrées) – cela me donne une majoration dans le pire des cas en $\mathcal{O}(f(n))$, pour une certaine fonction f, si n dénote la taille de l'entrée ;*
- *donner une minoration qui est vraie pour* au moins *une entrée de chaque taille (et comme le pire des cas de la même taille prend plus de temps que cette entrée-là, ça le minore aussi) – cela me donne une minoration dans le pire des cas en $\Omega(g(n))$ pour une certaine fonction g, si n dénote la taille de l'entrée ;*
- *si $f = g$, je conclus sur un ordre de grandeur exact en $\Theta(f(n))$.*

2.2 Tas binaires

On appelle *arbre* un arbre binaire étiqueté par des éléments de $\mathbb{N}$. Un tel arbre est implanté en `OCaml` à l'aide de la déclaration de type suivante :

```
type arbre =
  | Vide
  | Noeud of int * arbre * arbre
```

On définit la *hauteur* et la *taille* (appelée aussi nombre d'éléments) d'un arbre a, notées respectivement haut(a) et $|a|$, par induction sur la structure de l'arbre :
- haut(`Vide`) = 0 et haut(`Noeud(x, a1, a2)`) = 1 + max{haut(`a1`), haut(`a2`)} ;
- |`Vide`| = 0 et |`Noeud(x, a1, a2)`| = 1 + |`a1`| + |`a2`|.

pour tous arbres binaires `a1` et `a2` et tout entier `x`.
Si `a=Noeud(x, a1, a2)`, alors `x`, `a1` et `a2` sont appelés respectivement la *racine*, le *fils gauche* et le *fils droit* de l'arbre `a`.
On dit que deux arbres *ont mêmes éléments* s'ils ont les mêmes ensembles d'étiquettes et que chaque étiquette présente apparaît le même nombre de fois dans chacun des arbres.
On dit qu'un arbre binaire est *parfait* s'il s'agit de l'arbre vide `Vide`, ou s'il est de la forme `Noeud(x, a1, a2)` où `a1` et `a2` sont deux arbres parfaits de même hauteur.
On dit qu'un arbre binaire est un *tas binaire parfait* (ou simplement un *tas parfait*) si c'est un arbre parfait et que la valeur étiquetant chaque nœud de l'arbre est inférieure ou égale à celle de ses fils.
On dit qu'un arbre est un *quasi-tas* si c'est un arbre de la forme `Noeud(x, a1, a2)` et que `a1` et `a2` sont des tas binaires parfaits de même taille : aucune contrainte d'ordre n'est donc imposée sur l'étiquette de la racine `x`.
Étant donné un arbre non vide a, on note $\min_{\mathscr{A}}(a)$ le minimum des éléments qu'il contient.

Question 4 : Pour $k \in \mathbb{N}$, on note m_k la taille d'un arbre binaire parfait de hauteur k. Exprimer m_{k+1} en fonction de m_k et en déduire la valeur de m_k pour tout $k \in \mathbb{N}$.

Correction :
Il faut lire correctement le sujet : celui-ci précise clairement qu'il faut donner une relation entre m_k et m_{k+1} pour ensuite déterminer m_k. Si vous faites la preuve autrement, vous ne répondez pas à la question.
La définition récursive de la taille est donnée par |`Noeud(x, a1, a2)`| = 1 + |`a1`| + |`a2`|. Dans le cas d'un arbre parfait de hauteur $k+1$, ses deux sous-arbres sont des arbres parfaits de hauteur k par définition. On en déduit donc immédiatement que $m_{k+1} = 2m_k + 1$.
En faisant une somme téléscopique, on obtient

$$m_k = \sum_{i=0}^{k-1} 2^i = 2^k - 1 .$$

Et ce n'est pas possible de l'exprimer uniquement sous forme de somme sans mettre une formule fermée.
Il est prudent quand on vous demande une formule et que les premiers exemples sont simples, de vérifier sur les exemples ce que vous avez trouvé (ça évite par exemple de se décaler de 1).

Question 5 : Écrire la fonction `min_tas : arbre -> int` telle que pour tout tas binaire parfait `a` non vide, `min_tas a` s'évalue en $\min_{\mathscr{A}}$(a). On fera en sorte que la complexité soit constante (sans avoir à justifier cette complexité).

Question 6 : Écrire la fonction `min_quasi : arbre -> int` telle que pour tout quasi-tas `a`, `min_quasi a` s'évalue en $\min_{\mathscr{A}}$(a), en temps constant (sans avoir à justifier cette complexité).

Question 7 : Écrire la fonction `percole : arbre -> arbre` telle que `percole a` vaut `a` si `a` est l'arbre vide et, si `a` est un quasi-tas, s'évalue en un tas binaire parfait contenant les mêmes éléments que `a`. Donner en le justifiant l'ordre de grandeur exact de la complexité dans le cas le pire de la fonction `percole` en fonction de la hauteur k du quasi-tas `a`.

Correction :
Plusieurs questions vous demandaient explicitement des complexités en temps constant. Dans ces cas-là, se lancer dans des fonctions récursives devrait au moins vous mettre la puce à l'oreille et faire que vous vous posiez des questions.
Le min d'un tas-min non vide est à sa racine, le min d'un quasi-tas vide est soit à sa racine, soit au niveau inférieur (s'il y a quelque chose). *N'hésitez pas à mettre une phrase pour expliquer succintement votre code.*
Concernant les tas ou quasi-tas vides : il faut essayer d'éviter les filtrages non exhaustifs, c'est source d'erreur. Vous

pouviez soit faire un `failwith`*, soit donner une valeur supposée impossible, par exemple -1.*
La percolation est l'opération percoler-vers-le-bas que nous avons vue en cours.

```
(** précondition: l'argument est un tas; calcule son min *)
let min_tas = function
  | Vide -> failwith "cas impossible tas"
  | Noeud (x, _, _) -> x

(** précondition: l'argument est un quasi-tas; calcule son min *)
let min_quasi = function
  | Vide -> failwith "cas impossible quasi_tas"
  | Noeud (x, Noeud (g, _, _), Noeud (d, _, _)) -> min x (min g d)
  | Noeud (x, _, _) -> x

(** pécondition: l'argument est un quasi-tas; percole quasi s'évalue en
    un
     tas-min contenant les mêmes éléments que quasi *)
let rec percole quasi = match quasi with
  | Vide -> Vide
  | Noeud (x, Noeud (g, g0, g1), Noeud (d, d0, d1)) ->
     let m = min_quasi quasi in
     if m = x then quasi
     else if m = g then Noeud (g, percole (Noeud (x, g0, g1)), Noeud (d,
         d0, d1))
     else Noeud (d, Noeud (g, g0, g1), percole (Noeud (x, d0, d1)))
  | Noeud (x, _, _) -> quasi
```

2.3 Décomposition parfaite d'un entier

L'algorithme de tri que l'on va étudier repose sur une propriété remarquable des nombres m_k obtenus en question 4.
Étant donné un entier naturel r, on dit qu'un r-uplet $(k_1,\dots,k_r)$ d'entiers naturels non-nuls *vérifie la propriété QSC* (pour "quasi strictement croissant") si l'une des trois conditions suivantes est vérifiée :

- $r \leq 1$; ou
- $r = 2$ et $k_1 \leq k_2$; ou
- $r \geq 3$ et $k_1 \leq k_2 < k_3 < \cdots < k_r$.

En particulier, on convient qu'il existe un unique 0-uplet, noté $()$, et que ce 0-uplet vérifie la propriété QSC.
La propriété remarquable des nombres m_k qui nous intéressera est alors la suivante :

> pour tout entier naturel non nul n, il existe un unique entier r et un unique r-uplet $(k_1,\dots,k_r)$ d'entiers naturels non nuls vérifiant la propriété QSC et tel que $n = m_{k_1} + \cdots + m_{k_r}$, cette somme étant par convention nulle si $r = 0$.

Une telle écriture $n = m_{k_1} + \cdots + m_{k_r}$, où $(k_1,\dots,k_r)$ vérifie la propriété QSC, est appelée une *décomposition parfaite* de n. Par exemple, les entiers de 1 à 5 admettent les décompositions parfaites suivantes : $1 = m_1$, $2 = m_1 + m_1$, $3 = m_2$, $4 = m_1 + m_2$, $5 = m_1 + m_1 + m_2$.
On peut remarquer que, du fait de la stricte croissance de la suite d'entiers $(m_k)_{k\in\mathbb{N}}$, un r-uplet d'entiers naturels $(k_1,\dots,k_r)$ vérifie la propriété QSC si et seulement si le r-uplet $(m_{k_1},\dots,m_{k_r})$ vérifie également cette propriété.
L'unicité d'une décomposition parfaite ne nous préoccupe pas ici (on l'admettra donc), mais seulement son existence. Plus précisément, l'outil dont nous aurons besoin par la suite est un algorithme récursif d'obtention d'une décomposition parfaite.

Question 8 : Donner la décomposition parfaite des entiers 6, 7, 8, 9, 10, 27, 28, 29, 30, 31, 100 et 101.

Correction :
Si vous étiez vraiment perdus, la question suivante permettait de vérifier votre réponse. (Une vérification élémentaire consistait aussi à s'assurer qu'on n'écrivait pas des égalités tout simplement fausses.)

$$
\begin{aligned}
6 &= m_2 + m_2 \\
7 &= m_3 \\
8 &= m_1 + m_3 \\
9 &= m_1 + m_1 + m_3 \\
10 &= m_2 + m_3 \\
27 &= m_1 + m_1 + m_2 + m_3 + m_4 \\
28 &= m_2 + m_2 + m_3 + m_4 \\
29 &= m_3 + m_3 + m_4 \\
30 &= m_4 + m_4 \\
31 &= m_5 \\
100 &= m_2 + m_2 + m_5 + m_6 \\
101 &= m_3 + m_5 + m_6
\end{aligned}
$$

Question 9 : Soit n un entier naturel admettant une décomposition parfaite de la forme $n = m_{k_1} + \cdots + m_{k_r}$. Montrer qu'alors $n+1$ admet une décomposition parfaite de la forme

$$
n+1 = \begin{cases} m_{k_1+1} + (m_{k_3} + \cdots + m_{k_r}) & \text{si } r \geq 2 \text{ et } k_1 = k_2 \\ m_1 + m_{k_1} + \cdots + m_{k_r} & \text{sinon} \end{cases}
$$

Correction :
Cette question en cachait deux : il ne s'agissait pas juste de s'assurer de l'égalité entre $n+1$ et l'expression de droite, mais aussi de s'assurer que l'expression de droite est une décomposition parfaite, c'est-à-dire que la suite des indices est QSC.
Attention : plusieurs d'entre vous sont partis dans une démonstration par récurrence, mais ce n'est clairement pas ce qui est attendu ici, puisque qu'on suppose déjà que n possède une décompostion parfaite, et on doit juste montrer qu'on peut en déduire une décomposition parfaite pour $n+1$.
Supposons donc que $n = m_{k_1} + \cdots + m_{k_r}$ est une décomposition parfaite de n.
On a :

$$
n+1 = 1 + m_{k_1} + \cdots + m_{k_r}\,.
$$

On a trois cas :

- $r \leq 1$: alors en fait cette équation s'écrit $n+1 = 1$ ou $n+1 = 1 + m_{k_1}$ et on a bien une décomposition parfaite pour $n+1$ qui vérifie ce que l'on souhaite ;
- $r = 2$ et $k_1 \leq k_2$: alors

  $$
  n+1 = \begin{cases} 1 + m_{k_1} + m_{k_1} = m_{k_1+1} & \text{si } k_1 = k_2, \\ 1 + m_{k_1} + m_{k_2} & \text{sinon.} \end{cases}
  $$

  est une décomposition parfaite de $n+1$ car $1 \leq m_{k_1} < m_{k_2}$;
- $r \geq 3$ et $k_1 \leq k_2 < k_3 < \cdots < k_r$: alors

  $$
  n+1 = \begin{cases} \underbrace{1 + m_{k_1} + m_{k_1}}_{m_{k_1+1}} + m_{k_3} + \ldots m_{k_r} = m_{k_1+1} + m_{k_3} + \ldots m_{k_r} & \text{si } k_1 = k_2, \\ 1 + m_{k_1} + \cdots + m_{k_r} & \text{sinon.} \end{cases}
  $$

  est une décomposition parfaite de $n+1$ car $m_{k_1+1} \leq m_{k_3} < \cdots < m_{k_r}$ dans le premier cas et $1 \leq m_{k_1} < \cdots < m_{k_r}$ dans le deuxième cas.

Question 10 : Écrire la fonction `decomp_parf : int -> int list` telle que, pour tout entier naturel n, `decomp_parf n` s'évalue en la liste $(m_{k_1}, \ldots, m_{k_r})$ des entiers apparaissant dans la décomposition parfaite de `n` (dans cet ordre). Cette fonction devra avoir une complexité temporelle $\mathcal{O}(n)$ qu'on justifiera brièvement.

Correction :
La question précédente, même si vous ne saviez pas la montrer, vous donnait la relation entre la décomposition parfaite d'un entier et la décomposition parfaite de l'entier précédent. Il suffisait donc d'appliquer la formule. L'entier 0 possède une décomposition parfaite vide, il suffit donc d'appliquer la formule de la question précédente.

```
let rec decomp_parf n =
  if n = 0 then [ ]
  else match decomp_parf (n-1) with
       | [ ] -> [1]
       | [ x ] -> [1; x]
       | hd1 :: hd2 :: tl -> if hd1 = hd2 then (2*hd1 + 1) :: tl
                             else 1 :: hd1 :: hd2 :: tl
```

2.4 Création d'une liste de tas

On appelle *liste de tas* une liste de couples de la forme (a,t) où a désigne un tas binaire parfait et $t=|a|$ est la taille de l'arbre a : il existe donc un entier naturel k tel que $t=m_k$.
Une liste de tas est implantée en `OCaml` par le type `(arbre * int)list`.
Étant donnée une liste de tas h de la forme précédente, on définit

- la *longueur* de h, notée $\text{long}(h)$, par

$$\text{long}(h)=\begin{cases}0 & \text{si } h \text{ est la liste vide}\\ r & \text{si } h=\big((a_1,t_1),\dots,(a_r,t_r)\big)\end{cases}$$

- la *taille* de h, notée $|h|$, par

$$|h|=\begin{cases}0 & \text{si } h \text{ est la liste vide}\\ \underbrace{|a_1|}_{=t_1}+\cdots+\underbrace{|a_r|}_{=t_r} & \text{si } h=\big((a_1,t_1),\dots,(a_r,t_r)\big)\end{cases}$$

- la *hauteur* de h, notée $\text{haut}(h)$, par

$$\text{haut}(h)=\begin{cases}0 & \text{si } h \text{ est la liste vide}\\ \max\big(\text{haut}(a_1),\dots,\text{haut}(a_r)\big) & \text{si } h=\big((a_1,t_1),\dots,(a_r,t_r)\big)\end{cases}$$

- le *minimum* de h, noté $\min_{\mathscr{H}}(h)$, par

$$\min_{\mathscr{H}}(h)=\begin{cases}+\infty & \text{si } h \text{ est la liste vide}\\ \min\big(\min_{\mathscr{A}}(a_1),\dots,\min_{\mathscr{A}}(a_r)\big) & \text{si } h=\big((a_1,t_1),\dots,(a_r,t_r)\big)\end{cases}$$

Comme pour les arbres binaires, on dit que deux listes de tas *ont mêmes éléments* si les deux listes des arbres les constituant font apparaître exactement les mêmes étiquettes avec exactement le même nombre d'apparitions de chaque étiquette. De même, une liste l d'entiers naturels et une liste de tas constituée d'arbres dont les étiquettes appartiennent à $\mathbb{N}$ *ont mêmes éléments* si les deux structures font apparaître exactement les mêmes éléments avec le même nombre d'apparitions de chaque élément.
On dit qu'une liste de tas $h=\big((a_1,t_1),\dots,(a_r,t_r)\big)$ *vérifie la condition TC* (pour "tas croissants") si le r-uplet d'entiers naturels $(t_1,\dots,t_r)$ vérifie la propriété QSC. On peut remarquer qu'une liste de tas vérifie la condition TC si et seulement si $|h|=t_1+\cdots+t_r$ est une décomposition parfaite de $|h|$. En particulier, la liste de tas vide vérifie la condition TC ; on constate enfin que toute liste de tas de la forme $h=\big((a,|a|)\big)$ vérifie la condition TC.

Question 11 :

1. Si h est une liste non vide de tas, a-t-on nécessairement $\text{haut}(h)=\mathcal{O}(\log(|h|))$? A-t-on nécessairement $\text{long}(h)=\mathcal{O}(\log(|h|))$? Justifier.

Correction :
Question très peu traitée et très mal traitée par la plupart de ceux qui s'y sont lancés. Et pourtant pas compliquée si on prend le temps de comprendre les définitions et qu'on n'oublie pas ce qu'on a déjà prouvé. Avant de commencer, je rappelle que la notation $\mathcal{O}$ est bien souvent utilisée pour des calculs de complexité, n'est pas liée en soit à ces calculs. Et dans cette question en particulier, elle n'y est pas liée du tout : l'énoncé ne demande pas la complexité du calcul de la hauteur ou de la longueur d'une liste de tas comme certains ont cru le comprendre (ce d'autant qu'on n'a pas les algorithmes utilisés), mais une évaluation des valeurs de la hauteur et de la longueur d'une liste de tas.

Si $h = ((a_1, t_1), \dots (a_r, t_r))$ est une liste de tas non vide, alors sa hauteur est la hauteur d'un des tas, par exemple a_i : $\text{haut}(h) = \text{haut}(a_i)$. Or d'après la question 4, on a $\text{haut}(a_i) = \mathcal{O}(\log|a_i|)$. Comme $|a_i| \leq |h|$, on en déduit que

$$\text{haut}(h) = \mathcal{O}(\log|h|).$$

Je rappelle que $\mathcal{O}$ est une notion asymptotique, donner une relation entre la hauteur et la taille d'une liste de tas particulière ne permet pas de prouver quoi que ce soit.
Si on considère une liste non vide de tas $h = ((a_1, t_1), \dots (a_r, t_r))$ où tous les tas sont réduits à leur racine, alors : $\text{long}(h) = r$ et $|h| = r$. On en déduit donc que

$$\text{long}(h) \neq \mathcal{O}(\log|h|).$$

2. Même question si h est une liste de tas vérifiant la condition TC.

Correction :
La formule pour la hauteur reste vraie en ajoutant une contrainte puisque c'est une majoration. La formule pour la longueur devient vraie. En effet on peut se placer dans un cas où la suite des tailles est strictement croissante (quitte à éliminer le premier tas). On a une suite de r arbres binaires parfaits de plus en plus grands, donc le dernier est au moins de taille m_r, d'où : $|h| \geq |a_r| \geq 2^r - 1$ d'après la question 4, ce qui permet d'en déduire que $\log|h| + \varepsilon \geq r$ *(je mets ε à cause du -1)*, donc :

$$\text{long}(h) = r = \mathcal{O}(\log|h|).$$

Question 12 : Considérons un arbre réduit à sa racine (c'est à dire un couple $(a, 1)$ correspondant à un tas binaire parfait) et une liste de tas $h = \big((a_1, t_1), \dots, (a_r, t_r)\big)$ vérifiant la condition TC.
Si l'on ajoute le couple $(a, 1)$ en tête de la liste h, on obtient bien une liste de tas

$$(a, 1) :: h = \big((a, 1), (a_1, t_1), \dots, (a_r, t_r)\big),$$

mais qui ne vérifie peut-être plus la condition TC. L'objectif de cette question consiste à concevoir, en utilisant les outils mis en œuvre dans les questions précédentes, un algorithme qui construit une liste de tas h' ayant les mêmes éléments que $(a, 1) :: h$ telle que h' vérifie la condition TC.

1. On considère $h_1 = \big((a_1^1, 1), (a_1^2, 3), (a_1^3, 7)\big)$ et $h_2 = \big((a_2^1, 3), (a_2^2, 3), (a_2^3, 7)\big)$ deux listes de tas vérifiant la condition TC, où les arbres a_i^j sont donnés dans la figure 1.

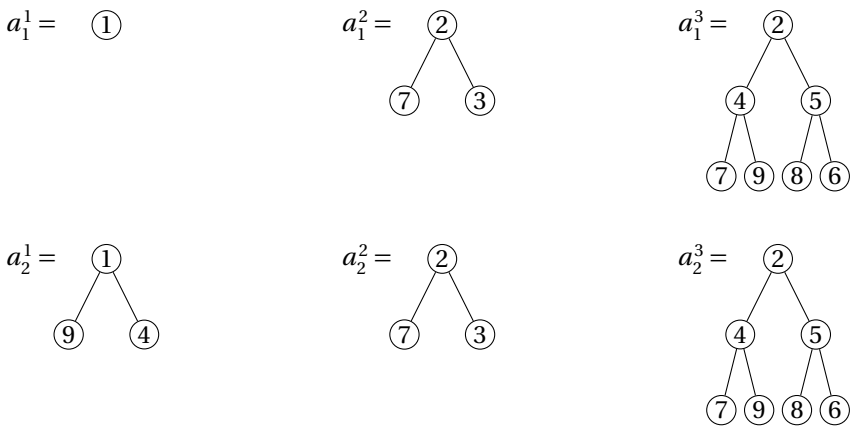
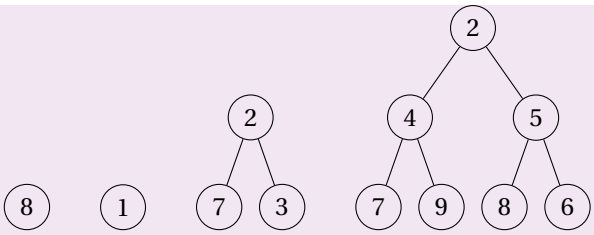


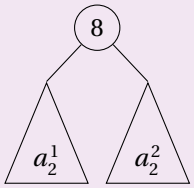
FIGURE 1 –

Expliquer de manière détaillée (à l'aide de représentations graphiques) comment on construit les listes de tas h_1' et h_2' lors de l'ajout de l'arbre a réduit à sa racine d'étiquette 8 dans chacune des listes de tas h_1 et h_2.

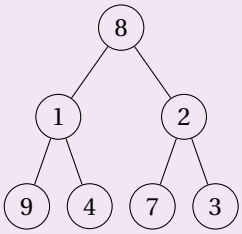
Correction :
L'énoncé spécifie à l'aide de représentations graphiques, *donc pourquoi s'en priver ? Certains écrivent du texte pas clair, là où un dessin suffit. L'algo est demandé à la question suivante, pas à celle-ci.*
Pour h_1' : on ajoute a au début et on constate qu'on a bien une liste de tas qui vérifie la condition TC :



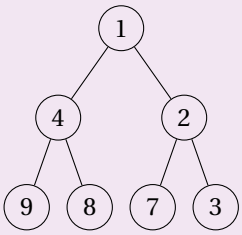
Pour $h'2$, si on se contente d'ajouter a au début, on n'a plus une liste qui vérifie la propriété TC car a_2^1 et a_2^2 ont la même taille et ne sont pas au tout début de la liste. On peut alors les combiner en utilisant a comme racine :



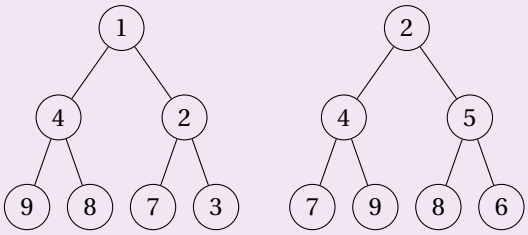
Mais on constate qu'on n'obtient certes un arbre parfait, mais pas un tas :



Il suffit alors de percoler :



Et la liste obtenue vérifie bien la propriété TC :



2. Décrire le plus précisément possible un algorithme qui consiste à construire h' à partir d'un arbre a réduit à sa racine et d'une liste de tas h vérifiant la condition TC. On fera en sorte que cet algorithme ait une complexité dans le cas le pire $\mathcal{O}(\text{haut}(a_1))$ (où a_1 est le premier tas de la liste h) et en $\mathcal{O}(1)$ dans le cas le meilleur. On justifiera soigneusement la correction de la fonction et brièvement sa complexité dans le cas le pire.

Correction :
Il suffit ici de s'inspirer de ce qui a été fait sur l'exemple qui regroupait quasiment tous les cas de figure possible (sauf la liste de départ vide).
Il faut conserver les bonnes habitudes : on précise l'entrée et la sortie de l'algorithme, et on structure celui-ci.
Il ne s'agit pas de donner une vague description, mais bien un algorithme.

entrée : tas a réduit à sa racine et liste de tas $h = ((a_1, t_1), \dots (a_r, t_r))$ qui vérifie TC
sortie : liste de tas qui contient exactement l'étiquette de a et les éléments de h, et qui vérifie TC
si h est vide **alors**
renvoyer $((a, 1))$
si si h a un seul élément ou que ses deux premiers éléments ont des tailles différentes **alors**
renvoyer $((a, 1), (a_1, t_1), \dots, (a_r, t_r))$
si les deux premiers éléments de h ont même taille **alors**
renvoyer $((a', 2t_1 + 1), (a_3, t_3), \dots, (a_r, t_r))$, où a' est l'arbre obtenu en percolant l'arbre de racine l'étiquette de a, de sous-arbre gauche a_1 et de sous-arbre droit a_2.

Il est clair que l'algorithme renvoie une liste d'arbres contenant les éléments voulus (étiquette de a et éléments de h).
Dans les deux premiers cas, il est immédiat que tous les arbres de la liste résultat sont des tas et que la liste vérifie la propriété TC : on ne doit le vérifier que dans le deuxième cas et on a bien $1 \leq t_1 < t_2 < \dots < t_r$.
Dans le troisième cas, on sait que $t_1 < t_3 < \dots < t_r$ par hypothèse. Comme on est dans le cas où $t_1 = t_2$, et que les t_i sont des tailles d'arbres parfaits, on a $2t_1 + 1 \leq t_3$ d'après la relation de récurrence de la question 4.
L'arbre a' est parfait car a_1 et a_2 sont parfaits de même taille, donc de même hauteur, et c'est un tas car on a appliqué `percole` à un quasi-tas.
Dans les deux premiers cas, la complexité est constante. Dans le troisième cas, elle dépend uniquement de l'opération de percolation qui se fait sur un arbre de hauteur haut$(a_1) + 1$, d'où le résultat.

3. Écrire la fonction
`ajoute : int -> (arbre * int)list -> (arbre * int)list`
telle que `ajoute x h` s'évalue en la liste de tas vérifiant la condition TC construite par l'algorithme de la question précédente à partir d'un arbre a réduit à sa racine `x` et une liste de tas h vérifiant la condition TC.

Correction :

```
let ajoute n lst = match lst with
  | [] | [_] -> ( Noeud (n, Vide, Vide), 1 ) :: lst
  | (_, t1) :: (_, t2) :: _ when t1 < t2
    ->  ( Noeud (n, Vide, Vide), 1 ) :: lst
  | (a1, t1) :: (a2, t2) :: lst'
    -> ( percole (Noeud (n, a1, a2)), 2*t1 + 1 ) :: lst'
```

Question 13 : On définit la fonction suivante, de type `int list -> (arbre * int)list` :

```
let rec constr_liste_tas l = match l with
  | [] -> []
  | x::r -> ajoute x (constr_liste_tas r)
```

1. Montrer que le coût de l'appel `constr_liste_tas l` pour une liste `l : int list` déjà triée de longueur n dans le cas le pire est en $\mathcal{O}(n)$.
2. Montrer que, pour une liste `l : int list` de longueur n, `constr_liste_tas l` a une complexité temporelle en $\mathcal{O}(n \log(n))$ dans le cas le pire. [1]

Correction :
Il est impossible de traiter la première question sans disposer de la fonction `ajoute`*, puisque celle-ci est appelée dans la fonction* `constr_liste_tas`*. On a beau connaître d'après l'énoncé la complexité dans le pire et dans le meilleur des cas de la fonction* `ajoute`*, sans la fonction on n'a pas moyen de savoir quand se produit le meilleur des cas. Ça pose question sur le degré de compréhension de ceux qui l'ont fait.*
La fonction `constr_liste_tas` appelle le fonction ajoute pour chaque élément de la liste, du dernier au premier. Si la liste est triée, l'ajout se fait donc du plus grand au plus petit élément. Donc quand on ajoute un élément, il n'y a jamais besoin de percoler, et l'ajoute se fait en temps constant, soit une complexité en $\mathcal{O}(n)$.
Dans le pire des cas, on percole à chaque fois, ce qui se fait en temps proportionnel à la hauteur du tas, qui est

1. On peut en fait montrer que cette complexité est un $\Theta(n)$ mais cela n'est pas demandé.

elle-même inférieure à la hauteur de la liste de tas et en $\mathcal{O}(\log n)$ d'après la question 11.2, soit un complexité en $\mathcal{O}(n \log n)$ puisqu'on le fait pour chaque élément de la liste.

2.5 Tri des racines

On dit qu'une liste de tas $h = \big((a_1, t_1), \dots, (a_r, t_r)\big)$ *vérifie la condition RO* (pour "racines ordonnées") si les tas présents dans la liste apparaissent par ordre croissant de leurs racines ou, ce qui est équivalent, par ordre croissant de leurs minimums : $\min_{\mathscr{A}}(a_1) \leq \cdots \leq \min_{\mathscr{A}}(a_r)$.
On considère une liste de tas $h = \big((a_1, t_1), \dots (a_r, t_r)\big)$ vérifiant la condition TC et ne vérifiant pas nécessairement la condition RO. On veut réarranger les éléments apparaissant dans h de façon à obtenir une liste de tas h' vérifiant à la fois les conditions TC et RO. Si l'on trie brutalement par insertion les tas de h dans l'ordre croissant des racines, on risque de perdre la condition TC. On va donc mettre en place un tri s'inspirant du tri par insertion mais consistant à échanger les racines des tas présents dans h plutôt que les tas eux-mêmes.

Question 14 : Écrire une fonction `echange_racines : arbre -> arbre -> (arbre * arbre)` de complexité constante telle que si `a1` et `a2` sont deux arbres binaires non vides, `echange a1 a2` s'évalue en une copie du couple d'arbres passés en argument en se contentant d'échanger les étiquettes de leurs racines.

Correction :

```
1 let echange_racines a1 a2 = match a1, a2 with
2   | Vide, _ | _, Vide -> a1, a2
3   | Noeud (r1, g1, d1), Noeud (r2, g2, d2) -> Noeud (r2, g1, d1), Noeud (
      r1, g2, d2)
```

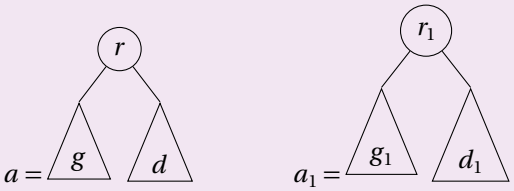
Question 15 : On considère une liste de tas non vide $h = \big((a_1, t_1), \dots, (a_r, t_r)\big)$ vérifiant la condition RO et un quasi-tas a de taille t. Montrer que :

1. si $\min_{\mathscr{A}}(a) \leq \min_{\mathscr{A}}(a_1)$, alors `(percole a, t)::h` est une liste de tas vérifiant RO ;

   Correction :
   `(percole a)` est un tas binaire parfait car `a` est un quasi-tas. Sa racine est donc son minimum et elle est inférieure à celle de `a1` par hypothèse. Donc `(percole a, t)::h` est une liste de tas vérifiant RO.
   Attention ici : la question n'est pas difficile, il faut d'autant plus la traiter avec soin. En particulier bannir tous les "`a` *se transforme en" ou "la racine de* `a` *devient" :* `a` *est* immuable, *donc ne peut pas se transformer. Il faut utiliser le vocabulaire adéquat. Si besoin vous pouvez introduire une nouvelle notation en désignant* `percole a` *par* `a'` *par exemple.*

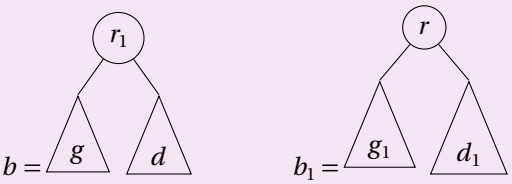
2. si $\min_{\mathscr{A}}(a) > \min_{\mathscr{A}}(a_1)$ et si on pose `(b,b1) = echange_racines a a1`, alors `b` est un tas binaire parfait, `b1` est un quasi-tas et $\min_{\mathscr{A}}(\texttt{b}) = \min_{\mathscr{A}}(a_1) \leq \min_{\mathscr{A}}(\texttt{b1})$.

   Correction :
   On dispose donc au départ de deux arbres : a qui est un quasi-tas et a_1 qui est un tas binaire parfait et dont la racine contient donc le minimum : $\min_{\mathscr{A}}(a_1) = r_1$.

   dans l'hypothèse où $\min_{\mathscr{A}}(a) \leq \min_{\mathscr{A}}(a_1)$,
   on sait que `percole a` est un tas binaire parfait (car `a` est un quasi-tas) et que sa racine qui est son minimum est donc inférieure à la racine de a_1. On en déduit que `(percole a, t)::h` est une liste de tas vérifiant RO.

   dans l'hypothèse où $\min_{\mathscr{A}}(a) > \min_{\mathscr{A}}(a_1)$,
   on construit les arbres b et b_1 :



- b est un quasi-tas car a est un quasi-tas et ils ne diffèrent que par la valeur de leurs racines.
  On a : $\min_{\mathscr{A}}(g) \geq \min_{\mathscr{A}}(a) > r_1$ et $\min_{\mathscr{A}}(d) \geq \min_{\mathscr{A}}(a) > r_1$. Comme g et d sont des tas parfaits de même taille (car a est un quasi-tas), on en déduit que b est un tas parfait.
- b_1 est un quasi-tas car a_1 est un tas parfait et ils ne diffèrent que par la valeur de leurs racines.
- b et a_1 étant des tas parfaits, leurs minimums respectifs sont à leurs racines respectives, on a donc : $\min_{\mathscr{A}}(b) = r_1 = min_{\mathscr{A}}(a_1)$.
  b_1 étant un quasi-tas, on a : $\min_{\mathscr{A}}(b_1) = \min(r, \min_{\mathscr{A}}(g_1), \min_{\mathscr{A}}(d_1)) \geq r_1$ car $r \geq \min_{\mathscr{A}}(a) > r_1$ par hypothèse et a_1 est un tas parfait.

Question 16 : On examine maintenant trois exemples de couples (a, h) pour lesquels a est un quasi-tas et h est une liste de tas non vide vérifiant la condition RO. On souhaite à chaque fois faire évoluer la liste de tas $(a, |a|) :: h$ jusqu'à obtenir une liste de tas vérifiant la condition RO, *en ne s'autorisant pour seules opérations que d'éventuelles permutations entre des étiquettes d'un même arbre ou entre des étiquettes de deux arbres distincts* (aucune modification de la forme ou de la taille d'aucun arbre en jeu n'est autorisée).
Les couples considérés sont notés (a_1, h_1), (a_2, h_2) et (a_3, h_3) avec $h_1 = \left((a_1^1, 7)\right)$, $h_2 = \left((a_2^1, 7)\right)$ et $h_3 = \left((a_3^1, 3), (a_3^2, 7)\right)$, où les arbres a_1, a_1^1, a_2, a_2^1 et a_3, a_3^1, a_3^2 sont donnés figure 2.

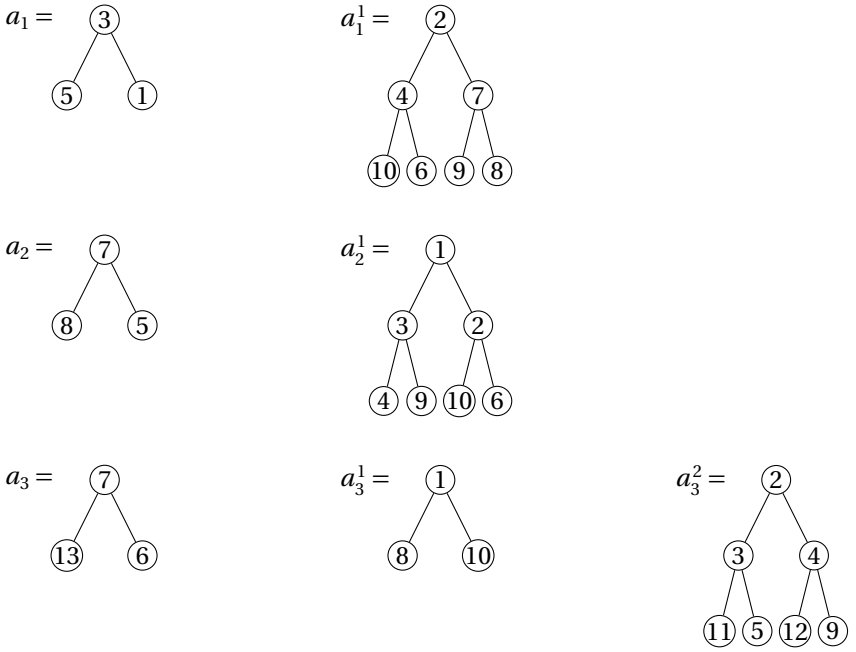
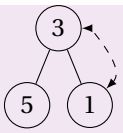


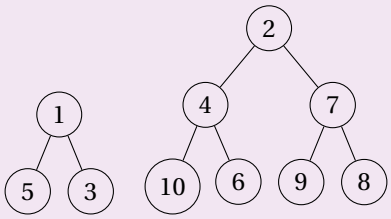
FIGURE 2 –

Pour chacun de ces trois couples, détailler (à l'aide de représentations graphiques) les étapes de la transformation de la liste $(a, |a|) :: h$ en une liste de tas vérifiant la condition RO. Chaque étape devra être clairement identifiée comme faisant appel à un procédé précédemment décrit.

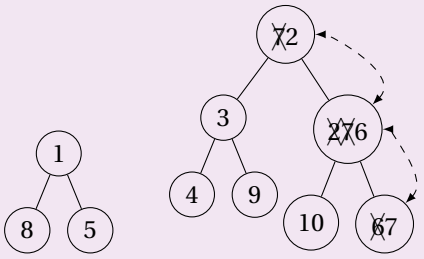
Correction :
Il s'agit ici de se servir de la question précédente pour faire cette question, et non de ré-inventer la roue (moins bien en général). Par ailleurs l'énoncé demande explicitement *de donner des représentations graphiques. Il faut répondre à la question et faire des dessins.*
Pour h_1 : $\min_{\mathscr{A}}(a_1) \leq \min_{\mathscr{A}}(a_1^1)$, on applique donc le premier cas en percolant a_1 :



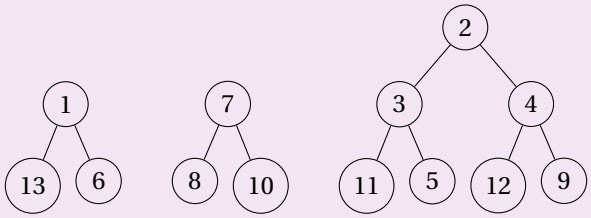
et on obtient :



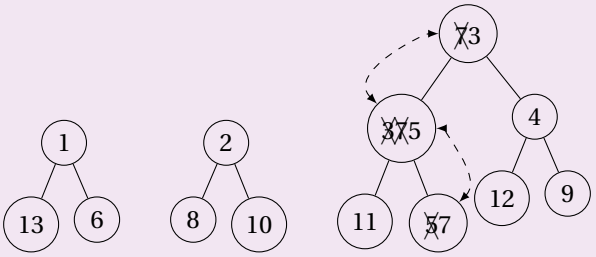
Pour h_2 : $\min_{\mathscr{A}}(a_2) > \min_{\mathscr{A}}(a_2^1)$. On commence donc par échanger les racines, puis on percole le deuxième arbre obtenu :



Pour h_3 : $\min_{\mathscr{A}}(a_3) > \min_{\mathscr{A}}(a_3^1)$, on échange donc les racines :



On constate alors que $\min_{\mathscr{A}}(a_3^{1'}) > \min_{\mathscr{A}}(a_3^2)$, et on échange les racines, puis on percole le dernier quasi-tas obtenu :



Question 17 : Décrire et justifier le plus précisément possible un algorithme qui, à partir d'un quasi-tas a, de sa taille t et d'une liste de tas h, s'évalue en une liste de tas h' identique à $(a,t) :: h$ à permutation près des étiquettes des arbres et telle que si h vérifie la condition RO alors h' vérifie la condition RO.

Correction :

Algorithme 1 insere_quasi

entrée : quasi-tas a de taille t, liste de tas $h = ((a_1, t_1), \ldots, (a_r, t_r))$ qui vérifie RO
sortie : liste de tas h' identique à $(a, t) :: h$, qui vérifie RO

si a est vide **alors renvoyer** h

si h est vide **alors renvoyer** `[(percole a, t)]`

si $\min_{\mathscr{A}}(a) \leq \min_{\mathscr{A}}(a_1)$ **alors renvoyer** `(percole a, t)::h`

si $\min_{\mathscr{A}}(a) > \min_{\mathscr{A}}(a_1)$ **alors**
`(b, b1)= echange_racines a a1`
renvoyer `b :: insere_quasi (b1, t1)h2`, où `h2` est obtenu en prenant tous les éléments de `h` excepté le premier.

Montrer que sa complexité en temps est $\mathcal{O}(1)$ si a est un tas non vide, que la liste de tas h vérifie la condition et que $\min_{\mathscr{A}}(a) \leq \min_{\mathscr{H}}(h)$.

Correction :
Si a est un tas non vide, h vérifie la condition RO et $\min_{\mathscr{A}}(a) \leq \min_{\mathscr{H}}(h)$, alors en particulier $\min_{\mathscr{A}}(a) \leq \min_{\mathscr{A}}(a_1)$ la complexité de cet algorithme est la complexité de la percolation de (a, t), qui se fait en temps constant car a est un tas.

Montrer que sa complexité en temps est $\mathcal{O}(k + r)$ où $k = \max\{\text{haut}(a), \text{haut}(h)\}$ et $r = \text{long}(h)$.

Correction :
Sinon, soit on percole a (en temps $\mathcal{O}(\text{haut}(a))$), soit on échange des racines en avançant tant que c'est nécessaire (au plus $\text{long}(h)$ fois), puis si besoin on percole (en temps $\mathcal{O}(\text{haut}(h))$), d'où une complexité en $\mathcal{O}(k + r)$ avec $k = \max\{\text{haut}(a), \text{haut}(h)\}$ et $r = \text{long}(h)$.

Question 18 : Écrire la fonction
`insere_quasi : arbre -> int -> (arbre * int)list -> (arbre * int)list`
telle que `insere_quasi a t h` s'évalue en la liste de tas vérifiant la condition RO construite par l'algorithme de la question précédente à partir d'un quasi-tas a de taille t et une liste de tas h vérifiant la condition RO.

Correction :
Il suffit de traduire l'algorithme de la question précédente en `OCaml`.

```
let rec insere_quasi a t h =
  if t = 0 then h
  else match h with
       | [] -> [ (percole a, t) ]
       | (a1, t1) :: h2 -> let min_a = min_quasi a in
                           let min_a1 = min_tas a1 in
                           if min_a <= min_a1 then (percole a, t) :: h
                           else let (b, b1) = echange_racines a a1 in
                                (b, t) :: insere_quasi b1 t1 h2
```

Question 19 : Écrire la fonction `tri_racines : (arbre * int)list -> (arbre * int)list` transformant une liste de tas h supposée vérifier la condition TC en une liste de tas h' vérifiant à la fois la condition RO et la condition TC et telle que h et h' aient les mêmes éléments.

Correction :
La fonction précédente permet d'insérer un quasi-tas dans une liste de tas vérifiant RO, donc en particulier elle peut être utilisée pour y insérer un tas. De plus elle ne modifie pas la forme des arbres, mais se contente de permuter des étiquettes, donc si la liste constituée du tas et de la liste de départ vérifie TC, ce sera également le cas de la liste résultat.

```
let rec tri_racines = function
  | [] -> []
```

```
3   | (a, t) :: h -> insere_quasi a t (tri_racines h)
```

Question 20 : Montrer que la fonction `tri_racines`, appliquée à une liste de tas h vérifiant la condition TC, a une complexité temporelle en $\mathcal{O}\big((\log|h|)^2\big)$.

Correction :
Il y a $\log(h)$ appels récursifs.. Chaque appel récursif donne lieu à une insertion, dont on sait que la complexité est en $\mathcal{O}(k+r)$ où $k = \max\{\text{haut}(a), \text{haut}(h)\}$ et $r = \log(h)$. Or d'après la question 11, comme h vérifie TC, on a $\log(a) \leq \log(h) = \mathcal{O}(\log|h|)$ et $\text{haut}(h) = \mathcal{O}(\log|h|)$, d'où le résultat.

2.6 Extraction des éléments d'une liste de tas

On souhaite dans cette sous-partie récupérer une liste d'étiquettes à partir d'une liste de tas vérifiant les propriétés précédentes. Soit $h = (\text{Noeud}(x, a_1, a_2), t) :: h'$ une liste de tas non vide vérifiant RO et TC. Pour supprimer x de h, si a_1 et a_2 ne sont pas vides, il suffit de construire $h'' = (\text{insere\_quasi } a_1\ |a_1|\ (\text{insere\_quasi } a_2\ |a_2|\ h'))$, où $|a_1|$ et $|a_2|$ peuvent se calculer en temps constant.

Question 21 : Montrer que h'' vérifie RO et TC.

Correction :
De façon immédiate, h' possède les propriétés RO et TC.
Par sa construction en question 18, la fonction `insere_quasi` appliquée à une liste de tas ayant la propriété RO renvoie une liste de tas ayant la propriété RO. On en déduit aisément que que h'' possède la propriété RO.
Si h possède la propriété TC, alors la taille de $\text{Noeud}(x, a_1, a_2)$ est inférieure ou égale à la taille du premier tas de h'. Si a_1 et a_2 sont vides, il en est donc de même de h''. Sinon leur taille commune est strictement inférieure à celle du premier tas de h', et h'' possède la propriété TC.

Question 22 : Donner la complexité temporelle de l'évaluation de $(\text{insere\_quasi } a_1\ |a_1|\ (\text{insere\_quasi } a_2\ |a_2|\ h'))$ dans le cas le pire, sous la forme $\mathcal{O}(f(|h|))$ pour une fonction f que l'on précisera.

Correction :
D'après la question 17, l'évaluation $\text{insere\_quasi } a_2\ |a_2|\ h'$ se fait en temps $\mathcal{O}(k+r)$, où $k = \max\{\text{haut}(a_2), \text{haut}(h')\}$ et $r = \log(h')$. Or H vérifie la propriété TC et a_2 est un sous-arbre du premier tas de h, donc $k = \text{haut}(h')$. Par ailleurs, h vérifiant TC, c'est également le cas de h', donc d'après la question 11.2, on a $k = \mathcal{O}(\log|h'|)$ et $r = \mathcal{O}(\log|h'|)$. On en déduit que l'évaluation $\text{insere\_quasi } a_2\ |a_2|\ h'$ se fait en temps $\mathcal{O}(|h'|)$.
Suite à cet appel, on obtient une liste de tas de taille $|h'| + |a_2| \leq |h|$. En appliquant le même raisonnement, on trouve donc une complexité temporelle pour $(\text{insere\_quasi } a_1\ |a_1|\ (\text{insere\_quasi } a_2\ |a_2|\ h'))$ en $\mathcal{O}(|h|)$, donc f est la fonction taille.

Question 23 : Écrire la fonction `extraire : (arbre * int)list -> int list` prenant en argument une liste de tas h vérifiant les conditions RO et TC et renvoyant la liste triée des éléments de h en utilisant les idées ci-dessus.

Correction :

```
1  let rec extraire h = match h with
2    | [] -> []
3    | (Noeud (x, Vide, Vide), 1) :: h' -> x :: extraire h'
4    | (Noeud (x, a1, a2), t) :: h' -> x :: extraire (insere_quasi a1 (t/2)
       (insere_quasi a2 (t/2) h'))
5    | _ -> failwith "situation impossible"
```

Question 24 : Montrer que la fonction `extraire`, appliquée à une liste de tas h vérifiant les conditions RO et TC, a une complexité temporelle en $\mathcal{O}(|h|\log|h|)$ dans le pire des cas.

Correction :
C'est une application directe de résultats précédents, je ne le fais, de même pour les questions de complexité de la partie suivante.

2.7 Synthèse

Question 25 : Écrire la fonction `tri_liste: int list -> int list` qui trie une liste en construisant une liste de tas intermédiaire vérifiant RO et TC avant d'en extraire les éléments.

Correction :

```
let tri_liste liste =
  let tc = constr_liste_tas liste in
  let ro = tri_racines tc in
  extraire ro
```

Question 26 : Montrer que la complexité temporelle dans le pire des cas de cette fonction est en $\mathcal{O}(n \log n)$ où n est la longueur de la liste donnée en argument.

Question 27 : Déterminer la complexité temporelle de la fonction `tri_liste` dans le cas particulier où la liste passée en argument est déjà triée.