Correction du devoir surveillé n°3

MPSI Lycée Clemenceau : option informatique

Du lundi 2 juin 2025

I Présentation

A) Motivation

Trier des données est un problème récurrent dans tous les systèmes dinformation. 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 sintéresse à la complexité dans le pire des cas.
Or, dans une application réelle, les données que lon 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 $\leqslant$ entre entiers.

B) Notations et préliminaires

Dans la suite, si un objet mathématique est noté t, on notera `t` l'objet Caml qui l'implante et, si t appartient à un ensemble T implanté en Caml par le type `T`, on écrira de manière équivalente $t \in T$ ou `t : T`. Par exemple, pour signifier que l désigne une liste dentiers, on notera `l : int list`.

Etant donné deux fonctions f et g à valeurs positives, on note :

- $f = O(g)$ pour exprimer qu'il existe une constante C telle que, pour tout n suffisamment grand, $f(n) \leq Cg(n)$;
- $f = \Omega(g)$ pour exprimer qu'il existe une constante $C > 0$ telle que pour, tout n suffisamment grand, $f(n) \geq Cg(n)$;
- $f = \Theta(g)$ pour exprimer qu'on a $f = O(g)$ et $f = \Omega(g)$.

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 Caml des listes ou des vecteurs dans lesquels un même élément peut apparaitre plusieurs fois, et des arbres dans lesquels la même valeur peut étiqueter plusieurs sommets 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 déléments. 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]`.

II Algorithme sur des arbres

A) Tri par insertion

Q.1) Ecrire la fonction `insere: int` $\to$ `int list` $\to$ `int list` insérant 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)` renvoie une liste `v` telle que

- `v` contient les mêmes éléments que `x::u` ;

- v est triée.

Correction : il suffit de chercher la position de l'élément dans la liste en parcourant la liste.

```
let rec insere x u =
   match u with
  |[] -> [x]
  |a::q -> if x<=a then x::u
           else a::(insere x q) ;;
```

Q.2) Ecrire la fonction `tri_insertion : int list → int list` triant la liste reçue en argument en utilisant la fonction précédente.

Correction : le principe est d'insérer la tête de la liste dans la queue qui a été triée auparavant (par un appel récursif)

```
let rec tri_insertion lst =
    match lst with
   |[] -> []
   |x::u -> insere x (tri_insertion u);;
```

Autre méthode : on crée une nouvelle liste triée à l'aide d'une fonction auxiliaire où l'on met les éléments en les triant.

```
let tri_insertion lst = let rec aux lst newlist = match lst with
          | [] -> newlist
          | t::q ->  aux q (insere t newlist)
          in aux lst [];;
```

Q.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 $P_I(n)$ et $M_I(n)$.

Correction : Dans le cas le meilleur, la fonction `insere` effectue une comparaison (quand la liste argument est non vide, 0 quand elle est vide) et dans le cas le pire elle en effectue $|u|$. On a donc

$$\forall n \geq 2,\ M_I(n) = 1 + M_I(n-1) \ \text{ et } \ \forall n \geq 1,\ P_I(n) = (n-1) + P_I(n-1)$$

Comme $P_I(0) = 0$ et $M_I(1) = 0$, une récurrence immédiate donne

$$\forall n \in \mathbb{N}^*,\ M_I(n) = n-1 \ \text{ et } \ \forall n \in \mathbb{N},\ P_I(n) = \sum_{k=0}^{n-1} k = \frac{n(n-1)}{2}$$

On a bien sûr $M_I(0) = 0$.

Dans la première version le pire des cas est atteint lorsque la liste initiale est triée par ordre décroissant, le meilleur des cas lorsque la liste initiale est déjà triée.

Dans la seconde version c'est le contraire.

B) Tas binaires

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

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

On définit la *hauteur* et la *taille* (appelée aussi nombre déléments) 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`.

`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 détiquettes et que chaque étiquette présente apparait 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 noeud 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`.

Etant donné un arbre non vide a, on note $\min_{\mathcal{A}}(a)$ le minimum des éléments qu'il contient.

Q.4) Pour $k \in \mathbb{N}$, on note m_k la taille d'un arbre binaire parfait de hauteur k. Déterminer m_k pour tout $k \in \mathbb{N}$. On justifiera la réponse en exprimant m_{k+1} en fonction de m_k.

Correction : On a $m_0 = 0$ (l'arbre vide est de taille nulle) et, comme les deux fils sont de même taille

$$\forall k \in \mathbb{N},\ m_{k+1} = 2m_k + 1$$

On en déduit, par exemple par récurrence, que

$$\forall k \in \mathbb{N},\ m_k = 2^k - 1$$

Q.5) Ecrire la fonction `min_tas : arbre → int` telle que pour tout tas binaire parfait `a` non vide, `(min_tas a)` renvoie $\min_{\mathcal{A}}(\texttt{a})$. On fera en sorte que la complexité soit constante.

Correction : Le minimum est à la racine

```
let min_tas (Noeud(x,a1,a2)) = x;;
```

Q.6) Ecrire la fonction `min_quasi : arbre → int` telle que pour tout quasi-tas `a`, `(min_quasi a)` renvoie $\min_{\mathcal{A}}(\texttt{a})$ en temps constant.

Correction : Les deux fils d'un quasi-tas sont simultanément vides ou non vides. Dans le second cas, le minimum est égal au plus petit de la racine et des racines des fils.

```
let min_quasi a = match a with
   | Vide -> failwith "arbre vide"
   | Noeud(x,Vide,Vide) ->  x
   | Noeud(x,a1,a2) -> min x (min (min_tas a1) (min_tas a2));;
```

Q.7) Ecrire la fonction `percole : arbre → arbre` telle que `(percole a)` renvoie `a` si `a` est l'arbre vide et, si `a` est un quasi-tas, renvoie un tas binaire parfait contenant les mêmes éléments. Donner la complexité dans le cas le pire en fonction de la hauteur k du quasi-tas `a`.

Correction : L'arbre est un tas si la racine x est plus petite que le minimum des racines des fils (quand elles existent). Dans ce cas, on renvoie l'arbre. Dans l'autre, on échange x avec la racine la plus petite et on percole dans l'arbre associé.

```
let rec percole a =
    match a with
   |Vide -> Vide
   |Noeud(x,Vide,Vide) -> a
   |Noeud(x,Noeud(x1,g1,d1),Noeud(x2,g2,d2)) ->
            if x<=(min x1 x2) then a
            else if x1<x2 then Noeud(x1,percole (Noeud(x,g1,d1)),Noeud(x2,g2,d2))
            else Noeud(x2,Noeud(x1,g1,d1),percole (Noeud(x,g2,d2)));;
```

L'appel récurrent se faisant sur un unique fils, on ne fait le parcours que d'une seule branche. A chaque étape, on effectue un nombre constant d'opération. La complexité est donc linéaire en fonction de la hauteur de l'arbre argument.

C) 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 **Q.4)**.

Etant donné un entier naturel r, on dit qu'un r-uplet $(k_1, \ldots, k_r)$ 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 $(k_1, \ldots, k_r)$
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 $(k_1, \ldots, k_r)$ 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 $(k_1, \ldots, k_r)$ vérifie la propriété QSC si et seulement si le r-uplet $(m_{k_1}, \ldots, m_{k_r})$ 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.

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

Correction : On a

$$6 = m_2 + m_2,\ 7 = m_3,\ 8 = m_1 + m_3,\ 9 = m_1 + m_1 + m_3,\ 10 = m_2 + m_3$$

$$27 = m_1 + m_1 + m_2 + m_3 + m_4,\ 28 = m_2 + m_2 + m_3 + m_4,\ 29 = m_3 + m_3 + m_4,\ 30 = m_4 + m_4,\ 31 = m_5$$

$$100 = m_2 + m_2 + m_5 + m_6,\ 101 = m_3 + m_5 + m_6$$

Q.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 : On suppose $n = m_{k_1} + \cdots + m_{k_r}$ avec $(k_1, \ldots, k_r)$ qui vérifie la propriété QSC. On distingue des cas.

- Si $r = 0$ alors $n = 0$ et $n + 1 = m_1$; ce cas ne correspond à la formule de l'énoncé qui n'a de sens que si $r \geq 1$.
- Si $r = 1$ alors $n = m_{k_1}$ et $n + 1 = m_1 + m_{k_1}$ ce qui correspond bien au second cas de l'énoncé.
- Si $r \geq 2$ et $k_1 \neq k_2$ alors $n + 1 = m_1 + m_{k_1} + \cdots + m_{k_r}$ ce qui correspond bien au second cas de l'énoncé.
- Sinon $r \geq 2$ et $k_1 = k_2$ et $n + 1 = m_1 + 2m_{k_1} + m_{k_3} + \cdots + m_{k_r} = m_{k_1+1} + (m_{k_3} + \cdots + m_{k_r})$.

Les formules de l'énoncé sont donc correctes (sauf si $n = 0$). Il reste à remarquer que dans les deux cas de l'énoncé, la décomposition donnée correspond bien à un uplet qui vérifie la propriété QSC, ce qui est immédiat.

Q.10) Ecrire la fonction `decomp_parf : int → int list` telle que, pour out entier naturel n, `(decomp_parf n)` renvoie 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 $O(n)$.

Correction : Une première fonction `transforme` modélise en termes de listes les transformations évoquées dans la question précédente. La fonction principale fait un appel à l'entier inférieur et transforme le résultat. On a vu ci-dessus que le cas $n = 0$ était douteux ; je l'ai donc mis à part dans la fonction.

```
let  decomp_parf n =
          let transforme lst =
                 match lst with
                      |[x] -> [1;x]
                      | x::y::q -> if x=y then (2*x+1)::q
                                   else 1::l
          in
          let rec aux n =
           if n=0 then []
           else if n=1 then [1]
                else transforme (aux (n-1))
          in aux n;;
```

La complexité $C(n)$ de cette fonction vérifie la relation $C(n) = C(n-1) + \Theta(1)$ donc $C(n) = \Theta(n)$.

D) Création d'une liste de tas

On appelle *liste de tas* une liste de couples de la forme (a, t) où a désigne un arbre 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 Caml par le type `(arbre * int) list`.

Etant donnée une liste de tas h de la forme précédente, on définit

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

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

- la *taille* de h, notée $|h|$, par

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

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

$$\text{haut}(h) = \begin{cases} 0 & \text{si } h \text{ est la liste vide} \\ \max(\text{haut}(h_1), \ldots, \text{haut}(h_r)) & \text{si } h = ((a_1, t_1), \ldots, (a_r, t_r)) \end{cases}$$

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

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

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 $h = ((a_1, t_1), \ldots, (a_r, t_r))$ *vérifie la condition TC* (pour "tas croissants") si le r-uplet d'entiers naturels $(t_1, \ldots, t_r)$ 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 $h = ((a, |a|))$ vérifie la condition TC.

Q.11) **a)** Si h est une liste non vide de tas, a-t-on nécessairement $\text{haut}(h)=O(\log_2(|h|))$? A-t-on nécessairement $\text{long}(h)=O(\log_2(|h|))$? Justifier

Correction : $h = ((a_1, t_1), \ldots, (a_r, t_r))$ une liste de tas. La hauteur de ce tas est égale à la hauteur de l'un des a_i. Sa taille est plus grande que la taille de $|a_i|$ qui vaut $2^{\text{haut}(a_i)}+1$. On a donc $2^{\text{haut}(h)} \leq |h|+1$ et donc $\text{haut}(h) = O(\log_2(|h|))$.

Considérons la liste de tas comportant r arbres réduits à leurs racines. La longueur de ce tas est r et sa taille est aussi égale à r. La longueur n'est donc en général pas dominée par le logarithme de la taille.

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

Correction : Montrons que le second des résultats précédents est valable quand la condition TC est vérifiée. Soit donc $h = ((a_1, t_1), \ldots, (a_r, t_r))$ une liste de tas croissants. La taille de a_i est égale à la somme des t_i. Comme $(t_1, \ldots, t_r)$ vérifie QSC, on a $t_1 \geq m_1$, $t_2 \geq m_1$, $t_3 \geq m_2$, ..., $t_r \geq m_{r-1}$ et donc

$$|h| \geq 1 + \sum_{k=1}^{r-1} m_k \geq 1 + \sum_{k=1}^{r-1} 2^k = 2^{r-1}$$

On en déduit que $r = \text{long}(h) \leq \log_2(|h|) + 1 = O(\log_2(|h|))$.

Q.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 $h = ((a_1, t_1), \ldots, (a_r, t_r))$ 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

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

mais qui ne vérifie peut-être plus la condition TC. L'objectif de cette question consiste à concevoir, en utilisant les outils mis en oeuvre dans les question 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.

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

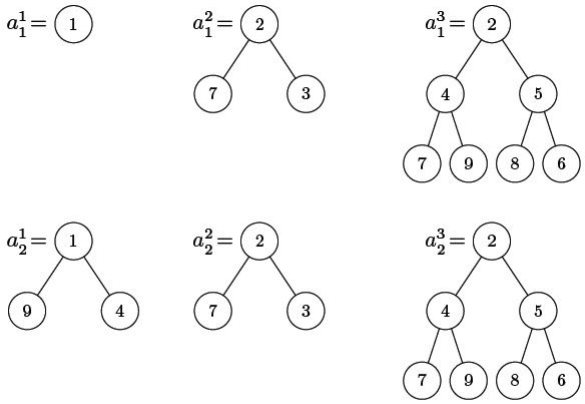


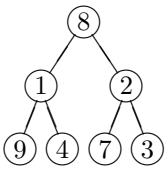
Figure 1

Expliquer de manière détaillée (à l'aide de représentation 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.

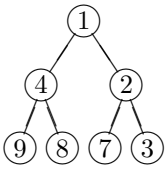
Correction : h_1 est de taille $11 = m_1 + m_2 + m_3$. h'_1 sera de taille $12 = m_1 + m_1 + m_2 + m_3$ et donc composée de 4 tas de tailles $1, 1, 3, 7$. Il suffit ici de poser

$$h'_1 = (a, 1) :: h_1 \quad \text{avec} \quad a = \texttt{Noeud}(\texttt{8}, \texttt{Vide}, \texttt{Vide})$$

h_2 est de taille $13 = m_2 + m_2 + m_3$ et h'_2 sera de taille $14 = m_3 + m_3$. a_2^1 et a_2^2 fusionnent avec l'arbre réduit à la racine 8. Dans un premier temps, cette fusion n'est qu'un quasi-tas représenté ci-dessous :



Il suffit alors d'utiliser l'opération de percolation pour obtenir



h'_2 est constituée de l'arbre ci-dessus et de a_2^3.

b) 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 $O(\text{haut}(a_1))$ (où a_1 est le premier tas de la liste h) et en $O(1)$ dans le cas le meilleur. on justifiera soigneusement la correction de la fonction et brièvement sa complexité dans le cas le pire.

Correction : Le processus général est le même et reprend l'idée de l'obtention d'une décomposition parfaite de $n + 1$ à partir de celle de n.

- Si h est vide ou de longueur 1, on pose $h' = (a, 1) :: h$.
- Sinon, notons (a_1, t_1) et (a_2, t_2) les deux premiers éléments de t. Si $t_1 < t_2$, on pose $h' = (a, 1) :: h$. Sinon $t_1 = t_2$ et on fusionne a, a_1, a_2 pour obtenir un quasi-tas (puisque a_1 et a_2 sont des tas de même taille) et on utilise la percolation. On obtient un tas a' de taille $t' = 1 + t_1 + t_2$ et on pose $h' = [(a', t'), (a_3, t_3), \dots]$.

Il est clair que la liste renvoyée a les mêmes éléments que $(a, 1) :: h$ et que c'est une liste de tas. Elle vérifie la condition TC car la suite des tailles vérifie QSC (c'est l'algorithme de décomposition parfaite qui le montre et cet algorithme a été prouvé).

Dans le meilleur cas, un consage suffit. Dans le pire, on utilise la percolation dont la complexité est linéaire en la hauteur du quasi-tas, c'est à dire aussi en la hauteur de a_1 (plus 1, ce qui ne change rien).

c) Ecrire la fonction `ajoute : int → (arbre * int) list → (arbre * int) list` telle que `(ajoute x h)` renvoie 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 : Il suffit de suivre fidèlement le programme précédent.

```
let ajoute x h =
     match h with
    |[] -> [Noeud(x,Vide,Vide),1]
    |[(a1,t1)] -> (Noeud(x,Vide,Vide),1)::h
    |(a1,t1)::(a2,t2)::q ->
               if t1<t2 then (Noeud(x,Vide,Vide),1)::h
               else (percole (Noeud(x,a1,a2)),1+t1+t2)::q ;;
```

Q.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) ;;
```

Il est clair que l'appel (`constr_liste_tas l`) renvoie une liste de tas vérifiant la condition TC et ayant les mêmes éléments que la liste `l`.

a) 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 $O(n)$.

Correction : Soit C_n le nombre d'opération dans le cas le pire dans l'appel `construit_tas l` pour `l` liste triée d'entiers. On a $C_{n+1} = C_n = \alpha$ où α est le coût de l'ajout de l'arbre de racine l_1 à une liste de tas d'étiquettes toutes $\geq l_1$. Cet ajout a un coût $O(1)$ sauf s'il induit une percolation. Mais si cette dernière a lieu, elle se fait en temps constant car, avec l'hypothèse sur les étiquettes, elle est appelée avec un tas (et non un quasi-tas). Ainsi $C_{n+1} = C_n + O(1)$. Comme $C_0 = O(1)$, on a bien $C_n = O(n)$.

b) Montrer que, pour une liste `l : int list` déjà triée de longueur n, `(constr_liste_tas l)` a une complexité temporelle en $O(n \log_2(n))$ dans le cas le pire. [1]

Correction : Dans l'appel `construit_tas l`, on effectue n ajouts avec des tas qui ont moins de n éléments et donc qui sont de hauteur au plus $\log_2(n)$. Un ajout se fait donc en temps inférieur à $O(\log_2(n))$ et le coût total est $O(n \log_2(n))$.

E) Tri des racines

On dit qu'une liste de tas $h = ((a_1, t_1), \ldots, (a_r, t_r))$ *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 : $\min_{\mathcal{A}}(a_1) \leq \cdots \leq \min_{\mathcal{A}}(a_r)$.

On considère une liste de tas $h = ((a_1, t_1), \ldots (a_r, t_r))$ 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.

Q.14) Ecrire 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)` renvoie le couple d'arbres passés en argument en se contentant d'échanger les étiquettes de leurs racines.

Correction : on suppose que les arbres ne seront jamais vides.

```
let echange_racines (Noeud(x1,g1,d1)) (Noeud(x2,g2,d2))=
    (Noeud(x2,g1,d1)),(Noeud(x1,g2,d2));;
```

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

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

Correction : La racine x du tas `percole a t` est inférieure à `a`. Dans le cas où $\min_{\mathcal{A}}(a) \leq \min_{\mathcal{A}}(a_1)$, `(percole a, t)::h` vérifie donc RO quand h le vérifie puisqu'alors $x \leq \min_{\mathcal{A}}(a) \leq \min_{\mathcal{A}}(a_1) \leq \ldots \min_{\mathcal{A}}(a_r)$.

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

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

Correction : Si $\min_{\mathcal{A}}(a_1) < \min_{\mathcal{A}}(a)$ alors dans l'opération `(b,b1)=(echange_racines a a1)`, `b` est comme `a` sauf la racine dont la valeur est diminuée. La contrainte d'ordre des étiquettes est donc préservée et `b` est un tas binaire parfait.

De même, `b1` est comme `a1` sauf la racine. La contrainte d'ordre des étiquettes est donc préservée pour les étiquettes sauf peut-être la racine et `b1` est un quasi-tas.

Enfin, on a placé en racine de `b` l'élément $\min_{\mathcal{A}}(a_1)$ qui plus petit que toutes les étiquettes de `a` et donc que toutes celles des fils de la racine de `b`. On trouve donc le minimum de `b` à la racine et il vaut $\min_{\mathcal{A}}(\texttt{b}) = \min_{\mathcal{A}}(a_1)$. Dans `b1`, on trouve des étiquettes de a_1 et $\min_{\mathcal{A}}(a)$ et toutes ces étiquettes sont plus grandes que $\min_{\mathcal{A}}(a_1)$. On a donc finalement

$$\min_{\mathcal{A}}(\texttt{b}) = \min_{\mathcal{A}}(a_1) \leq \min_{\mathcal{A}}(\texttt{b1})$$

Q.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 $h_1 = ((a_1^1, 7))$, $h_2 = ((a_2^1, 7))$ et $h_3 = ((a_3^1, 3), (a_3^2, 7))$, 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.

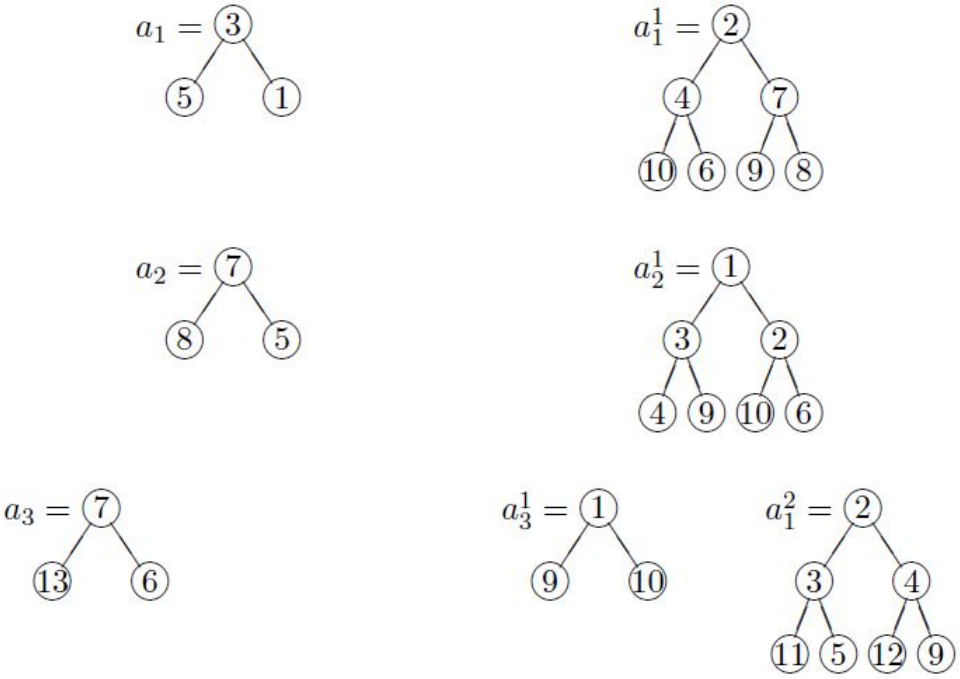


Figure 2

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.

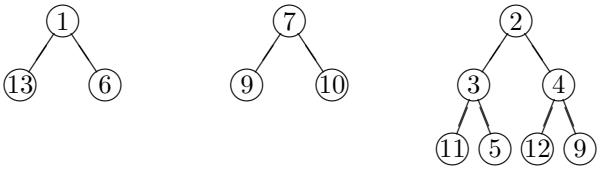
Correction : On remarque qu'une opération de percolation sur un quasi-tas est permise puisqu'elle n'induit que des échanges d'étiquettes dans le quasi-tas.

- Le minimum de a_1 étant plus petit que la racine de a_1^1 (qui est son minimum), une percolation dans a_1 suffit avec la question précédente.

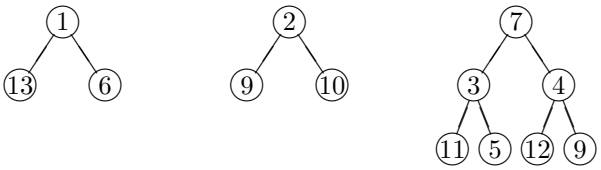
- Cette fois le minimum de a_2 est plus grand. On commence par échanger les racines

puis on effectue une percolation dans le second arbre

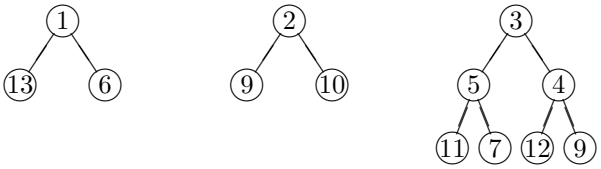
- On échange les racines des deux premiers arbres (car le minimum de a_3 est plus petit que celui de a_3^1)



On échange les racines des deux derniers arbres (car le minimum du second est plus petit que celui du troisième)



On effectue une percolation dans le troisième arbre



Q.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, renvoie une liste de tas h' identique à $(a,t) :: h$ à permutation près des étiquettes des arbres et tel que si h vérifie la condition RO alors h' vérifie la condition RO.

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

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

Correction : On distingue trois cas (deux cas de base et une étape récurrente pour un algorithme récursif).

- Si h est vide, il suffit de renvoyer la liste `[(percole a,t)]`.
- Si $h = (a_1, t_1) :: k$ avec $\min_{\mathcal{A}}(a_1) \geq \min_{\mathcal{A}}(a)$, on renvoie la liste `(percole a,t)::h`.
- Sinon $h = (a_1, t_1) :: k$ et on pose `(b,b1)=(echange_racines a a1)` puis on renvoie `(b,t)::(insere_quasi b1 t1 k)`.

Si a est un tas non vide et que l'étiquette de a est plus petite que toutes celles de h, on est dans l'un des deux premier cas mais la percolation ne coûte que $O(1)$ car a est un tas.La complexité est alors $O(1)$.
Dans le cas général, on effectue une unique percolation sur l'un des arbres et k échanges. Le coût est ainsi $O(k+r)$ où r est le maximum des hauteurs des arbres.

Q.18) Ecrire la fonction
`insere_quasi : arbre → int → (arbre * int) list → (arbre * int) list`
telle que `(insere_quasi a t h)` renvoie 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 : A ce niveau, les tailles ne servent pas (on ne cherche pas à imposer TC).

```
let rec insere_quasi a t h=
    match h with
   |[] -> [percole a,t]
   |(a1,t1)::k -> if (min_quasi a)<=(min_tas a1) then
                  (percole a,t)::h
```

```
    else begin
        let (b,b1)=echange_racines a a1 in
        (b,t)::(insere_quasi b1 t1 k)
        end;;
```

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

Correction : Il suffit d'utiliser ce qui précède avec le tas en tête de h et la version triée de la queue de h (processus récurrent, le cas de base est celui d'une liste vide). Le processus ne change pas les tailles et conserve la propriété TC.

```
let rec tri_racines h =
    match h with
   |[] -> []
   |(a1,t1)::k -> insere_quasi a1 t1 (tri_racines k);;
```

Q.20) Montrer que la fonction `tri_racines`, appliquée à une liste de tas h vérifiant la condition TC, a une complexité temporelle en $O((\log_2 |h|)^2)$.

Correction : Quand h est une liste de tas de taille n qui vérifie TC, on a haut$(h) = O(\log_2(n))$ et long$(h) = O(\log_2(n))$ (question 1.D.1). Dans l'appel `tri_racines h`, aucun des arbres mis en jeu ne change de forme. Ils sont donc tous et toujours de hauteur $O(\log_2(n))$. On effectue long(h) appels à `insere_quasi` qui sont tous de complexité $O(\log_2(n))$. On a finalement une complexité $O((\log_2(n))^2)$.

F) Extraction des éléments d'une liste de tas

On souhaite dans cette sous-partie récupérer une liste détiquettes à partir dune liste de tas vérifiant les propriétés précédentes. Soit $h = (\texttt{Noeud}(x, a_1, a_2), t) :: h'$ 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 $h'' = (\texttt{insere\_quasi}\ a_1\ |a_1|\ (\texttt{insere\_quasi}\ a_2\ |a_2|\ h'))$, où $|a_1|$ et $|a_2|$ peuvent se calculer en temps constant.

Q.21) Montrer que h'' vérifie RO et TC.

Correction : $(a_2, |a_2|) :: h'$ vérifie la condition TC car la taille de a_2 est strictement plus petite que t. L'appel (`insere_quasi` $a_2\ |a_2|\ h'$) renvoie donc une liste h_1 qui vérifie TC (comme expliqué en 1.E.6). De plus, comme h' vérifie RO, h_1 vérifie aussi RO (propriété de la fonction `insere_quasi`).
a_1 est un tas de même taille que le premier élément de h_1. Comme on vient de le voir, le premier tas de h_1 a la même taille $|a_1| = |a_2|$ mais les suivants ont des tailles strictement plus grandes. Ainsi $(a_1, |a_1|) :: h_1$ vérifie TC (on est dans le cas de QSC pour les tailles). Comme juste avant, l'appel (`insere_quasi` $a_1\ |a_1|\ h_1$) gardera les propriétés TC et RO.

Q.22) Donner la complexité temporelle de lévaluation de (`insere_quasi` $a_1\ |a_1|$ (`insere_quasi` $a_2\ |a_2|\ h'$)) dans le cas le pire, sous la forme $O(f(|h|))$ pour une fonction f que lon précisera.

Correction : Comme dans la question **Q.20)**, les deux appels à `insere_quasi` auront un coût $O(\log_2 |h|)$.

Q.23) Ecrire la fonction `extraire : (arbre * int) list` $\rightarrow$ `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 : Dans le cas récursif (liste non vide), on récupère le minimum (racine du premier tas) que l'on conse au résultat de l'appel sur la liste de tas h' obtenue en supprimant ce minimum. L'énoncé donne la formule pour h' dans le cas où le premier tas est de taille > 1 (condition a_1 et a_2 non vides). Le cas où a_1 et a_2 sont vides est immédiat.

```
let rec extraire h =
    match h with
   |[] -> []
   |((Noeud(x,Vide,Vide)),1)::k ->x::(extraire k)
   |((Noeud(x,a1,a2),t))::k ->
              x::(extraire (insere_quasi a1 (t/2) (insere_quasi a2 (t/2) k)));;
```

Q.24) Montrer que la fonction `extraire`, appliquée à une liste de tas h vérifiant les conditions RO et TC, a une complexité temporelleen $O(|h| \log_2 |h|)$ dans le pire des cas.

Correction : A chaque appel récursif, on diminue de 1 la taille de la liste de tase et on a par ailleurs $O(\log_2(|h|))$ opérations (question précédente). Le coût est donc globalement $O(|h| \log_2(|h|))$ puisque l'on a autant d'appels que d'étiquettes présentes dans la liste de tas.

G) Synthèse

Q.25) Ecrire la fonction `tri_lisse: int list → int list` qui trie une liste en construisant une liste de tas intermédiaire vérifiant RO et TC avant den extraire les éléments.

Correction : On combine toutes nos fonctions.

```
let tri_lisse l =
   extraire (tri_racines (constr_liste_tas l));;
```

Q.26) Montrer que la complexité de cette fonction est en $O(n \log_2 n)$ où n est la longueur de la liste donnée en argument.

Correction : La question 1D3 montre que la construction de la liste de tas a un coût $O(n \log_2(n))$. Le tri des racines a alors un coût négligeable $O((\log_2(n))^2)$. L'extraction ajoute un coût $O(n \log_2(n))$. On a finalement un coût quasi-linéaire en la taille de la liste.

Q.27) Déterminer la complexité temporelle de la fonction `tri_lisse` dans le cas particulier où la liste passée en argument est déjà triée.

Correction : Dans le cas d'une liste triée, on a vu que la construction coûte $O(n)$. Lors de cette construction, le dernier élément de la liste est le premier à être placé dans un tas puis on ajoute l'avant denier etc. Le dernier tas de la liste de tas contient donc les dernier éléments de la liste, puis l'avant dernier les précédente etc. Dans le cas de notre liste triée, tous les éléments du premier tas seront inférieurs à tous ceux du second eux mêmes plus petits que tous ceux du troisième etc. Quand on va appeler `inder_quasi` on sera toujours dans le premier cas de 1E4 et le coût sera constant. L'extraction des n éléments aura donc un coût $O(n)$. $(\log_2 n)^2$ étant négligeable devant n, le coût est linéaire en fonction de n.