## 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.

## 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 d'entiers, 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 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]`.

## Partie I - Tri par insertion

1) Écrire la fonction `insere: int → int list → 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.

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

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 $M_I(n)$.

## Partie II - Tas binaires

On appelle *arbre* un arbre binaire étiqueté par des éléments de $\mathbb{N}$. Un tel arbre est implanté en Caml 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 $\mathrm{haut}(a)$ et $|a|$ par induction sur la structure de l'arbre :

- $\mathrm{haut}(\mathtt{Vide})=0$ et $\mathrm{haut}(\mathtt{Noeud(x,a1,a2)})=1+\max\{\mathrm{haut}(\mathtt{a1}),\mathrm{haut}(\mathtt{a2})\}$.
- $|\mathtt{Vide}| = 0$ et $|\mathtt{Noeud(x,a1,a2)}| = 1 + |a1| + |a2|$

pour tous arbres binaires `a1` et `a2` et tout entier `x`.

Dans ce cas, `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 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 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`.

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

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 $m_k$.

5) Écrire la fonction `min_tas : arbre` $\rightarrow$ `int` telle que pour tout tas binaire parfait `a` non vide, l'instruction `(min_tas a)` renvoie $\min_{\mathcal{A}}(\mathtt{a})$. On fera en sorte que la complexité soit constante.

6) Écrire la fonction `min_quasi : arbre` $\rightarrow$ `int` telle que pour tout quasi-tas `a`, l'instruction `(min_quasi a)` renvoie $\min_{\mathcal{A}}(\mathtt{a})$ en temps constant.

7) Écrire la fonction `percole : arbre` $\rightarrow$ `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`.

## Partie III - 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).

Etant 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 $(k_1, \ldots, k_r)$ 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.

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

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

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

10) Écrire 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 $O(n)$.

## Partie IV - 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 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.

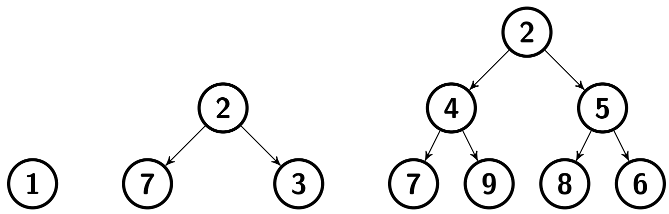
On dit qu'une liste de tas $h = ((a_1, t_1), \ldots, (a_r, t_r))$ *vérifie la condition TC* (pour "tas croissants") si le $r$-uplet d'entiers naturels $(t_1, \ldots, t_r)$ vérifie la propriété QSC. On peut remarquer qu'une liste de tas vérifie la condition TC si et seulement si $|h| = t_1 + \cdots + t_r$ 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.

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

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

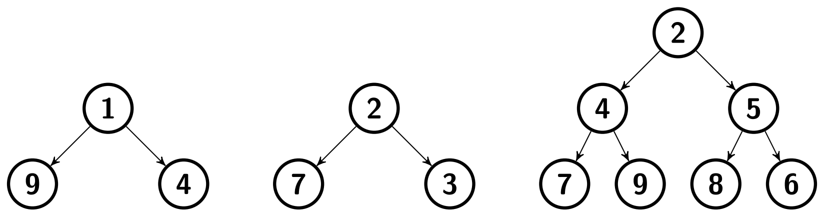
12) Considérons un arbre réduit à sa racine (c'est à dire un couple $(a, 1)$ correspondant à un tas binaire parfait) et une liste de tas $h = ((a_1, t_1), \ldots, (a_r, t_r))$ vérifiant la condition TC.
Si l'on ajoute le couple $(a, 1)$ en tête de la liste $h$, on obtient bien une liste de tas
$(a, 1)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



$a_1^1$ $a_1^2$ $a_1^3$

et



$a_2^1$ $a_2^2$ $a_2^3$

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 $h_2$.

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.

c) Écrire 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.

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) ;;
```

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 $O(n)$.

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

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