Sujet Centrale 2016 : correction

II. A. 1)

```
let insere x u = match u with
   | [] -> [x]
   | t::_ when x<=t -> x::u
   | t::q -> t:: (insere x q) ;;
```

II. A. 2)

```
let tri_insertion u = match u with
   | [] -> u
   | t::q -> insere t (tri_insertion q);;
```

II. A. 3) Pour insérer un élément avec la fonction "insere" dans une liste `u` de taille n, on fait au mieux une comparaison (pour une liste non vide) et zéro pour une liste vide, au pire n comparaisons. Pour tout $n \geq 1$, on a donc $P_I(n+1) = P_I(n)+n$ et $M_I(n+1) = M_I(n)+1$. Ainsi, pour tout $n \geq 1$, $P_I(n) = \sum_{i=1}^{n-1} i + P_I(1) = \frac{n(n-1)}{2} + 0 = \frac{n(n-1)}{2}$ et $M_I(n) = \sum_{i=1}^{n-1} 1 + M_I(1) = n-1$.
II. B. 1) Pour tout $k \in \mathbb{N}$, $m_{k+1} = 2m_k + 1$ et $m_0 = 0$. On a donc, pour tout $k \in \mathbb{N}$, $m_{k+1} + 1 = 2m_k + 2 = 2(m_k + 1)$ donc, pour tout $k \in \mathbb{N}$, $m_k + 1 = 2^k \times (m_0 + 1) = 2^k$ donc $m_k = 2^k - 1$.
II. B. 2)

```
let min_tas a = let Noeud(x,g,d)=a in x;;
```

II. B. 3)

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

II. B. 4)

```
let rec percole a = match a with
   | Vide -> Vide
   | Noeud(x,Vide,Vide) -> a
   | Noeud(x,Noeud(xg,gg,dg),Noeud(xd,gd,dd)) ->
         if x<= min xg xd then a
         else if xg<xd then Noeud(xg,percole Noeud(x,gg,dg),Noeud(xd,gd,dd))
         else Noeud(xd,Noeud(xg,gg,dg),percole Noeud(x,gd,dd));;
```

La complexité C_k dans le pire des cas de cette fonction pour un quasi-tas de hauteur k vérifie, pour tout k, $C_{k+1} = O(1) + C_k$ (avant d'utiliser un appel récursif sur un arbre de hauteur un de moins, on effectue un nombre borné de comparaisons ou de construction d'arbres). On a donc $C_k = O(k)$.
II. C. 1) $m_1 = 1$, $m_2 = 3$, $m_3 = 7$, $m_4 = 15$, $m_5 = 31$, $m_6 = 63$.
$6 = m_2 + m_2$, $7 = m_3$, $8 = m_1 + m_3$, $9 = m_1 + m_1 + m_3$, $10 = m_2 + m_3$, $27 = m_1 + m_1 + m_2 + m_3 + m_4$, $28 = m_2 + m_2 + m_3 + m_4$, $29 = m_3 + m_3 + m_4$, $30 = m_4 + m_4$, $31 = m_5$, $100 = m_2 + m_2 + m_5 + m_6$, $101 = m_3 + m_5 + m_6$.
II. C. 2) Supposons que $r \geq 2$ et $k_1 = k_2$. Alors $k_1 = k_2 < k_3 < \cdots < k_r$. On a donc $k_1 + 1 \leq k_3 < k_4 < \cdots < k_r$. Le $r-1$-uplet $(k_1 + 1, k_3, \cdots, k_r)$ vérifie bien la condition QSC. De plus,

$$n = \sum_{i=1}^{r} m_{k_i} = m_{k_1} + m_{k_2} + \sum_{i=3}^{r} m_{k_i} = 2m_{k_1} + \sum_{i=3}^{r} m_{k_i}.$$

Comme $2m_{k_1} + 1 = m_{k_1+1}$, on obtient $n + 1 = 2m_{k_1} + 1 + \sum_{i=3}^{r} m_{k_i} = m_{k_1+1} + \sum_{i=3}^{r} m_{k_i}$. Ainsi, $n+1$ admet la décomposition parfaite $n = m_{k_1+1} + m_{k_3} + \cdots + m_{k_r}$.
Supposons $r < 2$. Alors $n = m_{k_1}$ donc $n + 1 = m_1 + m_{k_1}$ et $1 \leq k_1$; le couple $(1, k_1)$ vérifiant la condition QSC, $n + 1 = m_1 + m_{k_1}$ est une décomposition parfaite.
Supposons $r \geq 2$ et $k_1 < k_2$. Alors $1 \leq k_1 < k_2 < \cdots < k_r$. Le $r+1$-uplet $(1, k_1, \cdots, k_r)$ vérifie la condition QSC et $n + 1 = 1 + m_{k_1} + \cdots + m_{k_r} = m_1 + m_{k_1} + \cdots + m_{k_r}$ est donc une décomposition parfaite.
II. C. 3)

```
let rec decomp_parf n = match n with
   | 1 -> [1]
   | n-> let t::q=decomp_parf (n-1) in
            if q=[] then [1;t] else let z::q1=q in
            if y=z then (2*y+1)::q1 else 1::(t::q);;
```

La complexité C_n vérifie $C_n = C_{n-1} + O(1)$ donc $C_n = O(n)$.
II. D. 1)
a) Prenons des tas $(a_1, t_1), \cdots (a_n, t_n)$ tous de tailles $m_1 = 1$ pour en faire une liste$h$ (de tas). La taille est $|h| = 1 + \cdots + 1 = n$ donc $\log_2(|h|) = \log_2(n)$; de plus, $\texttt{long}(h) = n$. La relation $n = O(\log_2(n))$ est fausse donc la relation $\texttt{long}(h) = O(\log_2(|h|))$ est fausse.
En revanche, pour une liste $h = ((a_1, t_1), \cdots, (a_r, t_r))$, pour tout i, $t_i \leq |h|$ donc $\log_2(t_i) \leq \log_2(|h|)$. De plus, un tas a_i de hauteur k_i a taille $t_i = m_{k_i} = 2^{k_i} - 1$; on a donc $k_i = \texttt{haut}(a_i) = k_i = O(\log_2(t_i)) = O(\log_2(|h|))$. On a donc $\texttt{haut}(h) = \max(k_i) = O(\log_2(|h|))$.
b) Dans le cas d'une liste de cas vérifiant la condition TC, si la liste h est de longueur r, les tailles $t_1, \cdots, t_r$ forment une décomposition parfaite de $|h|$. Ainsi, il existe une suite $(k_1, \cdots, k_r)$ telle que $t_i = m_{k_i}$. Chaque arbre de de hauteur k_i. La suite $(k_1, \cdots, k_r)$ étant QSC, la hauteur de cette liste h, le maximum des hauteurs successives, est k_r. La taille de cette liste est $|h| = \sum_{i=1}^{r} m_{k_i} \geq m_{k_r} = 2^{k_r} - 1$; par quasi stricte croissance de la suite (k_i), $k_r \geq r-1$ donc $|h| \geq 2^{r-1} - 1$. Ainsi, $\log_2(|h|+1) \geq \log_2(2^{r-1})$ donc $\texttt{long}(h) = r \leq \log_2(|h|)+1$ donc $\texttt{haut}(h) = O(\log_2(|h|))$.
II. D. 2)
a) Pour la liste h_1, il suffit de considérer la liste $((a, 1), (a_1^1, 1), (a_1^2, 3), (a_1^3, 7))$ où a a l'étiquette 8.
Pour la liste h_2, on percole le quasi-tas de sous-arbre gauche a_2^1, de sous-arbre droit a_2^2 et de racine 8. Cette percolation va construire l'arbre

$$\texttt{Noeud}(1, \texttt{Noeud}(4, \texttt{Noeud}(9, \texttt{Vide}, \texttt{Vide}), \texttt{Noeud}(8, \texttt{Vide}, \texttt{Vide})), \texttt{Noeud}(2, \texttt{Noeud}(7, \texttt{Vide}, \texttt{Vide}), \texttt{Noeud}(3, \texttt{Vide}, \texttt{Vide})))$$

que l'on notera b_2^1. La liste de tas obtenue est $((b_2^1, 7), (a_2^3, 7))$.
b) S'il y a au moins deux tas dans la liste et que les deux premiers tas a_1 et a_2 sont de même hauteur, on percole le quasi-tas ayant pour racine l'étiquette du nouvel arbre, sous-arbre gauche a_1 et sous-arbre droit a_2.
Les éléments des deux arbres et du nouvel arbre sont les mêmes que ceux de l'arbre percolé. Les éléments de la liste de tas sont donc ceux de l'ancienne liste de tas à laquelle on ajoute l'étiquette du nouvel arbre.
De plus, la suite des tailles est une décomposition parfaite de $|h| + 1 = |h'|$ d'après la question II.C.2).
Dans les autres cas, on ajoute le tas $(a, 1)$ en début de liste. Les éléments de la liste sont manifestement les "bons" et la suite des tailles est une décomposition parfaite de $|h| + 1 = |h'|$ d'après la question II.C.2).
Dans ce dernier cas, la complexité est en $O(1)$. Dans l'autre cas, la complexité est cella de la fonction de percolation d'un quasi-tas dont la hauteur est $\text{haut}(a_1) + 1$, c'est-à-dire $O(\texttt{haut}(a_1))$ d'après la question II.B.4).
c)

```
let ajoute n l = match l with
   | [] -> [(Noeud(n,Vide,Vide),1)]
   | x::Vide -> [(Noeud(n,Vide,Vide),1),x]
   | x::y::q when snd(x)<snd(y) -> (Noeud(n,Vide,Vide),1)::l
   | x::y::q -> (percole(Noeud(n,fst(x),fst(y))),1+2*snd(x))::q;;
```

II. D. 3)
a) Pour une liste triée, la fonction de percolation, utilisée dans la fonction `ajoute`, sera en $O(1)$ (puisque le quasi-tas est en fait un tas). Chaque utilisation de la fonction ajoute sera donc en $O(1)$. En notant C_n la complexité dans le pire des cas pour une liste de taille n triée vérifie donc $C_{n+1} = O(1) + C_n$ pour tout n. Ainsi, $C_n = O(n)$.
b) Dans le pire des cas, la complexité de la fonction `ajoute` est $O(\texttt{haut}(a_1)) = O(\texttt{haut}(h)) = O(\log_2(|h|))$ pour une liste de tas h.
La complexité C_n dans le pire des cas pour la fonction sur une liste de taille n vérifie, pour tout n, $C_{n+1} = C_n + O(\log_2(n))$. Par conséquent, $C_n = \sum_{i=1}^{n} O(\log_2(i)) = O(n \log_2(n))$.
II. E. 1)

```
let echange_racines a1 a2 =
     let Noeud(x1,g1,d1)=a1 and Noeud(x2,g2,d2)=a2 in Noeud(x2,g1,d1),Noeud(x1,g2,d2);;
```

II. E. 2)
a) On a bien $\min_{\mathcal{A}}(a) \leq \min_{\mathcal{A}}(a_1) \leq \cdots \leq \min_{\mathcal{A}}(a_r)$. La condition RO est bien vérifiée. De plus la liste obtenue est une liste de tas.
b) Dans ce cas, l'arbre `b` a pour racine $\min_{\mathcal{A}}(a_1)$; celle-ci étant inférieure strictement aux étiquettes de `a`, l'arbre `b`, dont les étiquettes sont celles de la `a` sans sa racine, a bien comme racine son plus petit élément ; les sous-arbres gauche et droit étant ceux de `a`, ce sont des tas binaires parfaits de même hauteur ; `b` est donc un tas binaire parfait.
Les sous-arbres droit et gauche de `b1` sont ceux de `a1` : ce sont donc deux tas binaires parfaits de même hauteur. Par conséquent `b1` est un quasi-tas. Enfin, $\min_{\mathcal{A}}(\texttt{b}) = \min_{\mathcal{A}}(\texttt{a1})$. De plus $\min_{\mathcal{A}}(\texttt{b1}) = \min(\min_{\mathcal{A}}(\texttt{a}), \min_{\mathcal{A}}(\texttt{a1prime}))$ où `a1prime` est la réunion des étiquette des sous-arbres droit et gauche de `a1`. Or $\min_{\mathcal{A}}(\texttt{a1}) < \min_{\mathcal{A}}(\texttt{a})$ et $\min_{\mathcal{A}}(\texttt{a1}) \leq \min_{\mathcal{A}}(\texttt{a1prime})$ donc $\min_{\mathcal{A}}(b) \leq \min_{\mathcal{A}}(\texttt{b1})$.
II. E. 3) Pour le couple (a_1, h_1), on percole l'arbre a_1 en un arbre a_1'. La liste (a_1', a_1^1) convient.
Pour le couple (a_2, h_2), on échange les racines de a_2 et a_2^1. On note b_2 et b_2^1 les arbres obtenus. Ensuite, on percole le quasi-tas b_2^1.
L'arbre b_2 est

$$\texttt{Noeud}(1, \texttt{Noeud}(8, \texttt{Vide}, \texttt{Vide}), \texttt{Noeud}(5, \texttt{Vide}, \texttt{Vide}))$$

L'arbre obtenu en percolant b_2^1 est

$$\texttt{Noeud}(2, \texttt{Noeud}(3, \texttt{Noeud}(4, \texttt{Vide}, \texttt{Vide}), \texttt{Noeud}(9, \texttt{Vide}, \texttt{Vide})), \texttt{Noeud}(6, \texttt{Noeud}(10, \texttt{Vide}, \texttt{Vide}), \texttt{Noeud}(7, \texttt{Vide}, \texttt{Vide})))$$

Cette liste convient.
Pour la dernière liste, on échange les deux premières racines (7 et 1), on note b_3, b_3^1 les deux nouveaux arbres. Ensuite, on échange les racines de b_3^1 et a_3^2. On note c_3^1 et c_3^2 les deux nouveaux arbres. Enfin, on percole le quasi-tas c_3^2.
L'arbre b_3 est

$$\texttt{Noeud}(1, \texttt{Noeud}(13, \texttt{Vide}, \texttt{Vide}), \texttt{Noeud}(6, \texttt{Vide}, \texttt{Vide}))$$

L'arbre c_3^1 est

$$\texttt{Noeud}(2, \texttt{Noeud}(8, \texttt{Vide}, \texttt{Vide}), \texttt{Noeud}(10, \texttt{Vide}, \texttt{Vide}))$$

L'arbre c_3^2 est

$$\texttt{Noeud}(3, \texttt{Noeud}(5, \texttt{Noeud}(11, \texttt{Vide}, \texttt{Vide}), \texttt{Noeud}(7, \texttt{Vide}, \texttt{Vide})), \texttt{Noeud}(4, \texttt{Noeud}(12, \texttt{Vide}, \texttt{Vide}), \texttt{Noeud}(9, \texttt{Vide}, \texttt{Vide})))$$

La liste (b_3, c_3^1, c_3^2) convient.
II. E. 4) Notons $h = ((a_1, t_1) \cdots, (a_r, t_r))$ une liste de tas vérifiant RO.
Si $\min_{\mathcal{A}}(a) \leq \min_{\mathcal{A}}(a_1)$, on percole a et on l'ajoute au début de la liste h. D'après II. E. 2) a), la liste obtenue est une liste de tas vérifiant RO.
Sinon, on échange les racines de a et a_1 ; on obtient une liste $((b,t),(b_1,t_1),(b_2,t_2)\cdots,(b_r,t_r))$ où $(b_i,t_i) = (a_i,t_i)$ si $i \geq 2$. La liste obtenue vérifie $\min_{\mathcal{A}}(b) \leq \min_{\mathcal{A}}(b_1)$, b est un tas et b_1 un quasi-tas. On a même pour tout $i \geq 2$, $\min_{\mathcal{A}}(b) \leq \min_{\mathcal{A}}(b_i)$.
Si $r = 1$, on percole l'arbre b_1. On obtient ainsi une liste de deux tas vérifiant encore la condition RO.
Si $r \geq 2$, on effectue récursivement la transformation sur la liste $((b_1,t_1),\cdots,(b_r,t_r))$; on ajoute à cette liste de tas vérifiant RO le tas (b,t). La liste obtenue vérifie encore la condition RO.
Lorsque a est non vide, que h vérifie RO et $\min_{\mathcal{A}}(a) \leq \min_{\mathcal{H}}(h)$, on ajoute (a,t) en début de liste et il n'est pas nécessaire de le percoler car a est déjà un tas.
La complexité est donc $O(1)$.
De façon générale, on parcourt une partie de la liste en comparant les minimum d'un tas et d'un quasi-tas à chaque étape du processus récursif ainsi qu'un échange de racines ; puis on effectue une percolation ; cette percolation faite, l'algorithme est terminé.
Le parcours de la liste effectuant comparaisons et échanges de racines se fera en $O(r)$. La percolation se fera en $O(k)$. La complexité totale est donc $O(k+r)$.
II. E. 5)

```
let rec insere_quasi a t h = match h with
   | Vide ->[(percole a,t)]
   | (a1,t1)::h1 when min_quasi(a)<=min_tas(a1) -> (percole a,t)::h
   | (a1,t1)::h1 -> let (b,b1)=echange_racines a a1 in (b,t)::(insere_quasi b1 t1 h1);;
```

II. E. 6)

```
let rec tri_racines h = match h with
   | Vide -> Vide
   | (a,t)::h1 -> insere_quasi a t (tri_racines h1);;
```

II. E. 7) La complexité de `insere_quasi` est $O(k+r)$ avec les notations déjà introduites. Comme $r = O(\log_2(|h|))$ et $h = O(\log_2(|h|))$ d'après II. D. 1), lors de l'utilisation de `tri_racines`, l'appel de `insere_quasi` se fait en $O(\log_2(|h|))$. La fonction `tri_racines` effectue un appel à `insere_quasi` puis un appel récursif. Celle-ci effectuera donc r appels à `insere_quasi`. La compexité est donc $O(r\log_2(|h|))$. Comme $r = O(\log_2(|h|))$, la complexité de `tri_racines` est $O(\log_2(|h|)^2)$.
II. F. 1) h' vérifie RO et TC. Comme la condition TC est vérifiée par h, $|a_1| = |a_2| < |a| \leq |a_3|$ (où a_3 est le premier arbre de h'), la condition TC sera vérifiée après les deux appels de `insere_quasi`. D'après les questions II. E., la condition RO sera vérifiée après les appels à `insere_quasi`. h'' vérifie donc RO et TC.
II. F. 2) Le premier appel `insere_quasi` se fait en $O(r+k) = O(\log_2(|h|))$. De même pour le deuxième appel. La complexité est donc $O(\log_2(|h|))$.
II. F. 3)

```
let rec extraire h = match h with
   | Vide -> []
   | (Noeud(x,Vide,Vide),1)::h1 -> x::extraire h1
   | (Noeud(x,a1,a2),t)::h1 -> x::(extraire (insere_quasi a1 (t/2) (insere_quasi a2 (t/2) h1)));;
```

II. F. 4) Pour chaque étiquette de h, on effectuera deux appels à `insere_quasi` (qui se font en $O(\log_2(|h|))$), un nombre borné d'opérations et un appel récursif.
La complexité C_n dans le pire des cas pour une liste de taille $|h| = n$ vérifie donc, pour tout n, $C_{n+1} = C_n + O(1) + O(\log_2(n))$. La complexité est donc en $O(n\log_2(n))$, c'est-à-dire $O(|h|\log_2(|h|))$.
II. G. 1)

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

II. G. 2) Pour une liste de taille n, la fonction de construction de la liste se fait en $O(n\log_2(n))$, le tri des racines en $O(\log_2(n)^2) = O(n\log_2(n))$, l'extraction en $O(n\log_2(n))$. La fonction de tri lisse se fait donc en $O(n\log_2(n))$.
II. G. 3) Pour une liste déjà triée, la construction de la liste se fait en $O(n)$ (cf. question II. D. 3) a)). Le tri des racines se fait en $O(\log_2(n)^2) = O(n)$. Pour une telle liste, les deux appels à `insere_quasi` effectueront directement des percolations sans parcourir la liste de tas ; ces percolations se feront en $O(1)$ car la liste d'origine est triée. Ces deux appels se font donc en $O(1)$. Au total, la complexité est en $O(n)$ pour une liste déjà triée.

III. A. Le tri lisse crée une liste de tas de longueur r. L'espace mémoire alloué est $r + \sum_{i=1}^{r} |a_i| = r + n = \Omega(n)$.

III. B.

```
let fg a = { donnees=a.donnes ; pos=a.pos+1 ; taille=a.taille/2 } ;;
let fd a = { donnees=a.donnes ; pos=a.pos+a.taille/2+1 ; taille=a.taille/2 } ;;
```

III. C.

```
let min_tas_vect a = a.donnes.(a.pos);;
let min_quasi_vect a =
   if a.taille=1 then a.donnes.(a.pos)
   else min (min (min_tas_vect (fg a)) (min_tas_vect (fd a))) a.donnes.(a.pos);;
```

III. D.

```
let echange t i j = let x=t.(i) in t.(i)<-t.(j); t.(j)<-x;;
let rec percole_vect a = match a.taille with
   | 0 -> ()
   | n -> if a.donnes(a.pos)<=a.donnees(a.pos+1) && a.donnes(a.pos)<=a.donnes(a.pos+a.taille/2+1)
            then ()
            else if a.donnees(a.pos+1)<=a.donnes(a.pos+a.taille/2+1)
                    then begin
                            echange a.donnees a.pos (a.pos+1);
                            percole_vect (fg a)
                            end;
                     else begin
                             echange a.donnes a.pos (a.pos+a.taille/2+1);
                             percole_vect (fd a)
                             end;;
```

III. E.

```
let ajoute_vect d p h = match h with
   | [] -> [{ donnes = d ; taille = 1 ; pos = p }]
   | [t] -> { donnes = d ; taille = 1 ; pos = p }::h
   | a1::a2::h1 when a1.taille<a2.taille -> {donnes = d ; taille = 1 ; pos = p}::h
   | a1::a2::h1 ->
   percole_vect { donnes = d ; taille = 1+2*a1.taille ; pos = p}; { donnees = d ; taille = 1+2*a1.taille ; pos=p}::h1;;
```

III. F.

```
let echange_racines a1 a2 =
   let x=a1.donnes.(a1.pos) in a1.donnes(a1.pos)<-a2.donnes(a2.pos);
   a2.donnes(a2.pos)<-x;;
```

III. G.

```
let rec insere_quasi_vect a h = match h with
   | [] -> percole_vect a; a::h
   | a1::h1 when min_quasi_vect a<=min_tas_vect a1-> percole_vect a; a::h1
   | a1::h1 -> echange_racines a a1; a::(insere_quasi_vect a1 h);;
```

III. H.

```
let rec tri_racines_vect h = match h with
   | [] -> []
   | a::h1 -> insere_quasi_vect a (tri_racines h1);;
```

III. I.

```
let rec extraire_vect h = match h with
   | [] -> ()
   | a::h1 when a.taille=1 -> extraire_vect h1
   | a::h1 -> extaire_vect (insere_quasi_vect (fg a) (insere_quasi_vect (fd a) h1);;
```

III. J.

```
let tri_lisse_vect t = extraire_vect (tri_racines_vect (constr_liste_tas_vect t)) ;;
```

III. K. L'algorithme est le même qu'avec des listes d'arbres ; le seul changement est l'espace mémoire alloué. Sa complexité temporelle est toujours $O(n \log_2(n))$ dans le pire des cas et $O(n)$ pour un tableau déjà trié.
III. L. Cf. question précédente.
III. M. Le champ "données" est le même, à savoir le tableau à trier. On stocke en plus r enregistrements contenants r positions et r tailles. La complexité spatiale est donc $O(r) = O(\log_2(n))$ (on utilise aussi localement des espaces pour effectuer les échanges mais en nombre borné).