MP* - OPTION INFORMATIQUE
Devoir surveillé 1 - corrigé

1 Algorithme sur des arbres

1.A. Tri par insertion

1.A.1
```
let rec insere x u =
    match u with
        [] -> [x]
        |a::q when x>a -> a::(insere x q)
        |_ -> x::u;;
```

1.A.2
```
let rec tri_insertion l =
    match l with
        [] -> []
        |a::q -> insere a (tri_insertion q);;
```

1.A.3 Dans le meilleur cas, la fonction `insere` effectue zéro comparaison si la liste est vide, et une sinon. Dans le pire cas, elle effectue $|u|$ comparaisons sur la liste 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 $M_I(0) = 0$.

1.B. Tas binaires

1.B.1 On a $m_0 = 0$ et $\forall k \in \mathbb{N},\ m_{k+1} = 2m_k + 1$
On en déduit par récurrence immédiate : $\forall k \in \mathbb{N},\ m_k = 2^k - 1$

1.B.2
```
let min_tas t =
    match t with
        Vide -> failwith "minimum d'un tas vide"
        |Noeud(x,g,d) -> x;;
```

1.B.3
```
let min_quasi t =
    match t with
        Vide -> failwith "minimum d'un quasi-tas vide"
        |Noeud(x,Vide,Vide) ->  x
        |Noeud(x,g,d) -> min x (min (min_tas g) (min_tas d));;
```

1.B.4
```
let rec percole a =
    match a with
        Vide -> a
        |Noeud(x,_,_) when x = min_quasi a -> a
        |Noeud(x,Noeud(x1,g1,d1),Noeud(x2,g2,d2)) ->
                if x1 = min_quasi a then
                    let g = percole (Noeud(x,g1,d1)) in
                    let d =  Noeud(x2,g2,d2) in
                    Noeud(x1,g,d)
```

```
else  let g = Noeud(x1,g1,d1) in
        let d = percole (Noeud(x,g2,d2)) in
        Noeud(x2,g,d);;
```

L'appel récursif se fait sur un seul fils, on parcourt donc au maximum une branche, avec une complexité à chaque étape en $O(1)$ La complexité de la fonction est donc en $O(k)$.

1.C. Décomposition parfaite d'un entier

1.C.1

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

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

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

1.C.2
— Si $r \geq 2$ et $k_1 = k_2$, on a $m_{k_1} + m_{k_2} = 2(2^{k_1} - 1) = m_{k_1+1} - 1$, d'où : $n + 1 = m_{k_1+1} + m_{k_3} + m_{k_4} + \ldots + m_{k_r}$.
Cette décomposition est parfaite car $k_1 + 1 \leq k_3 < k_4 < \ldots < k_r$.
— Si $r \leq 1$ ou $k_1 < k_2$, on a $n + 1 = m_1 + m_{k_1} + m_{k_2} + \ldots + m_{k_r}$.
Cette décomposition est parfaite car $1 \leq k_1 < k_2 < \ldots < k_r$.

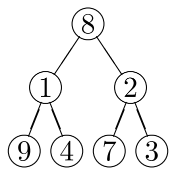
1.C.3
```
let rec decomp_parf n =
    match n with
        0 -> []
        |_ -> match decomp_parf (n-1) with
                    mk1::mk2::q when mk1 = mk2 -> (2*mk1 + 1)::q
                    |l -> 1::l;;
```

1.D. Création d'une liste de tas

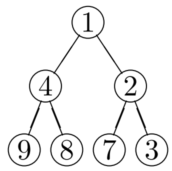
1.D.1 (a) Soit $k \in \mathbb{N}$ tel que la hauteur de h est la hauteur de a_k. On a $|a_k| = 2\,\text{haut}(a_k) - 1$ d'où $\text{haut}(a_k) = O(\log_2(|a_k|)$ puis $\text{haut}(h) = O(\log_2(|h|)$ (car $|a_k| \leq |h|$).
On n'a en revanche pas nécessairement $\text{long}(h) = O(\log_2(|h|)$. Par exemple, si h est formée de r tas de taille 1, on a $\text{long}(h) = |h|$.

(b) On a bien sûr encore $\text{haut}(h) = O(\log_2(|h|)$
Par ailleurs,$t_1 + t_2 + \ldots + t_r$ est une décomposition parfaite de $|h|$.
Soient $k_1, \ldots k_r$ tels que $\forall 1 \leq i \leq r, t_i = m_{k_i}$. On a $1 \leq k_1 \leq k_2 < \ldots < k_r$, d'où par récurrence immédiate, $\forall 1 \leq i \leq r, k_i \geq i - 1$ et donc $t_i \geq m_{i-1}$.
Ainsi, $|h| = \displaystyle\sum_{i=1}^{r} t_i \geq \sum_{i=1}^{r} m_{i-1} = 2^r - r - 1$, et $\text{long}(h) = r = O(\log_2(|h|))$.

1.D.2 (a) h_1 est de taille $11 = m_1 + m_2 + m_3$. h'_1 sera de taille $12 = m_1 + m_1 + m_2 + m_3$. On pose donc : $h'_1 = (a, 1) :: h_1$.
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 en prenant pour racine la valeur unique de a : 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) Similairement à l'obtention d'une décomposition parfaite :
- Si h est de longueur 0 ou 1, ou si $t_1 < t_2$, on pose $h' = (a, 1) :: h$.
- Sinon, on forme le quasi-tas (parfait d'après $t_1 = t_2$) dont la racine est l'élément de a, le fils gauche est a_1 et le fils droit est a_2, et on percole pour en faire un tas. On obtient un tas a' de taille $t' = 1 + t_1 + t_2$ et on pose $h' = [(a', t'), (a_3, t_3), \dots]$.

La liste renvoyée a les mêmes éléments que $(a, 1) :: h$ et est une liste de tas. D'après l'algorithme de décomposition parfaite, la suite des tailles vérifie bien QSC, donc la liste de tas vérifie bien TC.

Dans le meilleur cas (le premier), la complexité est en $O(1)$. Dans le pire cas (le second), l'appel à `percole` donne une complexité en $O(\log_2(|h|)$

(c)
```
let ajoute x h =
    match h with
        (a1,t1)::(a2,t2)::q when t1=t2 -> (percole (Noeud(x,a1,a2)),2*t1 +1)::q
        |_ -> (Noeud(x,Vide,Vide),1)::h;;
```

1.D.3 (a) Si la liste argument est déjà triée, à chaque fois que la fonction `ajoute` fait appel à `percole`, cet appel est fait sur un tas et a donc une complexité en $O(1)$. La complexité totale est donc en $O(n)$.

(b) Dans le cas général, il y a au plus n appels à la fonction `percole`, et chacun de ces appels se fait sur un quasi-tas de taille majorée par n. La complexité de chacun de ces appels est donc en $O(\log_2(n))$, d'où une complexité totale en $O(n \log_2(n))$.

1.E. Tri des racines

1.E.1
```
let echange_racines a1 a2 =
    match a1,a2 with
        Noeud(x1,g1,d1),Noeud(x2,g2,d2) -> Noeud(x2,g1,d1),Noeud(x1,g2,d2)
        |_ -> failwith "racine non existente";;
```

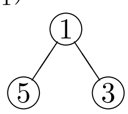
1.E.2 (a) On a $\min_{\mathcal{A}}((\texttt{percole a})) = \min_{\mathcal{A}}(a) \leq \min_{\mathcal{A}}(a_1) \leq \ldots \leq \min_{\mathcal{A}}(a_r)$ et `percole a` est bien un tas, donc `(percole a , t)::h` vérifie bien RO.

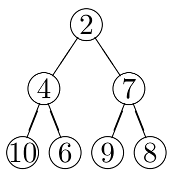
(b) b est formé du quasi-tas a dans lequel la racine est remplacée par $\min_{\mathcal{A}}(a_1)$, qui est inférieur à $\min_{\mathcal{A}}(a)$. La contrainte d'ordre est donc également respectée à la racine et b est un tas binaire parfait.

b_1 est formé du tas binaire parfait a_1 dans lequel la racine est modifiée, c'est donc un quasi-tas.

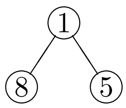
Enfin, la racine de b_1 (qui est la racine de a) est supérieure à $\min_{\mathcal{A}}(a)$, donc à $\min_{\mathcal{A}}(a_1)$, et les éléments non racines de b_1, qui sont les éléments non racines de a_1, sont supérieurs à $\min_{\mathcal{A}}(a_1)$, d'où $\min_{\mathcal{A}}(a_1) \leq \min_{\mathcal{A}}(b_1)$.

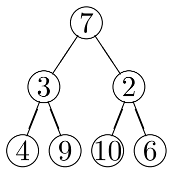
1.E.3 Pour respecter la contrainte de l'énoncé, on va se limiter aux opérations `percole` et `echange_racines`, en s'inspirant de la question précédente.
- On a $\min_{\mathcal{A}}(a_1) \leq \min_{\mathcal{A}}(a_1^1)$ donc il suffit de percoler a_1 et de l'ajouter en tête de h_1.





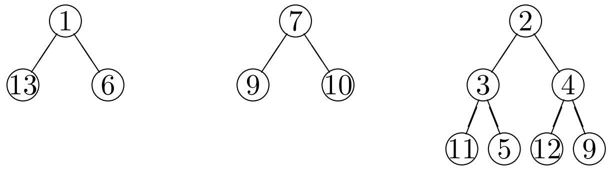
- Ici $\min_{\mathcal{A}}(a_2) > \min_{\mathcal{A}}(a_2^1)$. On échange les racines de a_2 et a_2^1 :



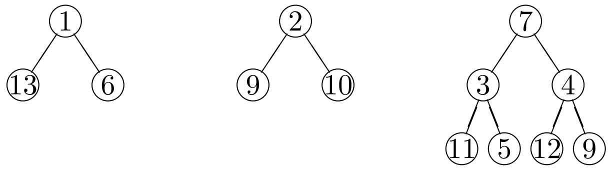


puis on percole le second arbre, qui est un quasi-tas :

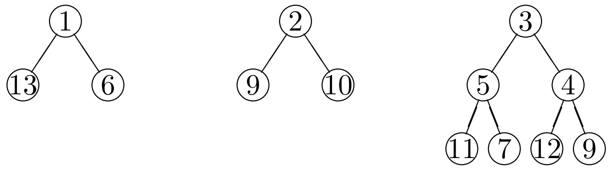
- On échange les racines des deux premiers arbres (car le minimum de a_3 est plus petit que celui de a_3^1)



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



On percole le troisième arbre, qui est un quasi-tas :



1.E.4 - Si h est vide, ou si $h = (a_1, t_1) :: k$ avec $\min_{\mathcal{A}}(a_1) \geq \min_{\mathcal{A}}(a)$, on percole a et on l'ajoute en tête de h.
- Sinon, $h = (a_1, t_1) :: q$, on échange alors les racines de a et a_1 et on procède récursivement pour ajouter le quasi-tas obtenu à partir de a_1 à q.

Si a est un tas non vide et $\min_{\mathcal{A}}(a) \leq \min_{\mathcal{H}}(h)$, on est dans le premier cas, la complexité est alors celle d'un appel de `percole` sur un tas, donc en $O(1)$.

Dans le cas général, on effectue au plus r échanges, chacun en $O(1)$, et une percolation, en $O(k)$, d'où une complexité totale en $O(k + r)$.

1.E.5
```
let rec insere_quasi a t h =
    match h with
        [] -> [(percole a ,t)]
        |(a1,t1)::q when min_quasi a <= min_tas a1 -> (percole a , t)::h
        |(a1,t1)::q -> let b,b1 = echange_racines a a1 in
                                   (b,t)::(insere_quasi b1 t1 q);;
```

1.E.6
```
let rec tri_racines h =
    match h with
        [] -> []
        |(a1,t1)::q ->  insere_quasi a1 t1 (tri_racines q);;
```

Puisque `insere_quasi` ne modifie pas la structure des arbres, la fonction `tri_racines` conserve bien la propriété TC.

1.E.7 Chaque appel à `insere_quasi` est en $O(k+r)$, la complexité de `tri_racines` est donc en $O(r(k+r))$. Or, si h vérifie la propriété TC, on a $k = O(\log_2(|h|))$ et $r = O(\log_2(|h|))$, d'où une complexité totale en $O((\log_2(|h|))^2)$.

1.F. Extraction des éléments d'une liste de tas

1.F.1 h' est une liste de tas vérifian RO, donc `insere_quasi` $a2$ $|a_2|$ h' également, donc
$h'' =$ `insere_quasi` ($a1$ $|a_1|$ (`insere_quasi` $a2$ $|a_2|$ h') également.

On a par ailleurs $|a_1| = |a_2| < t$ (car un tas est parfait). Comme h vérifie TC, on en déduit que les tailles des tas de h' sont supérieures à t et strictement croissantes, donc h'' vérifie TC.

1.F.2 Puisque $\text{haut}(a_1) = \text{haut}(a_2) < \text{haut}(h)$, les deux appels à `insere_quasi` gardent une complexité en $O(\log_2 |h|)$.

1.F.3
```
let rec extraire h =
    match h with
        [] -> []
        |(Noeud(x,Vide,Vide),t)::h' -> x::(extraire h')
        |(Noeud(x,a1,a2),t)::h' ->
            let h'' = insere_quasi a1 (t/2) (insere_quasi a2 (t/2) h') in
            x::(extraire h'')
        |_ -> failwith "bug dans extraire";;
```

1.F.4 La complexité vérifie la relation de récurrence $C(|h|) = C(|h| - 1) + O(\log_2 |h|)$, elle est donc en $O(|h| \log_2 |h|)$.

1.G. Synthèse

1.G.1
```
let tri_lisse l =
   extraire (tri_racines (constr_liste_tas l));;
```

1.G.2 L'appel `constr_liste_tas l` a un coût en $O(n \log_2(n))$, et produit une liste de tas de taille n. Le tri des racines a ensuite un coût en $O((\log_2(n))^2)$ et ne modifie pas la taille de la liste de tas. Enfin, l'extraction a un coût $O(n \log_2(n))$. La complexité totale est donc en $O(n \log_2(n))$.

1.G3 Lorsque la liste initiale est triée, l'appel `constr_liste_tas l` a un coût en $O(n)$. De plus, on a par récurrence immédiate que tous les éléments d'un tas sont inférieurs aux éléments des tas suivants. Les appels à `insere_quasi` sont donc en $O(1)$, l'appel à `tri_racines` est donc en $O(1)$, et l'appel à `extraire` est en $O(n)$. On en déduit une complexité totale en $O(n)$.

2 Implantation dans un tableau

2.A. Lors de l'appel à `tri_lisse`, on construit une liste de tas de taille n, nécessitant donc une quantité de mémoire proportionnelle à n. La complexité spatiale est donc en $\Omega(n)$.

2.B.
```
let fg t =
    {donnees = t.donnees ; pos = t.pos+1 ; taille = t.taille/2};;

let fg t =
    let k = t.taille/2 in
    {donnees = t.donnees ; pos = t.pos+1+k ; taille = k};;
```

2.C.
```
let min_tas_array t =
    t.donnees.(t.pos);;

let min_quasi_array t =
    if t.taille=1 then t.donnees.(t.pos)
    else min t.donnees.(t.pos)
            (min t.donnees.(t.pos+1) t.donnees.(t.pos+1+(t.taille/2)));;
```

2.D.
```
let rec percole_array t =
    if t.taille > 1 then
        let m = min_quasi_array t in
        let r = t.donnees.(t.pos) in
        match m with
            x when x = min_tas_array (fg t) ->
                                                    t.donnees.(t.pos) <- m;
```

```
                                                        t.donnees.(t.pos + 1) <- r;
                                                        percole_array (fg t);
                |x when x = min_tas_array (fd t) ->
                                                        t.donnees.(t.pos) <- m;
                                                        t.donnees.((fd t).pos) <- r;
                                                        percole_array (fd t);
                |_ -> ();;
```

2.E.
```
let ajoute_array d p h =
    match h with
        a1::a2::q when a1.taille=a2.taille -> let a = {donnees = d;
                                                       pos = p ;
                                                       taille = a1.taille * 2 + 1}
                                               in
                                               percole_array a;
                                               a::q
        |_ -> let a = {donnees = d; pos = p ; taille = 1} in a::h;;
```

2.F.
```
let echange_racines_array a1 a2 =
    let d = a1.donnees in
    let x = d.(a1.pos) in
    d.(a1.pos) <- d.(a2.pos);
    d.(a2.pos) <- x;;
```

2.G.
```
let rec insere_quasi_array a h =
    match h with
        [] -> percole_array a; [a]
        |a1::q when min_quasi_array a <= min_tas_array a1 -> percole_array a;a::h
        |a1::q -> echange_racines_array a a1;
                    a::(insere_quasi_array a1 q);;
```

2.H.
```
let rec tri_racines_array h =
    match h with
        [] -> []
        |a1::q ->  insere_quasi_array a1 (tri_racines_array q);;
```

2.I.
```
let rec extraire_array h =
    match h with
        [] -> ()
        |a::h' when a.taille = 1 -> extraire_array h'
        |a::h' ->
            let h'' = insere_quasi_array (fg a) (insere_quasi_array (fd a) h') in
            extraire_array h'';;
```

2.J.
```
let tri_lisse_array l =
  extraire_array (tri_racines_array (constr_liste_tas_array l));;
```

2.K. L'analyse de complexité est similaire au cas de `tri_lisse` et donne également une complexité en $O(n \log_2 n)$.

2.L. De même, la complexité dans le cas d'un tableau déjà trié est en $O(n)$.

2.M. Puisque le champs `donnees` de chaque tas est partagé avec le tableau argument, le coût en mémoire d'un tas est en $O(1)$. Le coût en mémoire de la liste de tas construite h est donc en $O(\text{long}(h)) = O(\log_2 n)$ (car h vérifie TC). La mémoire consommée par ailleurs par des variables locales est en $O(1)$, d'où une complexité spatiale totale en $O(\log_2(n))$.