Centrale 2016 - Un algorithme de tri
Un 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 -> if x<=a then x::u
            else a::(insere x q) ;;
```

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

1.A.3 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 $|u|$ (notation pour le nombre des éléments de 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 $m_0 = 0$ (l'arbre vide est de taille nulle) et

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

On en déduit, par exemple par récurrence, que

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

1.B.2 Le minimum est à la racine

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

1.B.3 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 = function
   |Noeud(x,Vide,Vide) ->  x
   |Noeud(x,a1,a2) -> min x (min (min_tas a1) (min_tas a2));;
```

1.B.4 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.

1.C. Décomposition parfaite d'un entier

1.C.1 On a

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

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

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

1.C.2 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.

1.C.3 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 transforme l =
   match l with
  |[x] -> [1;x]
  |x::y::q -> if x=y then (2*x+1)::q
              else 1::l ;;

let rec decomp_parf n =
    if n=0 then []
    else if n=1 then [1]
    else transforme (decomp_parf (n-1));;
```

1.D. Création d'une liste de tas

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

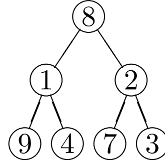
$$|h| \geq 1 + \sum_{k=1}^{r-1} m_k \geq 1 + \sum_{k=1}^{r-1} 2^k = 2^{r-1}$$

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

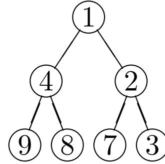
1.D.2 (a) h_1 est de taille $11 = m_1 + m_2 + m_3$. h_1' sera de taille $12 = m_1 + m_1 + m_2 + m_3$ et donc composée de 4 tas de tailles $1, 1, 3, 7$. Il suffit ici de poser

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

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) 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) 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 ;;
```

1.D.3 (a) Soit C_n le nombre d'opération dans le cas le pire dans l'appel `constr_liste_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) Dans l'appel `constr_liste_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 $O(n\log_2(n))$.

1.E. Tri des racines

1.E.1

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

1.E.2 (a) La racine x du tas `percole a t` vaut $\min_{\mathcal{A}}(a)$. 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 $x = \min_{\mathcal{A}}(a) \leq \min_{\mathcal{A}}(a_1) \leq \dots \min_{\mathcal{A}}(a_r)$.

(b) 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.

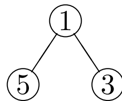
De même, `b1` est comme `a1` sauf la racine. La contrainte d'ordre des étiquettes est donc préservée pour les étiquettes sauf peut-être la racine et `b1` est un quasi-tas.

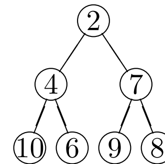
Enfin, on a placé en racine de `b` l'élément $\min_{\mathcal{A}}(a_1)$ qui plus petit que toutes les étiquettes de `a` et donc que toutes celles des fils de la racine de `b`. On trouve donc le minimum de `b` à la racine et il vaut $\min_{\mathcal{A}}(\texttt{b}) = \min_{\mathcal{A}}(a_1)$. Dans `b1`,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

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

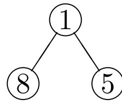
1.E.3 On remarque qu'une opération de percolation sur un quasi-tas est permise puisqu'elle n'induit que des échanges d'étiquettes dans le quasi-tas.

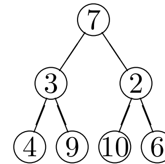
- Le minimum de a_1 étant plus petit que la racine de a_1^1 (qui est son minimum), une percolation dans a_1 suffit avec la question précédente.



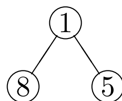


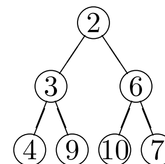
- Cette fois le minimum de a_2 est plus grand. On commence par échanger les racines



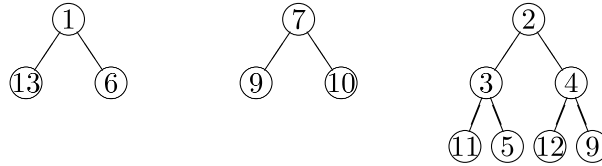


puis on effectue une percolation dans le second arbre

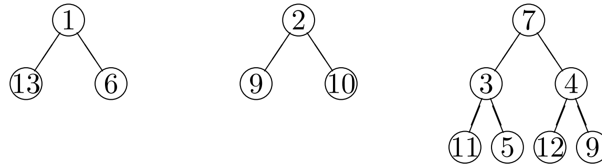




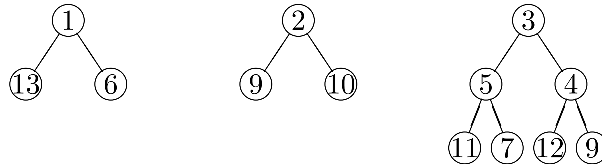
- On échange les racines des deux premiers arbres (car le minimum de a_3 est plus petit que celui de a_3^1)



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



1.E.4 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.

1.E.5 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;;
```

1.E.6 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);;
```

1.E.7 Quand h est une liste de tas de taille n qui vérifie TC, on a $\text{haut}(h) = O(\log_2(n))$ et $\text{long}(h) = O(\log_2(n))$ (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 $\text{long}(h)$

appels à `insere_quasi` qui sont tous de complexité $O(\log_2(n))$. On a finalement une complexité $O((\log_2(n))^2)$.

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

1.F.1 $(a_2, |a_2|) :: h'$ 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.

1.F.2 Comme en I.E.7, les deux appels à `insere_quasi` auront un coût $O(\log_2 |h|)$.

1.F.3 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)));;
```

1.F.4 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.

1.G. Synthèse

1.G.1 On combine toutes nos fonctions.

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

1.G.2 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.

1.G3 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.

2 Implantation dans un tableau

Dans la représentationsous forme de tableau, on va utiliser que le caractère mutable des tableaux pour les faire évoluer dynamiquement. Afin de comprendre le mécanisme de partage du tableau de données, il faut comprendre qu'en Caml la commande `let tab1=tab2;;` ne définit pas un nouveau tableau mais définit un pseudonyme. La modification de `tab2` entraîne celle de `tab1` (et réciproquement).

2.A. Dans tri_lisse, en plus de la liste donnée en argument, on construit une liste de tas ayant n éléments et nécessitant une place mémoire au moins de taille n. La complexité spatiale est donc $\Omega(n)$.

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

let fd t =
   {donnees = t.donnees ;
    pos     = t.pos+1+(t.taille/2) ;
    taille  = (t.taille/2)};;
```

2.C.
```
let min_tas_vect a = a.donnees.(a.pos);;

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

2.D. Dans la fonction ci-dessous, on utilise trois variables de stockage pour la lisibilité de la fonctions (pour stocker les racines du tas et des fils droit et gauche). On pourrait éviter un tel stockage. Seule une variable tampon est utile pour l'échange de case (et encore, il y a des stratégies d'échange n'utilisant pas de variable tampon). La fonction est essentiellement la même que percole et on fait un appel récursif à droite ou gauche selon les valeurs des racines.

```
let rec percole_vect t =
   if t.taille<=2 then ()
   else begin
        let x=t.donnees.(t.pos) in
        let xg=t.donnees.(t.pos+1) in
        let xd=t.donnees.(t.pos+1+(t.taille/2)) in
        if x<=(min xg xd) then ()
        else if xg<xd then begin
               t.donnees.(t.pos) <- xg ;
               t.donnees.(t.pos+1)<- x ;
               percole_vect (fg t)
               end
        else begin
               t.donnees.(t.pos) <- xd ;
               t.donnees.(t.pos/1+(t.taille/2)) <- x ;
               percole_vect (fd t)
             end
        end;;
```

2.E. La fonction est là encore très semblable. Il faut faire attention au fait que percole_vect ne renvoie pas un tas binaire mais modifie le tableau des données.

```
let ajoute_vect d p h =
    match h with
     [] -> [{donnees=d;taille=1;pos=p}]
   |[tas] -> {donnees=d;taille=1;pos=p}::h
   |tas1::tas2::q ->
         if tas1.taille<tas2.taille then {donnees=d;taille=1;pos=p}::h
         else begin
              percole_vect {donnees=d;taille=1+tas1.taille+tas2.taille;pos=p};
```

```
                {donnees=d;taille=1+tas1.taille+tas2.taille;pos=p}::h
                end;;
```

2.F.
```
let echange_racines_vect tas1 tas2=
    let x=tas1.donnees.(tas1.pos) in
    tas1.donnees.(tas1.pos) <- tas2.donnees.(tas2.pos);
    tas2.donnees.(tas2.pos) <- x;;
```

2.G.
```
let rec insere_quasi_vect a h=
    match h with
   |[] -> percole_vect a;[a]
   |a1::k -> if (min_quasi_vect a)<=(min_tas_vect a1) then
                 begin
                 percole_vect a;
                 a::h
                 end
             else begin
                 echange_racines_vect a a1 ;
                 a::(insere_quasi_vect a1 k)
                 end;;
```

2.H.
```
let rec tri_racines_vect h =
    match h with
   |[] -> []
   |a1::k -> insere_quasi_vect a1 (tri_racines_vect k);;
```

2.I. La fonction ne renvoie rien puisqu'elle se contente de modifier le tableau des données.

```
let rec extraire_vect h =
    match h with
   |[] -> ()
   |a::k -> if a.taille=1 then extraire k
              else extraire_vect (insere_quasi_vect (fg a)
                                             (insere_quasi_vect (fd a) k));;
```

2.J. La fonction ne renvoie rien puisqu'elle se contente de modifier le tableau des données.

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

2.K. La stratégie utilisée ne change pas et la complexité temporelle dans le cas le pire est $O(n \log_2 n)$.

2.L. La stratégie utilisée ne change pas et la complexité temporelle dans le cas le pire est $O(n)$.

2.M. Les seules variables supplémentaires introduites sont celles pour les échanges dans les tableaux, mais elles ne servent que localement et pourraient être omises, et les tas binaires des listes. Or, comme on garantit la propriété TC, ces listes sont de taille $O(\log_2 n)$. Comme on partage les données, un tas est stocké en espace constant. La complexité spatiale est donc $O(\log_2(n))$.