Un algorithme de tri (Centrale 2016)

Rédigé par Jean-Pierre Becirspahic (jp.becir@info-llg.fr).

II Algorithme sur des arbres

II.A – Tri par insertion

II.A.1) On définit la fonction :

```
let rec insere x = function
  | t::q when x > t -> t::(insere x q)
  | lst             -> x::lst ;;
```

II.A.2) On en déduit le tri par insertion :

```
let rec tri_insertion = function
  | []   -> []
  | t::q -> insere t (tri_insertion q) ;;
```

II.A.3) Appliquée à une liste de longueur $n \geqslant 1$ la fonction **insere** réalise entre 1 et n comparaisons donc les suites $(\mathrm{P_I}(n))_{n\in\mathbb{N}}$ et $(\mathrm{M_I}(n))_{n\in\mathbb{N}}$ vérifient les relations :

$$\mathrm{P_I}(0) = \mathrm{M_I}(0) = 0, \qquad \mathrm{P_I}(1) = \mathrm{M_I}(1) = 0, \qquad \text{et} \qquad \forall n \geqslant 1,\ \mathrm{P_I}(n+1) = \mathrm{P_I}(n) + n, \quad \mathrm{M_I}(n+1) = \mathrm{M_I}(n) + 1.$$

On en déduit que pour tout $n \in \mathbb{N}$, $\mathrm{P_I}(n) = \dfrac{n(n-1)}{2}$ et $\mathrm{M_I}(n) = \begin{cases} n-1 & \text{si } n \geqslant 1 \\ 0 & \text{si } n = 0 \end{cases}$.

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.

II.B – Tas binaires

II.B.1) On dispose des relations $m_0 = 0$ et $m_{k+1} = 2m_k + 1$ qui permettent de prouver par récurrence que $m_k = 2^k - 1$.

II.B.2) L'élément minimal d'un tas se trouve à la racine, d'où la fonction :

```
let min_tas = function
  | Vide            -> failwith "min_tas"
  | Noeud (x, _, _) -> x ;;
```

II.B.3) L'élément minimal d'un quasi-tas **Noeud (x, a1, a2)** est égal à x si a_1 et a_2 sont vides, et au minimum de x, $\min_{\mathcal{A}}(a_1)$ et $\min_{\mathcal{A}}(a_2)$ sinon. D'où la fonction :

```
let min_quasi = function
  | Vide                  -> failwith "min_quasi"
  | Noeud (x, Vide, Vide) -> x
  | Noeud (x, a1, a2)     -> min x (min (min_tas a1) (min_tas a2)) ;;
```

II.B.4) On définit :

```
let rec percole a = match a with
  | Noeud (x, _, _) when x = min_quasi a -> a
  | Noeud (x, Noeud (x1, b1, b2), a2) when x1 = min_quasi a ->
                                          Noeud (x1, percole (Noeud (x, b1, b2)), a2)
  | Noeud (x, a1, Noeud (x2, b1, b2)) when x2 = min_quasi a ->
                                          Noeud (x2, a1, percole (Noeud (x, b1, b2)))
  | _ -> Vide ;;
```

Dans le pire des cas, par exemple lorsque l'étiquette de la racine est l'élément maximal du quasi-tas, il faut procéder à k appels à la fonction **percole** pour descendre cet élément au niveau des feuilles, où k est la hauteur du quasi-tas. Le coût de cette fonction est donc un $\mathrm{O}(k)$.

II.C – *Décomposition parfaite d'un entier*

II.C.1) Les termes de la suite $(m_k)_{k\geqslant 1}$ inférieurs ou égaux à 101 sont : 1, 3, 7, 15, 31, 63 et permettent de décomposer les entiers :

$$
\begin{aligned}
6 &= 3+3 = m_2+m_2 \\
7 &= m_3 \\
8 &= 1+7 = m_1+m_3 \\
9 &= 1+1+7 = m_1+m_1+m_3 \\
10 &= 3+7 = m_2+m_3 \\
27 &= 1+1+3+7+15 = m_1+m_1+m_2+m_3+m_4
\end{aligned}
\qquad
\begin{aligned}
28 &= 3+3+7+15 = m_2+m_2+m_3+m_4 \\
29 &= 7+7+15 = m_3+m_3+m_4 \\
30 &= 15+15 = m_4+m_4 \\
31 &= m_5 \\
100 &= 3+3+31+63 = m_2+m_2+m_5+m_6 \\
101 &= 7+31+63 = m_3+m_5+m_6
\end{aligned}
$$

II.C.2) Si $r \geqslant 2$ et $k_1 = k_2$ on a $m_{k_1}+m_{k_2} = 2\times(2^{k_2}-1) = 2^{k_2+1}-2 = m_{k_2+1}-1$ donc $n+1 = m_{k_2+1}+(m_{k_3}+m_{k_4}+\cdots,m_{k_r})$ et cette décomposition est parfaite car $k_2+1 \leqslant k_3 < k_4 < \cdots k_r$.

Dans le cas ou $r \leqslant 1$ ou $k_1 < k_2$ la décomposition $n+1 = m_1+(m_{k_1}+\cdots+m_{k_r})$ est aussi parfaite puisque $1 \leqslant k_1 < k_2 \cdots < k_r$.

II.C.3) On en déduit la fonction :

```
let rec decomp_parf = function
  | 0 -> []
  | n -> match decomp_parf (n-1) with
           | m1::m2::q when m1 = m2 -> (2 * m2 + 1)::q
           | l                      -> 1::l ;;
```

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

II.D – *Création d'une liste de tas*

Il y a sans doute une erreur dans l'énoncé : une liste de tas est une liste de couples de la forme (a,t) où a désigne un tas parfait (et non pas un arbre) et $t = |a|$.

II.D.1)

a) Notons que si h est une liste non vide de tas *vides* on a $|h| = 0$ et $\log_2|h|$ n'est pas défini. Je suppose dans cette question qu'au moins un des tas de h est non vide.

Posons $h = ((a_1,t_1),\ldots,(a_r,t_r))$ et notons i_0 un entier vérifiant $\mathrm{haut}(h) = \mathrm{haut}(a_{i_0})$.

L'arbre a_{i_0} est parfait donc $\mathrm{haut}(a_{i_0}) = \mathrm{O}(\log_2|a_i|)$ et puisque $|a_i| \leqslant |h|$ on a aussi $\mathrm{haut}(h) = \mathrm{O}(\log_2|h|)$.

En revanche on a pas nécessairement $\mathrm{long}(h) = \mathrm{O}(\log_2|h|)$: il suffit de considérer une liste de r tas de taille 1 ; on a alors $\mathrm{long}(h) = r = |h|$.

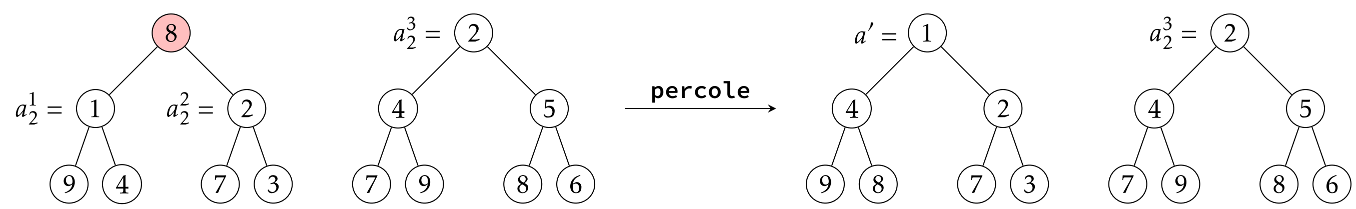
b) Supposons maintenant que h vérifie en plus la condition TC. Dans ce cas, $|h| = t_1+t_2+\cdots+t_r$ est une décomposition parfaite de $|h|$ donc on a $t_1 \geqslant 1$ et $t_i \geqslant m_{i-1}$ pour $i \geqslant 2$. Ainsi, $|h| \geqslant 1+\sum_{i=2}^{r}(2^{i-1}-1) = 2^r - r \geqslant 2^{r-1}$ et $r \leqslant 1+\log_2|h|$.

Dans le cas d'une liste non vide de tas vérifiant la condition TC on a donc aussi $\mathrm{long}(h) = \mathrm{O}(\log_2|h|)$.

II.D.2)

a) $12 = 1+1+3+7$ est une décomposition parfaite donc l'ajout de l'arbre a dans la liste de tas h_1 se fait simplement en insérant $(a,1)$ en tête de la liste h_1 : $h_1' = ((a,1),(a_1^1,1),(a_1^2,3),(a_1^3,7))$.

En revanche, $14 = 1+3+3+7$ n'est pas une décomposition parfaite ; la décomposition parfaite est $14 = 7+7$. On obtient h_2' en créant le quasi-tas **`Noeud(x, a1, a2)`** avec ici $x = 8$, $a_1 = a_2^1$ et $a_2 = a_2^2$ puis en reformant un tas en suivant l'algorithme implémenté par la fonction **`percole`**. Graphiquement cela donne :



et $h_2' = ((a',7),(a_2^3,7))$.

b) Considérons de nouveau la formule établie à la question II.C.2. Si h est de longueur inférieure ou égale à 1 ou si $t_1 < t_2$ il suffit d'insérer $(a, 1)$ en tête de h pour obtenir h'.

Reste le cas où h est de longueur supérieure ou égale à 2 avec $t_1 = t_2$. Dans ce cas, on crée le quasi-tas **`Noeud(x, a1, a2)`** où x est l'étiquette de a, et on le transforme en tas parfait a' à l'aide de la fonction **`percole`**. Alors $h' = ((a', 2t_2 + 1), (a_3, t_3), \cdots, (a_r, t_r))$ est une liste de tas vérifiant la condition TC d'après la formule de la question II.C.2.

Dans le premier cas la complexité de cette fonction est en $O(1)$, dans le second en $O(\log_2 |a_1|)$ d'après la question II.B.4.

c) On en déduit la fonction :

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

II.D.3)

a) Lorsque la liste initiale est déjà triée, chaque quasi-tas qui est construit dans le processus expliqué aux questions précédentes se trouve être en réalité un tas parfait, puisque l'étiquette de la racine est inférieure aux étiquettes de ses fils. Chaque utilisation de la fonction **`percole`** se réalise donc en temps constant, et la complexité totale est en $O(n)$.

b) Dans le cas général, le nombre d'appels à la fonction **`percole`** est majoré par n, et chaque quasi-tas passé en argument a une taille majorée par n, ce qui permet de majorer la complexité de chaque appel à la fonction **`percole`** par un $O(\log_2 n)$, et par la suite de majorer la complexité de la fonction **`constr_liste_tas`** par un $O(n \log_2 n)$.

II.E – Tri des racines

II.E.1) On définit la fonction :

```
let echange_racines = fun
  | (Noeud (x, a1, a2)) (Noeud (y, b1, b2)) -> Noeud (y, a1, a2), Noeud (x, b1, b2)
  | _ _                                     -> failwith "echange_racine" ;;
```

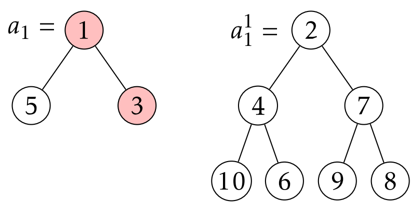
II.E.2)

a) Après percolation, l'étiquette de la racine de a est $\min_{\mathcal{A}}(a)$ donc **`(percole a, t)::h`** vérifie la condition RO.

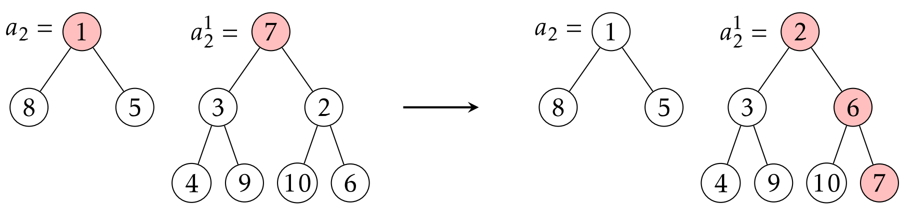
b) Lorsqu'on échange l'étiquette de la racine d'un tas par une étiquette plus petite, on garde un tas donc b est un toujours un tas parfait. En revanche, b_1 n'est plus qu'un quasi-tas.

$\min_{\mathcal{A}}(b)$ est l'étiquette de sa nouvelle racine, à savoir $\min_{\mathcal{A}}(a_1)$; Quant à b_1, l'étiquette de sa racine ayant augmenté, il en est de même de $\min_{\mathcal{A}}(b_1)$ et $\min_{\mathcal{A}}(b_1) \geqslant \min_{\mathcal{A}}(a_1)$.

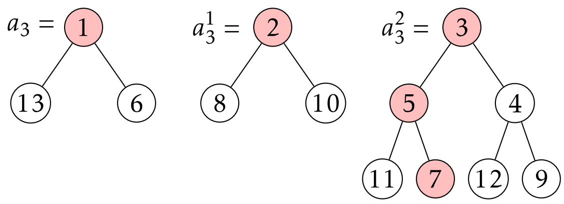
II.E.3) Pour le couple (a_1, h_1), il suffit de percoler le quasi-tas a_1 puisque $\min_{\mathcal{A}}(a_1) \leqslant \min_{\mathcal{A}}(a_1^1)$:



Pour le couple (a_2, h_2), on a $\min_{\mathcal{A}}(a_2) > \min_{\mathcal{A}}(a_1^1)$ donc on échange les étiquettes des racines de a_2 et de a_2^1 (ce qui fait de a_2 un tas d'après la question précédente), puis on percole le quasi-tas a_2^1 pour récupérer un tas parfait :



Enfin, pour le couple (a_3, h_3), on réalise successivement : un échange des étiquettes des racines de de a_3 et de a_3^1 (ce qui fait de a_3 un tas), un échange des racines de a_3^1 et de a_3^2 (ce qui fait de a_3^1 un tas), puis une percolation de a_3^2 pour obtenir la liste de tas vérifiant la condition RO :



II.E.4) Si h est non vide on notera $h = (a_1, t_1) :: q$.
Si h est vide ou si $\min_{\mathcal{A}}(a) \leqslant \min_{\mathcal{A}}(a_1)$ il suffit de percoler a_1 puis de poser $h' = (a_1, t_1) :: h$ pour obtenir une liste de tas vérifiant la condition RO.
Dans le cas contraire, on permute les étiquettes des racines de a et de a_1, ce qui fait de a un tas et de a_1 un quasi-tas puis on procède récursivement pour ajouter a_1 à la liste q.

Dans le premier cas, la complexité se résume au coût de la percolation de a, donc c'est un O(1) si a est déjà un tas.
Dans le cas général on réalise au plus r permutations des étiquettes des racines, ce qui se réalise en O(r), suivi d'une percolation qui se réalise en O(k) permutations d'étiquettes au sein du quasi-tas à percoler. La complexité totale de cette fonction est donc en O($k + r$).

II.E.5) Traduisons maintenant cet algorithme en Caml :

```
let rec insere_quasi a t h = match h with
  | []                                   -> [percole a, t]
  | (a1, _)::_ 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) ;;
```

II.E.6) On procède maintenant a l'instar du tri par insertion :

```
let rec tri_racines = function
  | []        -> []
  | (a, t)::h -> insere_quasi a t (tri_racines h) ;;
```

II.E.7) Posons $h = (a, t) :: h'$. La complexité temporelle de la fonction précédente vérifie une relation de la forme $\mathrm{C}(h) = \mathrm{C}(h') + \mathrm{O}(k + r)$, où $k = \max(\mathrm{haut}(a), \mathrm{haut}(h')) = \mathrm{haut}(h)$ et $r = \mathrm{long}(h') = \mathrm{long}(h) - 1$.
On en déduit que $\mathrm{C}(h) = \mathrm{O}(r(k + r))$. Or d'après la question II.D.1, puisque h vérifie la condition TC on a $k = \mathrm{O}(\log_2 |h|)$ et $r = \mathrm{O}(\log_2 |h|)$, donc la complexité de la fonction **`tri_racines`** est en $\mathrm{O}((\log_2 |h|)^2)$.

II.F – Extraction des éléments d'une liste de tas

II.F.1) h' est une liste de tas vérifiant RO et TC et a_1 et a_2 sont des quasi-tas puisque ce sont des tas, donc h'' est une liste de tas vérifiant RO (d'après la question II.E.5), et vérifiant toujours TC puisque $|a_1| = |a_2| < t$ et que le premier tas de h' (si h' n'est pas vide) a une taille supérieure ou égale à t.

II.F.2) Le coût du calcul de h'' est en $\mathrm{O}(k_1 + r_1) + \mathrm{O}(k_2 + r_2)$ avec $k_2 = \max(\mathrm{haut}(a_2), \mathrm{haut}(h')) \leqslant \mathrm{haut}(h)$, $r_2 = \mathrm{long}(h) - 1$, $k_1 = \max(\mathrm{haut}(a_1), \mathrm{haut}(a_2), \mathrm{haut}(h')) \leqslant \mathrm{haut}(h)$, $r_1 = \mathrm{long}(h)$ soit une complexité en $\mathrm{O}(\mathrm{haut}(h) + \mathrm{long}(h))$.
Puisque h vérifie la condition TC, cette complexité est en $\mathrm{O}(\log_2 |h|)$ (question II.D.1).

II.F.3) Dans une liste de tas vérifiant les conditions RO et TC l'élément minimal se trouve à la racine du premier de ces tas. D'où la fonction :

```
let rec extraire = function
  | []                        -> []
  | (Noeud (x, Vide, Vide), 1)::h -> x::(extraire h)
  | (Noeud (x, a1, a2), t)::h     -> let u = (t / 2) in
                                     let hh = (insere_quasi a1 u (insere_quasi a2 u h)) in
                                     x::(extraire hh)
  | _                         -> failwith "extraire" ;;
```

II.F.4) La complexité $\mathrm{C}(h)$ de cette fonction vérifie d'après la question précédente une relation de la forme $\mathrm{C}(h) = \mathrm{C}(h'') + \mathrm{O}(\log_2 |h|)$ avec $|h''| = |h| - 1$ donc $\mathrm{C}(h) = \mathrm{O}(|h| \log_2 |h|)$.

II.G – Synthèse

II.G.1) À partir d'une liste l on construit une liste de tas vérifiant la condition TC à l'aide de la fonction **`constr_liste_tas`**. On transforme cette liste en une liste de tas vérifiant les conditions RO et TC à l'aide de la fonction **`tri_racines`**. Enfin, on en extrait les éléments triés à l'aide de la fonction **`extraire`**.

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

II.G.2) D'après la question II.D.3 la complexité de la fonction **`constr_liste_tas`** est en $O(n \log_2 n)$ (voire en $\Theta(n)$ d'après la note de bas de page), et la liste de tas obtenue h vérifie $|h| = n$. D'après la question II.E.7, la complexité de la fonction **`tri_racines`** est en $O((\log n)^2)$. Enfin, d'après la question II.F.4 la complexité de la fonction **`extraire`** est en $O(n \log_2 n)$, donc la fonction **`tri_lisse`** a une complexité temporelle en $O(n \log_2 n)$.

II.G.3) Dans le cas où la liste initiale est déjà triée, La question II.D.3 a montré que la construction de la liste de tas h est en $O(n)$. De plus, la liste obtenue $h = ((a_1, t_1), \dots, (a_r, t_r))$, outre les conditions RO et TC, vérifie la propriété suivante :

Si $1 \leqslant i < j \leqslant r$, toute étiquette de a_i est inférieure ou égale à toute étiquette de a_j.

Ainsi, les différentes applications de la fonction **`insere_quasi`** dans la fonction **`extraire`** se réalisent toutes en temps constant et la relation établie à la question II.F.4 s'écrit dans ce cas particulier : $C(h) = C(h'') + O(1)$, ce qui conduit à $C(h) = O(|h|) = O(n)$.

La complexité totale de **`tri_lisse`** dans ce cas particulier est donc en $O(n) + O((\log_2 n)^2) + O(n) = O(n)$.

III Implantation dans un tableau

III.A –

La place occupée en mémoire par la liste de tas h créée lors de l'exécution de **`tri_lisse`** est au minimum proportionnelle à la taille $|h|$ de cette liste, donc a un coût spatial au moins en $\Omega(n)$.

III.B –

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

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

III.C –

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

let min_quasi_vect a = let x = a.donnees.(a.pos) in
                       if a.taille = 1 then x
                       else let y = min_tas_vect (fg a) and z = min_tas_vect (fd a)
                            in min x (min y z) ;;
```

III.D –

Il y a une erreur d'énoncé, le champ **`donnees`** est par erreur appelé **`data`**.

```
let rec percole_vect a =
  if a.taille > 0 then
    let m = min_quasi_vect a and x = a.donnees.(a.pos) in
    if m = x then ()
    else if m = min_tas_vect (fg a) then
         ( a.donnees.(a.pos) <- a.donnees.(a.pos + 1) ;
           a.donnees.(a.pos + 1) <- x ;
           percole_vect (fg a)
         )
    else ( a.donnees.(a.pos) <- a.donnees.(a.pos + 1 + a.taille / 2) ;
           a.donnees.(a.pos + 1 + a.taille / 2) <- x ;
           percole_vect (fd a)
         )
```

III.E –

Selon moi, l'énoncé est faux ; l'élément appelé **h** dans l'énoncé doit être de type *tasbin list* et la fonction **ajoute_vect** de type *int vect –> int –> tasbin list –> tasbin list* et se définir ainsi :

```
let ajoute_vect d p = function
  | a1::a2::q when a1.taille = a2.taille ->
        let a = {donnees = d; pos = p; taille = 2 * a2.taille + 1}
        in percole_vect a ; a::q
  | h -> let a = {donnees = d; pos = p; taille = 1} in a::h ;;
```

La fonction **constr_liste_tas_aux** doit alors être de type *int vect –> int –> tasbin list –> tasbin list* et se définir ainsi :

```
let rec constr_liste_tas_aux d p h =
  if p = 0 then h
  else let h' = ajoute_vect d (p-1) h in constr_liste_tas_aux d (p-1) h' ;;
```

(l'énoncé écrit h au lieu de h' sur la dernière ligne).
Enfin, il y a aussi une erreur dans la définition de **constr_liste_tas_vect** (il manque un **_aux**) qui doit s'écrire :

```
let constr_liste_tas_vect d = constr_liste_tas_aux d (vect_length d) [] ;;
```

III.F –

Il serait plus cohérent d'appeler la fonction demandée **echange_racines_vect** :

```
let echange_racines_vect a b =
  let d = a.donnees in
  let x = d.(a.pos) in
  d.(a.pos) <- d.(b.pos) ; d.(b.pos) <- x ;;
```

III.G –

Selon moi, cette fonction devrait être de type *tasbin –> tasbin list –> tasbin list*.

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

III.H –

Selon moi, cette fonction devrait être de type *tasbin list –> tasbin list*.

```
let rec tri_racines_vect = function
  | []   -> []
  | a::h -> insere_quasi_vect a (tri_racines_vect h) ;;
```

III.I –

```
let rec extraire_vect = function
  | [] -> ()
  | a::h when a.taille = 1 -> extraire_vect h
  | a::h                   -> let a1 = fg a and a2 = fd a in
                              let hh = (insere_quasi_vect a1 (insere_quasi_vect a2 h)) in
                              extraire_vect hh ;;
```

III.J –

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

III.K –

Les opérations élémentaires qui étaient de coût constant dans la partie II restent de coût constant dans la partie III même si l'implantation a changé, donc la complexité temporelle de la fonction **`tri_lisse_vect`** est identique à celle de **`tri_lisse`**, à savoir un $O(n \log_2 n)$ dans le pire des cas...

III.L –

... et un $O(n)$ dans le cas d'un vecteur préalablement trié.

III.M –

En revanche, puisque tous les tas partagent le même espace mémoire pour stocker les données, la complexité spatiale de cette fonction est proportionnelle à la longueur r de la liste de tas h créée. Or la question II.D.1 a montré que lorsque la liste de tas vérifie la condition TC sa longueur vérifie $r = O(\log_2 |h|) = O(\log_2 n)$. La complexité spatiale de la fonction **`tri_liste_vect`** est donc en $O(\log_2 n)$.