Question II.A.1) –

```
let rec insere x u = match u with
  | [] -> [x]
  | e::q when e >= x -> x :: u
  | e::q -> e :: (insere x q);;
```

Question II.A.2) –

```
let rec tri_insertion li = match li with
  | [] -> []
  | e::q -> insere e (tri_insertion q);;
```

Question II.A.3) – Soit $P'_I(n)$ le nombre de comparaisons effectuées par l'appel à « `insere x u` » dans le pire cas pour une liste de longueur n. Alors :

$$\begin{cases} P'_I(0) = 0 \\ \forall n \geqslant 1 : P'_I(n) = 1 + P'_I(n-1) \end{cases} \qquad \begin{cases} P_I(0) = 0 \\ \forall n \geqslant 1 : P_I(n) = P_I(n-1) + P'_I(n-1) \end{cases}$$

Ainsi, pour tout $n \in \mathbb{N}$, $P'_I(n) = n$ et $P_I(n) = \sum_{k=0}^{n-1} P'_I(k)$. En conclusion :

$$\boxed{\forall n \in \mathbb{N} : P_I(n) = \frac{n(n-1)}{2}}$$

Soit $M'_I(n)$ le nombre de comparaisons effectuées par l'appel à « `insere x u` » dans le meilleur cas pour une liste de longueur n. Alors :

$$\begin{cases} M'_I(0) = 0 \\ \forall n \geqslant 1 : M'_I(n) = 1 \end{cases} \qquad \begin{cases} M_I(0) = 0 \\ \forall n \geqslant 1 : M_I(n) = M_I(n-1) + M'_I(n-1) \end{cases}$$

Ainsi, pour tout $n \in \mathbb{N}$, $M_I(n) = \sum_{k=0}^{n-1} M'_I(k)$. En conclusion :

$$\boxed{M_I(0) = 0 \text{ et pour tout } n \geqslant 1, M_I(n) = n - 1}$$

Remarque. Le pire cas (resp. meilleur cas) est atteint lorsque la liste est décroissante (resp. croissante).

Question II.B.1) – On a : $\begin{cases} m_0 = 0 \\ \forall k \geqslant 0, m_{k+1} = 2m_k + 1 \end{cases}$

Par une récurrence immédiate sur k, on obtient :

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

Question II.B.2) –

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

Question II.B.3) –

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

Question II.B.4) –

```
let rec percole a = match a with
  | Vide -> Vide
  | a -> let mini = min_quasi a in
         match a with
         | Noeud(x, _, _) when x = mini -> a
         | Noeud(x, Noeud(x1, b1, b2), a2) when mini = x1 ->
            Noeud(x1, percole (Noeud(x, b1, b2)), a2)
         | Noeud(x, a1, Noeud(x2, b1, b2)) when mini = x2 ->
            Noeud(x2, a1, percole (Noeud(x, b1, b2)))
         | _ -> failwith "percole: cas impossible";;
```

Rappel. Pour les calculs de complexités, l'énoncé demande de compter le nombre de comparaisons par la relation d'ordre entre entiers.

Lorsqu'on appelle la fonction `percole` sur un quasi-tas de hauteur k, on a 0 ou 1 appel récursif sur un arbre de hauteur $k-1$. Dans le pire cas, la racine du quasi-tas est son élément maximal et il y a alors $k+1$ appels à `percole`. De plus, sans compter les appels récursifs, chaque appel à `percole` utilise $\Theta(1)$ comparaisons entre entiers. Ainsi :

La complexité de `percole` dans le pire cas est $\Theta(k)$

Question II.C.1) – On a :

$6 = 3 + 3 = m_2 + m_2$ $\quad$ $7 = 7 = m_3$

$8 = 1 + 7 = m_1 + m_3$ $\quad$ $9 = 1 + 1 + 7 = m_1 + m_1 + m_3$

$10 = 3 + 7 = m_2 + m_3$ $\quad$ $27 = 1 + 1 + 3 + 7 + 15 = m_1 + m_1 + m_2 + m_3 + m_4$

$28 = 3 + 3 + 7 + 15 = m_2 + m_2 + m_3 + m_4$ $\quad$ $29 = 7 + 7 + 15 = m_3 + m_3 + m_4$

$30 = 15 + 15 = m_4 + m_4$ $\quad$ $31 = 31 = m_5$

$100 = 3 + 3 + 31 + 63 = m_2 + m_2 + m_5 + m_6$ $\quad$ $101 = 7 + 31 + 63 = m_3 + m_5 + m_6$

Question II.C.2) – On traite les deux cas évoqués dans l'énoncé.

⋆ Si $r \geqslant 2$ et $k_1 = k_2$: $m_{k_1+1} = 2m_{k_1} + 1 = m_{k_1} + m_{k_2} + 1$. Ainsi :

$$1 + n = 1 + m_{k_1} + \ldots + m_{k_r} = m_{k_1+1} + (m_{k_3} + \ldots + m_{k_r})$$

Il reste à montrer que le uplet $u = (k_1 + 1, k_3, \ldots, k_r)$ vérifie la condition QSC. On note $r' = r - 1$ le nombre d'éléments dans u :

→ Si $r = 2$, alors $r' = 1$. Donc u vérifie la première condition de la propriété QSC.

→ Si $r = 3$, alors $r' = 2$ et comme $k_1 < k_3$, on a $k_1 + 1 \leqslant k_3$. Donc u vérifie la deuxième condition de la propriété QSC.

→ Sinon, on a $r' \geqslant 3$ et $k_3 < k_4 < \ldots < k_r$. Comme $k_1 < k_3$, on a $k_1 + 1 \leqslant k_3$. Donc u vérifie la troisième condition de la propriété QSC.

$\star$ Sinon, $r < 2$ ou $k_1 \neq k_2$.
On a bien $n + 1 = 1 + (m_{k_1} + \ldots + m_{k_r}) = m_1 + m_{k_1} + \ldots + m_{k_r}$
De plus, $1 \leqslant k_1 < k_2 < \ldots < k_r$, donc $(1, k_1, k_2, \ldots, k_r)$ vérifie la propriété QSC.

Question II.C.3) –

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

Remarque. Il y a une erreur d'énoncé au début de la partie $II.D$: « a désigne un arbre binaire parfait » doit être remplacé par « a désigne un tas binaire parfait ».

Question II.D.1.a) – Soit $h = ((a_1, t_1), \ldots, (a_r, t_r))$ une liste de tas de taille $|h| > 0$. Soit i l'indice tel que $\text{haut}(h) = \text{haut}(a_i)$. Comme a_i est un arbre binaire parfait, d'après la question II.B.1), on a $|a_i| = 2^{\text{haut}(a_i)} - 1$. Ainsi :

$$\text{haut}(h) = \text{haut}(a_i) = \log_2(|a_i| + 1) \leqslant \log_2(|h| + 1) = \mathcal{O}(\log_2(|h|)).$$

Si h est une liste non vide de tas, on a nécessairement $\text{haut}(h) = \mathcal{O}(\log_2(|h|))$

En revanche, la deuxième égalité n'est pas nécessairement vérifiée. Par exemple, si h est une liste de r arbres réduits à une feuille, on obtient $\text{long}(h) = r = |h|$.

Si h est une liste non vide de tas, on n'a pas nécessairement $\text{long}(h) = \mathcal{O}(\log_2(|h|))$

Remarque. Dans la question ci-dessus, j'ai supposé qu'une liste de tas ne peut pas contenir l'arbre vide. Même si cette condition n'est pas écrite explicitement dans l'énoncé, autoriser l'un des a_i à être vide pose problème pour la définition de $\min_{\mathcal{H}}(h)$ (car $\min_{\mathcal{A}}(\texttt{Vide})$ n'est pas défini) et contredit la remarque « toute liste de tas de la forme $h = ((a, |a|))$ vérifie la condition TC ».

Question II.D.1.b) – Avec la condition TC, montrons que la deuxième égalité est vérifiée. Si $h = ((a_1, t_1), \ldots, (a_r, t_r))$ vérifie la condition TC, alors $(t_1, \ldots, t_r)$ est QSC. On a donc $t_1 \geqslant m_1 \geqslant m_0, t_2 \geqslant m_1, t_3 \geqslant m_2, \ldots, t_r \geqslant m_{r-1}$. Ainsi :

$$|h| = \sum_{k=1}^{r} |a_k| \geqslant |a_r| = t_r \geqslant m_{r-1} = 2^{r-1} - 1$$

On obtient :

$$\text{long}(h) = r \leqslant \log_2(|h| + 1) + 1 = \mathcal{O}(\log_2(|h|))$$

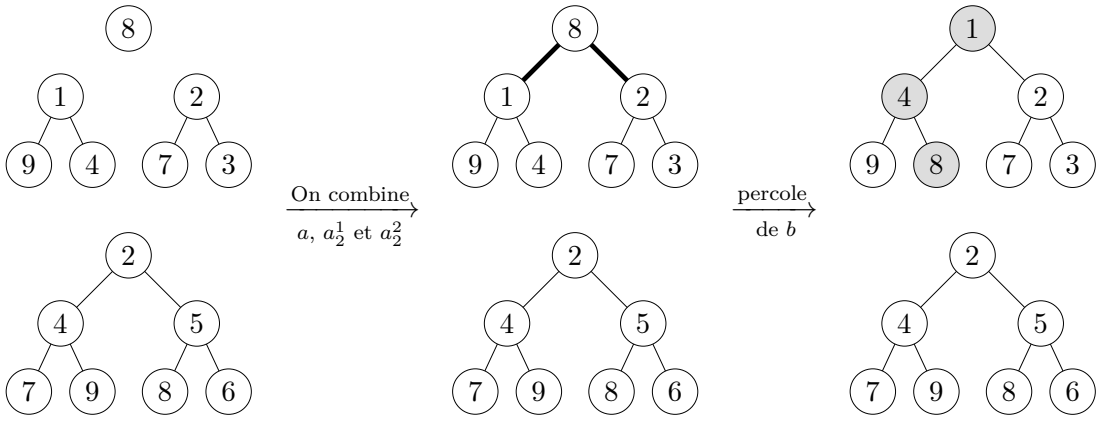
Si h est une liste non vide de tas vérifiant la condition TC, on a nécessairement $\begin{cases} \text{haut}(h) = \mathcal{O}(\log_2(|h|)) \\ \text{long}(h) = \mathcal{O}(\log_2(|h|)) \end{cases}$

Question II.D.2.a) – Lorsqu'on ajoute a en tête de h_1, on obtient directement une liste de tas vérifiant la condition TC. En effet $12 = 1 + 1 + 3 + 7$ est une décomposition parfaite de 12.

$h'_1 = ((a, 1), (a^1_1, 1), (a^2_1, 3), (a^3_1, 7))$ convient.

Lorsqu'on ajoute $(a, 1)$ en tête de h_2, on obtient une liste de tas qui ne vérifie pas la condition TC car $14 = 1 + 3 + 3 + 7$ n'est pas une décomposition parfaite. Grâce à la question II.C.2), on obtient la décomposition parfaite $14 = 7 + 7$. On construit une liste d'arbres correspondant à cette décomposition parfaite de la manière suivante :

- Soit b l'arbre de taille 7 dont la racine est 8, dont le fils gauche est a_2^1 et dont le fils droit est a_2^2. Notez que b est un quasi-tas.
- Soit c l'arbre obtenu après application de la fonction `percole` à b.
- La suite $h'_2 = ((c, 7), (a_2^3, 7))$ est une liste de tas vérifiant la condition TC.



a, a_2^1, a_2^2 et a_2^3 | b et a_2^3 | c et a_2^3

$h'_2 = ((c, 7), (a_2^3, 7))$ convient

Question II.D.2.b) –

⋆ Soit $h = ((a_1, t_1), \ldots, (a_r, t_r))$ une liste de tas vérifiant la condition TC dans lequel on souhaite ajouter un arbre a réduit à sa racine. On reprend les cas de la question II.C.2) :

- Si $r < 2$ ou $t_1 \neq t_2$, on ajoute a au début de la liste pour obtenir la liste de tas $h' = ((a, 1), (a_1, t_1), \ldots, (a_r, t_r))$ qui vérifie la condition TC.
- Sinon :
  - On construit b le quasi-tas dont la racine est égale à la racine de a, dont le fils gauche est a_1 et dont le fils droit est a_2.
  - On applique la fonction `percole` à b pour obtenir un arbre c.
  - Alors, la liste de tas $h' = ((c, 2t_1 + 1), ((a_3, t_3), \ldots, (a_r, t_r)))$ vérifie la condition TC.

⋆ Pour la correction, on remarque d'abord que h' contient les mêmes éléments que $(a, 1) :: h$. Il s'agit donc de montrer que la liste h' est bien une liste de tas et vérifie la condition TC. La suite des tailles de h' vérifie la condition QSC d'après la question II.C.2). De plus, on vérifie facilement que pour tout couple (a, t) de h, l'arbre a est un tas binaire et $t = |a|$.

L'algorithme est correct : h' contient les mêmes éléments que $(a, 1) :: h$ et est une liste de tas vérifiant la condition TC.

⋆ Pour la complexité dans le cas le pire, l'appel à la fonction `percole` s'exécute en temps $\mathcal{O}(\text{haut}(b))$ (voir question II.B.4)). Or, $\text{haut}(a_1) = \text{haut}(b) - 1$.

La complexité dans le cas le pire de cet algorithme est en $\mathcal{O}(\text{haut}(a_1))$

Question II.D.2.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;;
```

Question II.D.3.a) – On remarque que lors de l'évaluation de « `constr_liste (x :: r)` », la fonction s'appelle récursivement sur `r` pour obtenir une liste de tas notée `h`, puis ajoute `x` dans `h` avec la fonction `ajoute`. Ainsi, tous les appels de la forme « `ajoute x h` » sont tels que `x` est inférieur ou égal à tous les éléments de `h`. Dans la fonction `ajoute`, l'appel éventuel à la fonction `percole` s'exécute donc en temps constant. Finalement, chaque appel à `ajoute` est en temps constant.

Pour `constr_liste_tas`, si on note C_n le coût dans le cas le pire pour une liste de taille n, alors il existe une constant $K \in \mathbb{R}_+^*$ telle que :

$$\begin{cases} C_0 \leqslant K \\ \forall n \in \mathbb{N} : C_{n+1} \leqslant C_n + K \end{cases}$$

On obtient $C_n \leqslant K(n+1) = \mathcal{O}(n)$ pour tout $n \in \mathbb{N}$.

Si la liste en entrée est déjà triée, le coût en temps de la fonction `constr_liste_tas` dans le cas le pire est $\mathcal{O}(n)$

Question II.D.3.b) – On note D_n la complexité pire cas de `constr_liste_tas` pour une liste de taille n. Alors, il existe une constante $K \in \mathbb{R}_+^*$ telle que :

$$\begin{cases} D_0 \leqslant K \\ \forall n \in \mathbb{N} : D_{n+1} \leqslant D_n + T_n + K \end{cases}$$

avec T_n la complexité de `ajoute` dans la cas le pire pour une liste de taille n. Ainsi :

$$\forall n \in \mathbb{N} : D_n \leqslant K(n+1) + \sum_{k=0}^{n-1} T_k$$

Pour tout $k \in [\![0, n-1]\!]$, d'après la question II.D.2.b), $T_k = \mathcal{O}(\text{haut}(a_1))$ avec a_1 le premier arbre de la liste h en entrée de `ajoute`. D'après la question II.D.1.b) :

$$\text{haut}(a_1) \leqslant \text{haut}(h) = \mathcal{O}(\log_2(|h|)) \qquad \text{et} \qquad |h| = k \leqslant n$$

Finalement $T_k = \mathcal{O}(\log_2 n)$ et donc :

$$D_n = \mathcal{O}(n) + \mathcal{O}(n \log_2 n) = \mathcal{O}(n \log_2 n)$$

Pour une liste de taille n, `constr_liste_tas` a une complexité temporelle dans le cas le pire en $\mathcal{O}(n \log_2 n)$.

Question II.E.1) –

```
let echange_racines a1 a2 = match a1, a2 with
  | Vide, _ | _, Vide -> failwith "echange_racines: l'un des deux arbres est vide"
  | Noeud(x1, g1, d1), Noeud(x2, g2, d2) -> Noeud(x2, g1, d1), Noeud(x1, g2, d2);;
```

Question II.E.2.a) – Si `a` est un quasi-tas de taille t, alors « `percole a` » est un tas binaire parfait de taille t dont la racine est $\min_{\mathcal{A}}(a)$. Comme h vérifie la condition RO, on a $\min_{\mathcal{A}}(a) \leqslant \min_{\mathcal{A}}(a_1) \leqslant \ldots \leqslant \min_{\mathcal{A}}(a_r)$.

« `(percole a, t) :: h` » est une liste de tas vérifiant RO

Question II.E.2.b) – Dans cette question, étant donné un arbre a, on note $R(a)$ la racine de a.

Comme a_1 est un tas binaire parfait, $R(a_1) = \min_{\mathcal{A}}(a_1)$. Ainsi, $R(b) = R(a_1) = \min_{\mathcal{A}}(a_1) < \min_{\mathcal{A}}(a)$. De plus, le fils gauche et le fils droit de b sont égaux à ceux de a et sont donc des tas binaires parfaits ne contenant que des éléments supérieurs à $\min_{\mathcal{A}}(a) > R(b)$. Finalement :

$$\boxed{b \text{ est un tas binaire parfait}}$$

Les fils gauche et droit de b_1 sont des tas binaires parfaits de même taille car ce sont les fils gauche et droit de a_1 qui est un tas binaire parfait.

$$\boxed{b_1 \text{ est un quasi-tas.}}$$

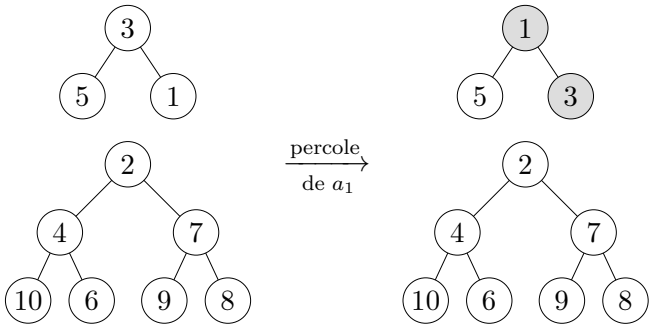
Toute étiquette e de b qui n'est pas à la racine est une étiquette de a et vérifie donc $e \geqslant \min_{\mathcal{A}}(a) > \min_{\mathcal{A}}(a_1)$. Comme $R(b) = R(a_1) = \min_{\mathcal{A}}(a_1)$, on a :

$$\boxed{\min_{\mathcal{A}}(b) = \min_{\mathcal{A}}(a_1)}$$

Toute étiquette e de b_1 qui n'est pas à la racine est une étiquette de a_1 et vérifie donc $e \geqslant \min_{\mathcal{A}}(a_1)$. Comme $R(b_1) = R(a) = \min_{\mathcal{A}}(a) > \min_{\mathcal{A}}(a_1)$, on a :

$$\boxed{\min_{\mathcal{A}}(a_1) \leqslant \min_{\mathcal{A}}(b_1)}$$

Question II.E.3) – Pour le couple (a_1, h_1). On lance une percolation sur l'arbre a_1 pour obtenir deux arbres b_1 et $b_1^1 = a_1^1$:

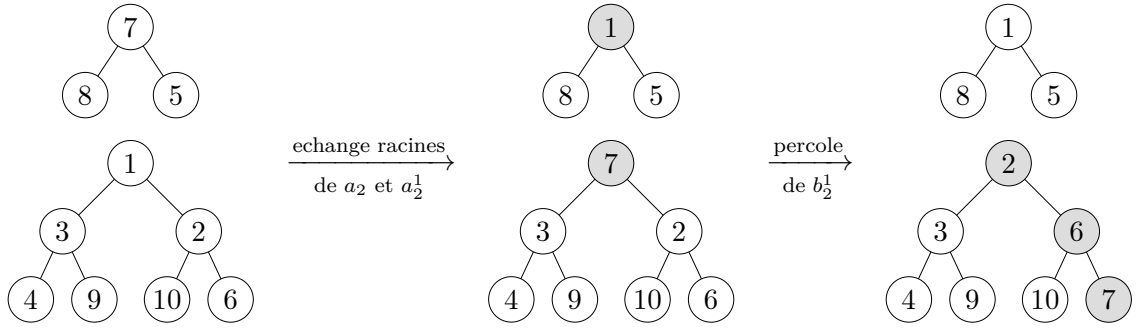


a_1 et a_1^1 $\qquad$ b_1 et b_1^1

On a $\min_{\mathcal{A}}(b_1) = 1 \leqslant 2 = \min_{\mathcal{A}}(b_1^1)$ donc :

$$\boxed{\text{La liste } ((b_1, 3), (b_1^1, 7)) \text{ est une liste de tas vérifiant la condition RO.}}$$

Pour le couple (a_2, h_2). On échange les racines de a_2 et a_2^1 pour obtenir deux arbres b_2 et b_2^1. On lance ensuite une percolation sur b_2^1 pour obtenir deux arbres $c_2 = b_2$ et c_2^1.

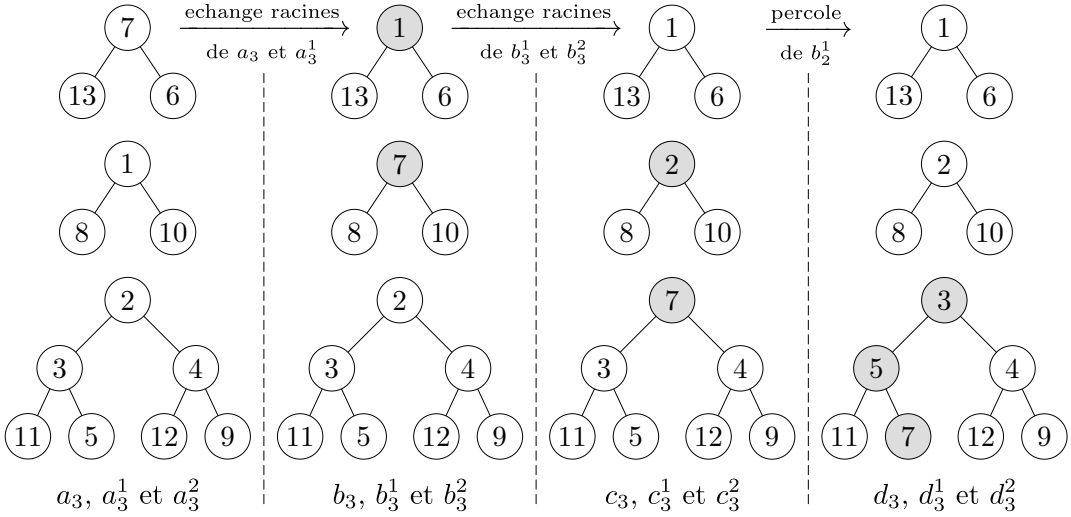


a_2 et a_2^1 $\qquad$ b_2 et b_2^1 $\qquad$ c_2 et c_2^1

On a $\min_{\mathcal{A}}(c_2) = 1 \leqslant 2 = \min_{\mathcal{A}}(c_2^1)$ donc :

La liste $((c_2, 3), (c_2^1, 7))$ est une liste de tas vérifiant la condition RO.

Pour le couple (a_3, h_3). On échange les racines de a_3 et a_3^1 pour obtenir trois arbres b_3, b_3^1 et $b_3^2 = a_3^2$. On échange ensuite les racines de b_3^1 et b_3^2 pour obtenir trois arbres $c_3 = b_3$, c_3^1 et c_3^2. Pour terminer, on lance une percolation sur c_3^2 pour obtenir trois arbres $d_3 = c_3$, $d_3^1 = c_3^1$ et d_3^2.



On a $\min_{\mathcal{A}}(d_3) = 1 \leqslant 2 = \min_{\mathcal{A}}(d_3^1) \leqslant 3 = \min_{\mathcal{A}}(d_3^2)$ donc :

La liste $((d_3, 3), (d_3^1, 3), (d_3^2, 7),)$ est une liste de tas vérifiant la condition RO.

Question II.E.4) – On utilise un algorithme récursif :

> **Entrée.** Un quasi-tas a, la taille de a notée t et une liste de tas $h = ((a_1, t_1), \ldots, (a_r, t_r))$ vérifiant RO.
> **Sortie.** La liste de tas h' demandée dans l'énoncé.
> **si** h est vide ou $\min_{\mathcal{A}}(a) \leqslant \min_{\mathcal{A}}(a_1)$ **alors**
> soit b le tas parfait renvoyé par un appel à `percole` sur a
> **renvoyer** $(b, t) :: h$
> **sinon** h s'écrit $h = (a_1, t_1) :: q$ avec $\min_{\mathcal{A}}(a) > \min_{\mathcal{A}}(a_1)$
> soit (b, b_1) le couple d'arbres renvoyé par `echange_racines` avec a et a_1 en arguments
> soit q' la liste de tas parfaits obtenue par un appel récursif avec b_1, t_1 et q en arguments
> **renvoyer** $(b, t) :: q'$
> **fin si**

Pour la correction, on remarque que la fonction fait appel à `echange_racines` et à `percole`, donc la liste renvoyée, notée h', est identique à $(a, t) :: h$ à permutation près des étiquettes des arbres. Il reste à montrer que si h est une liste de tas alors h' aussi et que si h vérifie RO alors h' aussi. Montrons ces deux propriété par induction structurelle sur h.

Si h est vide, comme a est un quasi-tas, l'appel à `percole` renvoie un tas binaire b parfait contenant les mêmes éléments que a. Ainsi, la liste $h' = (b, t) :: h$ renvoyée par la procédure est bien une liste de tas binaires vérifiant RO.

Si $h = (a_1, t_1) :: q$, on suppose que la procédure est correcte pour q et on le montre pour h. Si $\min_{\mathcal{A}}(a) \leqslant \min_{\mathcal{A}}(a_1)$, la question II.E.2.a) assure la correction de la procédure. Sinon $\min_{\mathcal{A}}(a) > \min_{\mathcal{A}}(a_1)$, la question II.E.2.b) assure que l'arbre b (défini dans le pseudo-code) est un tas binaire parfait, que l'arbre b_1 est un quasi-tas et que $\min_{\mathcal{A}}(b) \leqslant \min_{\mathcal{A}}(b_1)$. Par l'hypothèse de récurrence, la liste q' obtenue par l'appel récursif est une liste de tas, donc la liste $h' = (b, t) :: q'$ renvoyée est aussi une liste de tas. De plus, si h vérifie RO, alors q vérifie RO et donc q' vérifie RO par l'hypothèse de récurrence. Comme $\min_{\mathcal{A}}(b) = \min_{\mathcal{A}}(a_1)$ et que h vérifie RO, la racine de b qui vaut $\min_{\mathcal{A}}(b)$ est inférieure à tous les éléments de h et de a, donc à tous les éléments de q'. Finalement, comme q' vérifie RO, la liste h' vérifie RO.

La procédure est correcte.

Pour la complexité, si a est un tas non vide, que h vérifie RO et que $\min_{\mathcal{A}}(a) \leqslant \min_{\mathcal{H}}(h)$, alors la procédure renvoie (`percole a, |a|) :: h`. Comme a est déjà un tas, l'appel à `percole` s'exécute en temps constant.

Dans ce cas, on a une complexité en temps en $\mathcal{O}(1)$

Dans le cas général, la procédure lance au plus $r+1$ appels récursifs (en comptant l'appel principal). Toutes les opérations sont en temps constant, sauf l'unique appel à `percole` qui en temps $\mathcal{O}(k)$.

La complexité en temps de la procédure est en $\mathcal{O}(k+r)$

Question II.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
                     let q' = insere_quasi b1 t1 q in
                     (b, t) :: q';;
```

Question II.E.6) –

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

Question II.E.7) – Lors de l'appel à `tri_racines` sur une liste de tas h, il y a exactement $\text{long}(h)+1$ appels récursifs (en comptant l'appel principal) sur des listes de tas h' telles que $|h'| \leqslant |h|$. Pour chacun de ces appels récursifs, d'après la question II.E.4), la complexité de l'appel à `insere_quasi` est $T_{h'} = \mathcal{O}(k+r)$ avec $k = \text{haut}(h') \leqslant \text{haut}(h)$ et $r = \text{long}(h') - 1 \leqslant \text{long}(h)$. Avec la question II.D.1.b), on obtient $T_{h'} = \mathcal{O}(\log_2(|h|))$. En comptant les $\text{long}(h)+1 = \mathcal{O}(\log_2(|h|))$ appels à `insere_quasi`, on conclut :

La fonction `tri_racines` a une complexité temporelle en $\mathcal{O}\Big((\log_2|h|)^2\Big)$

Question II.F.1) – Comme h vérifie RO, a_1 et a_2 sont des tas (donc des quasi-tas) et h' vérifie RO. En appliquant deux fois le résultat de la question II.E.4), la liste de tas h'' vérifie RO.

Pour TC, soit $h''' = (a_1, |a_1|) :: (a_2, |a_2|) :: h'$. Étant donné que la suite des tailles des arbres de h'' est la même que pour h''', il suffit de vérifier que h''' vérifie TC. C'est bien le cas puisque $|a_1| = |a_2| < t$ et que h vérifie TC.

Question II.F.2) – Par la question II.E.4), la complexité temporelle est en $\mathcal{O}(k_2 + r_2 + k_1 + r_1)$ avec :

$$\begin{cases} k_2 = \max\Big(\text{haut}(a_2), \text{haut}(h')\Big) \leqslant \text{haut}(h) = \mathcal{O}(\log_2|h|) \\ r_2 = \text{long}(h') = \text{long}(h) - 1 = \mathcal{O}(\log_2|h|) \\ k_1 = \max\Big(\text{haut}(a_1), \text{haut}((a_2, |a_2|) :: h')\Big) \leqslant \text{haut}(h) = \mathcal{O}(\log_2|h|) \\ r_1 = \text{long}((a_2, |a_2|) :: h') = \text{long}(h) = \mathcal{O}(\log_2|h|) \end{cases}$$

La complexité temporelle est en $\mathcal{O}(\log_2|h|)$

Question II.F.3) –

```
let rec extraire h = match h with
  | [] -> []
  | (Noeud(x, Vide, Vide), 1) :: h' -> x :: extraire h'
  | (Noeud(x, a1, a2), t) :: h' ->
     let t' = t/2 in (* ou (t-1)/2 *)
     let q = insere_quasi a1 t' (insere_quasi a2 t' h') in
     x :: extraire q
  | _ -> failwith "extraire: cas impossible";;
```

Question II.F.4) – Lors de l'évaluation de « `extraire h` », la fonction `extraire` est appelée $|h|+1$ fois sur des listes de taille $|h|, |h|-1, |h|-2, \ldots, 1, 0$. Pour chacune de ces listes h', d'après la question II.F.2), le temps d'exécution sans compter les appels récursifs est en $\mathcal{O}(\log_2 |h'|) = \mathcal{O}(\log_2 |h|)$. Au total :

La fonction `extraire` a une complexité en $\mathcal{O}\Big(|h| \log_2 |h|\Big)$ dans le pire cas

Question II.G.1) –

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

Question II.G.2) – D'après la question II.D.3.b), la complexité de l'appel à `constr_liste_tas` est en $\mathcal{O}(n \log_2 n)$. D'après la question II.E.7), la complexité de l'appel à `tri_racines` est en $\mathcal{O}((\log_2 n)^2)$ D'après la question II.F.4), la complexité de l'appel à `extraire` est en $\mathcal{O}(n \log_2 n)$. Finalement :

Le temps d'exécution de `tri_lisse` est en $\mathcal{O}(n \log_2 n)$

Question II.G.3) – Si la liste est déjà triée, d'après la question II.D.3.a), la complexité de l'appel à `constr_liste_tas` est en $\mathcal{O}(n)$. De plus, on remarque que la liste de tas obtenue après l'appel à `constr_liste_tas` notée $h = ((a_1, t_1), \ldots, (a_r, t_r))$ vérifie les deux propriétés suivantes :

- Pour tout i, tous les éléments de a_i sont inférieurs ou égaux à $\min_{\mathcal{A}}(a_{i+1})$.
- Pour tout i, les éléments de a_i lus dans l'ordre du parcours préfixe sont triés.

Notons en particulier que ces deux propriétés sont conservées lorsque la fonction `ajoute` est appliquée sur une liste $h = ((a_1, t_1), (a_1, t_2), \ldots)$ avec $t_1 = t_2$ et lorsque la fonction `extraire` construit « `insere_quasi a1 |a1| (insere_quasi a2 |a2| h')` ».

Ainsi, l'appel à `tri_racines` sur h renvoie h en temps $\mathcal{O}((\log_2 n)^2)$. De plus, lors de l'exécution de `extraire`, tous les appels à `insere_quasi` s'exécutent en $\mathcal{O}(1)$ d'après le résultat de la question II.E.4). Au total, la complexité de l'appel à `extraire` est en $\mathcal{O}(n)$. En résumé :

Si la liste passée en argument est triée, la complexité de `tri_lisse` est en $\mathcal{O}(n)$

Question III.A) – La fonction `tri_lisse` créé une liste de tas contenant tous les éléments de la liste initiale. Le quantité de mémoire utilisée pour stocker cette liste est linéaire en le nombre d'éléments. D'où :

La complexité spatiale de `tri_lisse` est un $\Omega(n)$

Question III.B) –

```
(* ou taille = (a.taille-1)/2 *)
let fg a =
  {donnees = a.donnees; pos = a.pos + 1; taille = a.taille/2};;
```

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

Question III.C) –

```
(* a supposé non vide *)
let min_tas_vect a =
  a.donnees.(a.pos);;
```

```
(* Un quasi-tas est non vide par définition *)
let min_quasi_vect a =
  let m1 = a.donnees.(a.pos) in
  if a.taille = 1 then m1 else
    min m1 (min (min_tas_vect (fg a)) (min_tas_vect (fd a)));;
```

Question III.D) –

```
let rec percole_vect a =
  if a.taille <> 0 then
    let mini = min_quasi_vect a in
    let a1 = fg a and a2 = fd a in
    match () with
    | () when mini = a.donnees.(a.pos) -> ()
    | () when mini = min_tas_vect a1 -> a1.donnees.(a1.pos) <- a.donnees.(a.pos);
                                        a.donnees.(a.pos) <- mini;
                                        percole_vect a1;
    | _ (* mini = min_tas_vect a2 *) -> a2.donnees.(a2.pos) <- a.donnees.(a.pos);
                                        a.donnees.(a.pos) <- mini;
                                        percole_vect a2;;
```

Question III.E) –

Remarque. Il y a un problème de type dans l'énoncé : la fonction `constr_liste_tas_vect` appelle la fonction `constr_liste_tas_aux` qui appelle la fonction `ajoute_vect` avec `h = []`. J'imagine que :

`ajoute_vect: int array -> int -> tasbin list -> tasbin list.`

De plus, il y a deux erreurs d'énoncé dans la définition de `constr_liste_tas_aux` et de `constr_liste_tas_vect` :

```
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';; (* Erreur d'énoncé h -> h' *)
```

```
let constr_liste_tas_vect d =
  constr_liste_tas_aux d (Array.length d) [];;
(* Erreur d'énoncé: constr_liste_tas -> constr_liste_tas_aux *)
```

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

Question III.F) –

```
let echange_racines a1 a2 =
  let tmp = a1.donnees.(a1.pos) in
  a1.donnees.(a1.pos) <- a2.donnees.(a2.pos);
  a2.donnees.(a2.pos) <- tmp;;
```

Question III.G) – Il semble y avoir une erreur d'énoncé dans le type de la fonction :

```
insere_quasi_vect : tasbin -> tasbin list -> tasbin list
```

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

Question III.H) – Il semble y avoir une erreur d'énoncé dans le type de la fonction :

```
tri_racines_vect : tasbin list -> tasbin list
```

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

Question III.I) –

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

Question III.J) –

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

Question III.K) – Lorsqu'on compare l'implémentation de la partie III avec celle de la partie II, on se rend compte que la structure de données change, mais que le déroulement de l'algorithme est le même. En particulier les fonctions ont le même temps d'exécution :

La complexité temporelle de `tri_lisse_vect` dans le cas le pire est en $\mathcal{O}(n \log_2 n)$

Question III.L) – Pour la même raison :

La complexité temporelle de `tri_lisse_vect` pour un tableau déjà trié est en $\mathcal{O}(n)$

Question III.M) – Comme expliqué dans l'énoncé, la place mémoire occupée par une liste de tas est linéaire en sa longueur. Puisque toutes les listes de tas manipulées vérifient TC, leurs longueurs sont en $\mathcal{O}(log_2 n)$ (voir question II.D.1.b)). Ainsi :

La complexité spatiale de `tri_lisse_vect` dans le cas le pire est en $\mathcal{O}(\log_2 n)$