1 Présentation

RAS

2 Algorithme sur les arbres

2.1 Tri par insertion

1.
```
let rec insere a l =
  match l with
    |[] → [a]
    |b :: tl → if a < b then a :: l else b :: (insere a tl)
;;
```

2.
```
let rec tri_insertion l =
  match l with
    |[] → []
    |a :: tl → insere a (tri_insertion tl)
;;
```

3. Ecrit comme cela, le tri par insertion va parcourir la liste puis insérer chaque élément dans une liste initialement vide. Ainsi, dès que cette liste aura son premier élément dedans, l'appel d'insertion va nécessité au moins une comparaison et une seule si l'élément à insérer est le plus petit, ce qui arrive à chaque fois si la liste en entrée est déjà triée dans l'ordre croissant. Donc $M_I(n) = n - 1$. Inversement, si la liste en entrée est triée dans l'ordre décroissant, chaque appel d'insertion devra parcourir la totalité de la liste en train d'être crée, faisant ainsi le maximum de comparaisons possibles à chaque passage. On aura donc $P_I(n) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$.

2.2 Tas binaires

1. D'après la définition, si un arbre binaire parfait n'est pas vide, il est clairement composé de sa racine et de deux arbres binaires parfait d'une hauteur de moins. Ainsi on a la relation $m_{k+1} = 1 + 2m_k$ pour tout $k \in \mathbb{N}$ et $m_0 = 0$. On peut ensuite, soit en utilisant une récurrence, soit en cherchant le point fixe de cette suite arithmético-géométrique, conclure que $m_k = 2^k - 1$ pour tout $k \in \mathbb{N}$.

2.
```
let min_tas a =
  match a with
    |Vide → failwith
    |Noeud(x, fg, fd) → x
;;
```

3.
```
let min_quasi a =
  match a with
    |Vide → failwith
    |Noeud(x,Vide,Vide) → x
    |Noeud(x, fg, fd) → min x (min (min_tas fg) (min_tas fd))
;;
```

4.
```
let rec percole a =
  match a with
    |Vide → Vide
    |Noeud(x,Vide,Vide) → a
    |Noeud(x,Noeud(y, fgg, fgd),Noeud(z, fdg, fdd)) → if min_quasi a = x then a
        else if min_quasi a = y then Noeud(y, percole (Noeud(x, fgg, fgd),Noeud(z, fdg, fdd))
        else Noeud(z,Noeud(y, fgg, fgd) , percole (Noeud(x, fdg, fdd))
;;
```

Dans le pire des cas la racine est en fait plus grande que chacun des descendants auxquels elle va être comparé et on devra donc la descendre jusqu'à une feuille, rappelant ainsi la fonction percole k fois.

2.3 Décomposition parfaite d'un entier

1. — $6 = 3 + 3$
   — $7 = 7$

— $8 = 1 + 7$
— $9 = 1 + 1 + 7$
— $10 = 3 + 7$
— $27 = 1 + 1 + 3 + 7 + 15$
— $28 = 3 + 3 + 7 + 15$
— $29 = 7 + 7 + 15$
— $30 = 15 + 15$
— $31 = 31$
— $100 = 3 + 3 + 31 + 63$
— $101 = 7 + 31 + 63$

2. — Si $r \geqslant 2$ et $k_1 = k_2$ alors $m_{k_1} + m_{k_2} = 2m_{k_1}$ et par propriété de la suite (m_k) on a $1 + 2m_{k_1} = m_{k_1+1}$, ainsi $n + 1 = m_{k_1+1} + (m_{k_3} + \ldots + m_{k_r})$. Si de plus $r \leqslant 3$ il n'y a rien à ajouter et si $r \geqslant 4$ alors par propriété de la décomposition parfaite de n on a $k_3 < k_4$ et ainsi $m_{k_1+1} + (m_{k_3} + \ldots + m_{k_r})$ forme bien une décomposition parfaite de $n + 1$.
— Sinon, soit $r = 1$ et il n'y a rien à dire, soit $k_1 \neq k_2$ ainsi $1 \leqslant k_1 < k_2 < \ldots < k_r$ et $m_1 + (m_{k_1} + \ldots + m_{k_r})$ sera bien une décomposition parfaite de $n + 1$.

3.
```
let decomp_parfait n =
  let rec boucle l k =
    if k = n then l else
      match l with
        |a :: a :: tl → boucle ((2 * a + 1) :: tl) k + 1
        |_ → boucle 1 :: l k + 1
  in boucle [1] 1
;;
```
Cet algorithme est clairement linéaire en complexité car la boucle est clairement une boucle for (dont la variable est k) et l'opération effectuée à l'intérieur de celle-ci est clairement élémentaire.
(remarque : cette écriture récursive à la place d'une boucle for explicite nous permet d'éviter d'avoir à manipuler une référence sur une liste, qui pourrait s'avérer extrêmement désagréable à effectuer)

2.4 Création d'une liste de tas

1. (a) On sait que pour un tas binaire a on a haut(a) =$\log_2(|a|)$ donc pour une liste de tas h on va clairement avoir haut(h) =$\log_2(|h|)$ d'où haut$(h) = \mathcal{O}(\log_2(|h|))$. par contre, prenons une liste de tas h dont tous les tas n'ont qu'un élément. On a alors long$(h) = r$ et $|h| = r$. Est-ce que $r = \mathcal{O}(\log_2(r))$? La réponse est bien sure : NON.

   (b) On a le résultat sur la hauteur, puisqu'on l'avait déjà pour une liste de tas quelconque. Il s'agit donc de vérifier si la condition TC nous permet de l'acquérir pour la longueur. De plus, vu que l'on cherche à majorer l'asymptotique de la longueur on peut supposer celle-ci supérieur ou égale à 3. Ainsi, si $t_1 + \ldots + t_r$ est une décomposition parfaite alors $t_1 \leqslant t_2 < t_3 < \ldots < t_r$ et donc on a forcément $t_r \geqslant m_{r-1} = 2^{r-1} - 1$. Ainsi la condition TC nous permet d'avoir effectivement long$(h) = \mathcal{O}(\log_2(|h|))$.

2. (a) Pour l'insertion dans la première liste h_1 il suffit d'insérer l'élément en tête de liste. Pour celle dans h_2 c'est plus complexe, car simplement insérer l'élément en tête de liste ferait que les tas en position 2 et 3 auraient le même nombre d'éléments (à savoir 3 chacun) et on ne vérifierait plus la condition TC. Mais la, on se souvient de ce qu'on a fait à la question 2.3.2 sur les décompositions parfaites d'entiers. On prend nos deux premiers arbres de même taille et avec le nouvel élément que l'on insère on crée un tas binaire d'une hauteur plus grande.

   (b) Si il n'y a qu'un élément dans h ou que les deux premiers sont différents il suffit (comme dans l'exemple précédent) d'insérer le nouveau tas en position 1. Sinon, on crée un quasi-tas avec pour racine l'élément que l'on voulait insérer, pour fils gauche le premier tas de h et pour fils droit le second. On applique ensuite percole sur ce quasi-tas afin d'en faire un tas. On aura alors une liste de tas, et celle-ci vérifiera bien la condition TC par le même argument utilisé à la question 2.3.2 sur les décomposition parfaites.
   La complexité quand à elle n'est rien d'autre que la complexité au pire de percole sur le premier tas de la liste h, d'où le résultat.

   (c)
```
let ajoute x h =
  match h with
    |(a, k) :: (b, k) :: tl → (percole Noeud(x, a, b), 2k + 1) :: tl
    |_ → (Noeud(x,Vide,Vide), 1) :: tl
;;
```

3. (a) Si la liste est l est déjà triée (dans l'ordre croissant) alors chaque appel d'ajoute se fera avec un élément déjà plus petit que tous les élément déjà présent dans la liste de tas. Et donc percole ne fera rien. On est donc à chaque fois dans le cas favorable ou ajoute se fera systématiquement en temps constant et donc l'appeler n fois se fera bien en temps linéaire.

   (b) Par contre si l n'est pas triée (voir même triée dans l'ordre décroissant), cette fois percole va potentiellement être toujours dans le pire cas et donc chaque appel d'ajoute se fera en $\mathcal{O}(\log_2(n))$. D'où le résultat.

2.5 Tri des racines

1.
```
let echange_racines a1 a2 =
  match a1, a2 with
    |Noeud(x1, fg1, fd1),Noeud(x2, fg2, fd2) → Noeud(x2, fg1, fd1),Noeud(x1, fg2, fd2)
;;
```

Remarquez bien ici que, puisqu'il est précisé dans la question que $a1$ et $a2$ sont des arbres non vides, il n'est pas nécessaire de mettre des assertion et autres failwith dans le code.

2. (a) si $\min_{\mathcal{A}}(a) \leqslant \min_{\mathcal{A}}(a_1)$ alors percole a va transformer ce quasi tas en un tas dont la racine sera plus petite que la racine du premier tas (déjà censé être la racine minimum par condition RO) de la liste h. D'où le résultat.

   (b) a_1 étant un tas, son minimum est sa racine. Donc si $\min_{\mathcal{A}}(a) > \min_{\mathcal{A}}(a_1)$, cela veut dire que la racine de a_1 est plus petite que la racine de a et les racines de ses éventuels fils. Donc en échangeant les racines de a et a_1 le quasi-tas b que l'on a crée est bien un tas. De plus, a_1 étant un tas à la base, toucher juste à sa racine ne changera pas sa condition de quasi-tas. Il ne nous reste plus qu'à prouver l'inégalité $\min_{\mathcal{A}}(b) \leqslant \min_{\mathcal{A}}(\mathbf{b1})$ avant de retourner écouter notre album de metal favoris. Et c'est la encore quasi immédiat. En effet, l'inégalité $\min_{\mathcal{A}}(a) > \min_{\mathcal{A}}(a_1)$ nous assure que $\min_{\mathcal{A}}(b)$ est déjà plus petit que la racine de $b1$ sauf que les éventuels fils gauche et droit de $b1$ sont les anciens fils de la nouvelle racine de b qui "à l'époque" était un tas, d'où le résultat.

3. Si on a bien compris la question précédente, celle-ci est aussi évidente, il ne s'agit que d'un raisonnement de récursivité. Soit on est dans le cas $\min_{\mathcal{A}}(a) \leqslant \min_{\mathcal{A}}(a_1)$ qui constitue notre cas terminal, on utilise juste percole sur le quasi tas que l'on va ensuite insérer au début et on aura une liste de tas vérifiant la condition RO. Soit on est dans l'autre cas et après avoir échangé les racines, afin d'avoir en tête de liste un tas dont la racine est plus petite que celle des autres tas de la listes et du minimum du quasi-tas $b1$, on ré-applique le procédé en avec la suite de tas $((a_2, t_2), \ldots, (a_r, t_r))$ et le quasi-tas $b1$.

   Pour ce qui est de la complexité, étant donné que la liste h vérifie la condition RO, le $\min_{\mathcal{H}}(h)$ correspond à la racine de son premier tas et donc si effectivement $\min_{\mathcal{A}}(a) \leqslant \min_{\mathcal{H}}(h)$ cela veut dire que l'on est dans le premier cas et donc que l'on va donc juste une fois percole sur un tas (donc cette dernière ne fera rien) et on va s'arrêter. On aura donc bien une complexité constante.

   En revanche, si systématiquement on est dans le second cas, on fera donc r échange de racines, et, de plus, si au dernier moment lorsque l'on va appeler percole on le fait sur le quasi-tas de plus grande hauteur de la liste et que cette dernière est dans son pire cas on va rajouter une complexité de k opération, d'où le résultat.

4.
```
let rec insere_quasi a t h =
  match h with
    |[] → (percole a, t)
    |(a1, t1) :: tl when min_quasi a <= min_tas a1 → (percole a, t) :: h
    |(a1, t1) :: tl → let (b, b1) = (echange_racines a a1) in
        b ::(insere_quasi b1 t1 tl)
;;
```

5.
```
let rec tri_racines h =
  match h with
    |[] → []
    |(a, t) :: tl → insere_quasi a t (tri_racines tl)
;;
```

Comment cela j'ai juste copié coller la correction de la troisième question du sujet sur le tri insertion ?

remarque : Les propriétés d'insere_quasi justifie automatiquement que la liste de sortie vérifiera bien la condition RO, pour ce qui est de la condition TC, il faut s'appuyer sur le fait cet algorithme ne modifie clairement pas la taille des arbres de la listes et comme initialement cette dernière vérifiée la condition TC elle la vérifie toujours en sortie.

6. cette fonction va appeler insere_quasi autant de fois que la longueur de la liste h, sauf que d'après la question 2.4.1.b, comme h vérifie la condition TC cette longueur est bien majoré par du $\log_2(|h|)$. De plus, d'après la question 2.5.3 chaque appel d'insere_quasi est en $\mathcal{O}(k+r)$ sauf que toujours d'après la question 2.4.1.b aussi bien k que r est en $\mathcal{O}(\log_2(|h|)$, d'où le résultat.

2.6 Extraction des éléments d'une liste de tas

1. Si h vérifie RO et TC alors h' aussi, de plus le fait de composer deux insere_quasi ne changera pas le respect de la condition RO par construction de cette dernière fonction. Il s'agit donc uniquement de justifier qu'en insérant les fils du premier tas dans le reste de la liste on continue de respecter la condition TC, ce qui est évident puisque la taille de ces derniers est inférieur strictement à la taille du premier tas de la liste mentionné.
2. Dans le pire des cas, chacun des deux appels d'insere_quasi fera le maximum de dégât possible, c'est à dire le fameux $\mathcal{O}(k+r)$ de la question 2.5.3 lui même majoré par le résultat de la question 2.4.1.b. Par conséquent la fonction f dont il est question ici n'est autre que le $\log_2$.
3.
```
let rec extraire h =
  match h with
    |[] → []
    |(Noeud(x,Vide,Vide), 1) :: tl → x ::extraire tl
    |(Noeud(x, a1, a2), t) :: tl → x ::(extraire (insere_quasi a1 ((t − 1)/2) (insere_quasi a2 ((t − 1)/2) tl)))
;;
```
4. Le pire cas de ce match est le troisième qui donne lieu aux appels d'insere_quasi en $\log_2(|h|)$. mais ce cas ne peut se répéter plus que le nombre d'éléments dans notre liste de tas d'où le résultat.

2.7 Synthèse

1.
```
let tri_lisse l =
  extraire (tri_racines (constr_liste_tas l))
;;
```
2. D'après la question 2.4.3.b l'appel de constr_liste_tas est en $\mathcal{O}(n\log_2(n))$. D'après la question 2.5.6 l'appel de tri_racines est en $\mathcal{O}(\log_2(n)^2)$ (donc c'est encore mieux). Enfin, d'après la question 2.6.4 l'appel d'extraire est en $\mathcal{O}(n\log_2(n))$, d'où le résultat.
3. Cette fois on invoque les résultats des questions 2.4.3.a et 2.5.3 pour nous dire que les appels de constr_liste_tas et tri_racines seront bien linéaire (le premier immédiatement et le second parce que les sous appels des insere_quasi se feront en temps constant. Il n'y a donc plus qu'à justifier que les sous appels d'insere_quasi dans extraire se feront aussi en temps constant pour avoir la linéarité de la complexité de notre algorithme dans le meilleur cas. Et ça, c'est presque évident, parce que dans ce cas la, à chaque étape de la construction de la liste de tas on avait une liste de tas vérifiant la condition RO (et pas seulement la condition TC, c'est à dire que dans ce cas la, l'appel de tri_ racines est même inutile) et donc au moment de la déconstruire par extraction les appels d'insere_quasi se feront aussi immédiatement. Donc au final sur une liste déjà triée le tri lisse se fait en temps linéaire.