X 2014 – Arbres croissants – Corrigé

1 Structure d'arbre croissant

1. La première question ne pose pas de difficulté, on vérifie la bonne compréhension de la structure d'arbre croissant et la connaissance des bases de Caml :

```
let minimum = function
  | E -> failwith "Arbre vide"
  | N (_, x, _) -> x
```

2. Pour qu'un arbre non-vide soit croissant, il faut et suffit qu'il soit une feuille, ou que ses sous-arbres soient croissants, et que l'élément à la racine soit plus petit que les éléments à la racine des sous-arbres non réduits à une feuille :

```
let rec est_un_arbre_croissant = function
  | E -> true
  | N (g, x, d) ->
        (g = E || x <= minimum g) && est_un_arbre_croissant g
     && (d = E || x <= minimum d) && est_un_arbre_croissant d
```

3. Dans un arbre croissant étiqueté par les entiers de 1 à n, le 1 se trouve nécessairement dans la racine. Les autres entiers peuvent indifféremment se trouver dans le sous-arbre droit et le sous-arbre gauche.

S'il y a p entiers dans le sous-arbre gauche et $q = n-1-p$ dans le sous-arbre droit, le nombre de sous-arbres gauches possibles est le même que le nombre d'arbres croissants étiquetés par les entiers de 1 à p (resp. q à droite). Le nombre d'arbres croissants étiquetés par les entiers de 1 à n vérifie donc :

$$\mathcal{N}(n) = \sum_{p=0}^{n-1} \binom{n-1}{p} \mathcal{N}(p)\,\mathcal{N}(n-1-p) = \sum_{p=0}^{n-1} \frac{(n-1)!}{p!(n-1-p)!}\,\mathcal{N}(p)\,\mathcal{N}(n-1-p)$$

Un calcul pour de petits n donne $\mathcal{N}(1) = 1$, $\mathcal{N}(2) = 2$ (le 2 peut être le fils gauche ou le fils droit), $\mathcal{N}(3) = 6$ (deux arbres de hauteur 2, deux abres avec seulement un fils gauche au niveau de la racine, deux arbres avec seulement un fils droit au niveau de la racine), et $\mathcal{N}(4) = 24$. Montrons par récurrence que $\mathcal{N}(n) = n!$.

C'est vrai pour $n = 0$ (l'arbre vide) et $n = 1$ (un unique arbre étiqueté par 1).

Supposons que $\mathcal{N}(k) = k!$ pour tout entier k inférieur strictement à n. On a alors

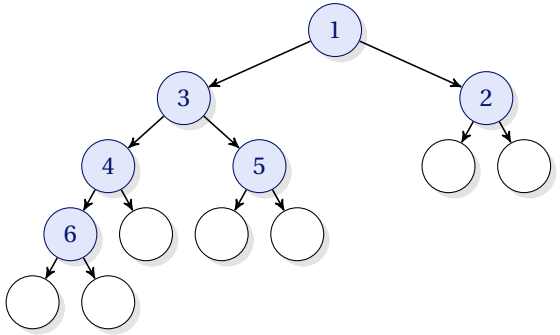
$$\mathcal{N}(n) = \sum_{p=0}^{n-1} \frac{(n-1)!p!(n-1-p)!}{p!(n-1-p)!} = \sum_{p=0}^{n-1} (n-1)! = n \times (n-1)! = n!$$

C'est donc également vrai pour $k = n$. Par récurrence, on a donc bien, pour tout entier strictement positif k, $\mathcal{N}(k) = k!$.

Alternativement, on peut, toujours par récurrence, voir qu'un arbre croissant construit avec les entiers 1 à n est nécessairement tel que le nœud étiqueté par n ne peut avoir que `E` comme fils. En remplaçant ce nœud par un nœud `E`, on obtient un arbre croissant étiqueté par les entiers de 1 à $n-1$. Il y a, par récurrence, $(n-1)!$ tels arbres. Or, ces arbres ont $n-1$ nœuds internes, et donc n feuilles `E`. Il y a donc n endroits où le sommet `N(E,n,E)` pouvait se trouver, et donc $n \times (n-1)! = n!$ arbres croissants étiquetés par les entiers de 1 à n.

2 Opérations sur les arbres croissants

4. L'arbre obtenu par la fusion des deux arbres donnés est :



5. On peut montrer ce résultat (à savoir que la fusion d'arbres croissants t_1 et t_2 donne un arbre croissant t, et que les occurrences de x dans t correspondent à la somme des occurrences de x dans t_1 et de x dans t_2) par induction structurelle.

C'est vrai si t_1 =`E`, puisque $t = t_2$ est donc nécessairement un arbre croissant, et le nombre d'occurrences de x dans t_1 étant nulle, celles de x dans t est bien la somme de celles de x dans t_1 (zéro, donc) et dans t_2. Il en est de même si t_2 =`E`.

Pour les deux autres situations, supposons que le minimum x_1 de t_1 et le minimum x_2 de t_2 vérifient $x_1 \leqslant x_2$ (soit le troisième cas proposé, le quatrième cas se vérifiera de la même façon). Supposons t_1 =`N(`g_1, x_1, d_1`)`, et t le résultat de la fusion proposée.

g_1, le fils gauche de t_1 est un arbre croissant (puisque sous-arbre de t_1). La fusion de l'arbre croissant d_1 avec l'arbre croissant t_2 est un arbre croissant par induction structurelle. Notons t' le résultat de cette fusion.

Le minimum de g_1 est nécessairement supérieur ou égal à x_1, puisque sous-arbre de l'arbre croissant t_1 dont x_1 est le minimum. Les éléments de t' correspondent à l'union

de ceux de t_2 et ceux de d_1. Le minimum de t' est donc supérieur ou égal à x_1 car tous les éléments de t_2 sont supérieurs ou égaux à x_2, lui-même supérieur à x_1, et tous les éléments de d_1 sont supérieurs ou égaux à x_1 puisque d_1 est un sous-arbre de l'arbre croissant t_1 dont x_1 est le minimum. Par conséquent, l'arbre constitué d'un nœud étiqueté par x_1 et dont les fils sont t' et g_1 est bien un arbre croissant.

Pour un x quelconque, le nombre d'occurrences de x dans t est le nombre d'occurrences dans g_1 et dans t', plus 1 si $x = x_1$. Par induction structurelle, le nombre d'occurrences de x dans t' est égal à la somme des nombre d'occurrences de x dans d_1 et dans t_2. Par conséquent, le nombre d'occurrences dans t est égale au nombre d'occurence dans t_2, plus la somme du nombre d'occurrences dans g_1 et dans d_1, plus 1 si $x = x_1$, soit le nombre d'occurrences dans t_1. La fusion conserve donc bien le nombre d'occurrences dans l'arbre d'un entier x quelconque.

6. On commence par définir la fonction `fusion` proposée :

```
let rec fusion t1 t2 = match (t1, t2) with
  | t1, E -> t1
  | E, t2 -> t2
  | N (g1, x1, d1), N (_, x2, _) when x1 <= x2
    -> N (fusion d1 t2, x1, g1)
  | _, N (g2, x2, d2) -> N (fusion d2 t1, x2, g2)
```

Pour ajouter un élément à un arbre croissant, il suffit alors de fusionner l'arbre avec un arbre réduit à une feuille étiquetée par x, soit :

```
let ajoute x = fusion (N (E, x, E))
```

7. L'idée est de supprimer la racine de l'arbre croissant, mais on se retrouve avec deux arbres : les sous-arbres gauche et droit, que l'on fusionne en un seul arbre croissant :

```
let supprime_minimum = function
  | E -> failwith "Arbre vide"
  | N (g, _, d) -> fusion g d
```

8. On implémente la fonction comme demandé, comme une fusion successive d'arbres réduits à une unique feuille :

```
let ajouts_successifs x =
  let t = ref E in
  for i = 0 to Array.length x - 1
    do t := ajoute x.(i) !t done;
  !t
```

On remarquera que l'on fait ici usage de la fonction `ajoute` déjà écrite pour éviter les doublons ! On peut aussi écrire une version plus succinte avec un `Array.fold_left` :

```
let ajouts_successifs =
  Array.fold_left (fun arbre x -> ajoute x arbre) E
```

9. Si l'on prend une suite décroissante d'entiers pour $x_0, \dots, x_{n-1}$, les ajouts successifs des x_i construisent des arbres dans lesquels le x_i se retrouve à la racine (puisque plus petit que x_{i-1}, la racine de l'arbre à l'étape précédente), avec `E` pour fils droit et l'arbre de l'étape précédente pour fils gauche. On construit ainsi un arbre de hauteur n.

10. Pour tout k, notons g_k et d_k les fils respectivement gauche et droits de l'arbre t_k.

Puisque x_0 est le plus petit élément de la suite des éléments insérés dans l'arbre :
- g_k est obtenu par insertion de x_k dans d_{k-1} ;
- $d_k = g_{k-1}$.

Par conséquent, les éléments $x_0, x_1, \dots, x_k$ sont insérés alternativement dans deux arbres : l'un reçoit $x_0, x_2, \dots$, l'autre $x_1, x_3, \dots$. Les deux sous-arbres de t_k diffèrent donc au plus d'un élément (l'arbre de gauche ayant une taille supérieure ou égale à celui de droite).

Mais si l'on s'intéresse à ces sous-arbres, les séquences d'éléments $x_k, x_{k+2}, x_{k+4}, \dots$ sont également croissantes, dont il se passe le même phénomène : les deux sous-arbres de chacun de ces sous-arbres ont une taille qui ne diffère également que d'un seul élément.

Montrons donc par récurrence la propriété (légèrement plus générale que celle demandée) « la hauteur d'un arbre t_n construit en insérant $n > 0$ éléments $x_0, \dots, x_{n-1}$ dans l'ordre croissant est $1 + \lfloor \log_2(n) \rfloor$ » :
- C'est vrai pour $n = 1$ (arbre `N (E, x, E)`, de hauteur 1).
- Supposons $\forall k < n$, $h(t_k) = 1 + \lfloor \log_2(k) \rfloor$.

  Pour t_n, son fils gauche contient $\lceil (n-1)/2 \rceil$ éléments et son fils droit $\lfloor (n-1)/2 \rfloor$.

  Par récurrence, la hauteur de l'arbre t_n est donc $1 + \left(1 + \lfloor \log_2(\lceil (n-1)/2 \rceil) \rfloor\right)$

  Mais $\lceil (n-1)/2 \rceil \leqslant n/2$, donc $h(t_n) \leqslant 2 + \lfloor \log_2(n/2) \rfloor = 1 + \lfloor \log_2 n \rfloor$.

  Seulement, la hauteur d'un arbre binaire à n noeuds internes ne peut être strictement inférieure à $1 + \lfloor \log_2(n) \rfloor$. On a donc $h(t_n) = 1 + \lfloor \log_2(n) \rfloor$, ce qui vérifie la récurrence.

3 Analyse

11. Pour obtenir la complexité demandée, on écrit une fonction auxiliaire `potentiel_et_taille` de signature `arbre -> int * int`, qui retourne le potentiel en même temps que la taille de arbre passé en argument. On ne gardera que le premier de ces deux éléments comme dernier résultat :

```
let potentiel arbre =
  let rec potentiel_et_taille = function
    | E          -> (0, 1)
    | N(g, _, d) -> let pg, tg = potentiel_et_taille g
                    and pd, td = potentiel_et_taille d
                    in (pg+pd+(if tg<td then 1 else 0),
                        tg+td+1)
  in fst (potentiel_et_taille arbre)
```

12. On vérifie le résultat par induction structurelle :

- si t_1 =`E` ou t_2 =`E`, $C(t_1,t_2)=0$, et $\Phi(t_1)+\Phi(t_2)-\Phi(t)=0$, donc l'inégalité est vérifiée.
- sinon, notons t_1 =`N(g1,x1,d1)` et t_2 =`N(g2,x2,d2)` et supposons par exemple que l'on a $x_1 \leqslant x_2$.

  Le résultat de la fusion de t_1 et t_2 est l'arbre t =`N(t',x1,g1)` où t' est le résultat de la fusion des arbres d_1 et t_2. On a donc $C(t_1,t_2)=1+C(d_1,t_2)$.

  Par induction $C(d_1,t_2)\leqslant\Phi(d_1)+\Phi(t_2)-\Phi(t')+2\big(\log(d_1)+\log(t_2)\big)$.

  On a par ailleurs $\Phi(t_1)=\Phi(g_1)+\Phi(d_1)+\mathbb{1}_{|g_1|<|d_1|}$ et $\Phi(t)=\Phi(t')+\Phi(g_1)+\mathbb{1}_{|t'|<|g_1|}$.

  En outre, $|d_1|<|t'|$, donc trois cas sont possibles :

— $|g_1|<|d_1|$: $C(t_1,t_2)\leqslant\Phi(t_1)+\Phi(t_2)-\Phi(t)+2\big(\log|d_1|+\log|t_2|\big)$

  Et donc $C(t_1,t_2)\leqslant\Phi(t_1)+\Phi(t_2)-\Phi(t)+2\big(\log|t_1|+\log|t_2|\big)$

— $|d_1|\leqslant|g_1|\leqslant|t'|$: $C(t_1,t_2)\leqslant\Phi(t_1)+\Phi(t_2)-\Phi(t)+1+2\big(\log|d_1|+\log|t_2|\big)$

  Or $1+2\log|d_1|\leqslant 2\log(2|d_1|)\leqslant 2\log\big(|d_1|+|g_1|\big)\leqslant 2\log|t_1|$

  Donc $C(t_1,t_2)\leqslant\Phi(t_1)+\Phi(t_2)-\Phi(t)+2\big(\log|t_1|+\log|t_2|\big)$

— $|t'|\leqslant|g_1|$: $C(t_1,t_2)\leqslant\Phi(t_1)+\Phi(t_2)-\Phi(t)+2+2\big(\log|d_1|+\log|t_2|\big)$

  Là aussi, $2+2\log|d_1|\leqslant 2\log(2|d_1|)\leqslant 2\log\big(|d_1|+|g_1|\big)\leqslant 2\log|t_1|$

  Donc $C(t_1,t_2)\leqslant\Phi(t_1)+\Phi(t_2)-\Phi(t)+2\big(\log|t_1|+\log|t_2|\big)$

  L'inégalité est donc vérifiée dans chacun des trois cas.

Ce qui prouve l'inégalité demandé par le principe d'induction structurelle.

13. Soit c_k le coût de la construction d'un arbre t_k à k nœuds internes (et donc $2k+1$ nœuds). t_0 =`E`, et pour $k>0$, t_k est obtenu par la fusion de l'arbre t_{k-1} avec `N(E,x,E)` (où `x` correspond à x_{k-1}.

On a donc, pour $k>0$, $c_k\leqslant c_{k-1}+\Phi(t_{k-1})+0-\Phi(t_k)+2\big(\log_2(2(k-1)+1)+\log_2(3)\big)$.

Soit $c_n\leqslant c_0+\Phi(t_0)+2n\log_2(3)+2\sum_{k=1}^{n}\log_2(2k-1)$.

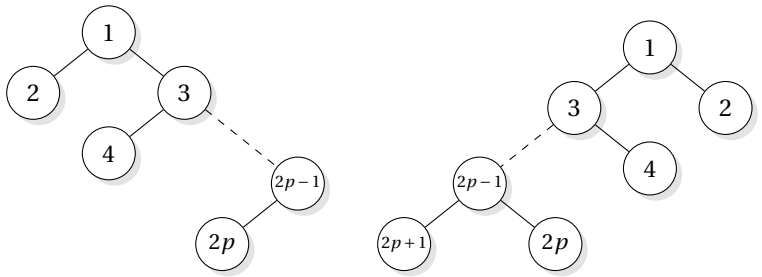
Avec $c_0=0$ et $\Phi(t_0)=0$, on a bien $c_n\leqslant 2\big(n\log_2(3)+\log_2(n!)\big)$.

Soit une complexité en $O\big(n\log(n)\big)$.

14. Il s'agit ici de construire un arbre de hauteur de l'ordre de $n/2$ avec n éléments, puis d'ajouter un élément qui devra être inséré $n/2$ fois. Il s'agit en fait de montrer que la complexité d'une fusion peut être linéaire en n, donc on supposera n pair, construire un exemple pour n impair ne poserait pas de problème particulier.

Posons $p=n/2$ et commençons par construire un arbre croissant de hauteur $p=n/2$. Pour ce faire, on peut remarquer que la construction d'un arbre via la succession d'entiers $2p-1,2p,2p-3,2p-2,\ldots,3,4,1,2$ conduit à l'arbre ci-dessous à gauche (par souci de lisibilité, les feuilles `E` n'ont pas été représentées). En effet, les ajouts des entiers impairs construisent un nouvel étage (ils sont plus petits que les précédents), les pairs sont ajoutés à droite de la racine à la place d'une feuille `E`, avant que les enfants en soient permutés, et donc se retrouvent à gauche, ce qui replace la branche à droite.

L'ajout de $2p+1$ à un tel arbre nécessite de parcourir toute la branche de droite, et donne ultimement l'arbre de droite, après $n/2$ fusions.



La complexité de l'ajout est ici linéaire en p, donc en la taille de l'arbre. Toutefois, l'ajout de n éléments étant en $O\big(n\log(n)\big)$, *en moyenne*, l'ajout d'un élément a bien un coût logarithmique $O\big(\log(n)\big)$.

15. On a $C\big(g_i,d_i\big)\leqslant\Phi\big(g_i\big)+\Phi(d_i)-\Phi(t_{i+1})+2\big(\log_2|g_i|+\log_2|d_i|\big)$.

Mais $\Phi\big(g_i\big)+\Phi(d_i)\leqslant\Phi(t_i)$, $|g_i|\leqslant|t_0|$, $|d_i|\leqslant|t_0|$ et $|t_0|=2n+1$.

Donc $C\big(g_i,d_i\big)\leqslant\Phi(t_i)-\Phi(t_{i+1})+4\log_2(2n+1)$.

La complexité totale c_n vérifie donc $c_n\leqslant\Phi(t_0)-\Phi(t_n)+4n\log_2(2n+1)$.

Soit, avec $\Phi(t_0) \leqslant n$ et $\Phi(t_n) = 0$, c_n est une complexité en $O\big(n\log(n)\big)$.

4 Applications

16. On construit dans un premier temps un arbre croissant avec la fonction `ajouts_successifs`, puis, n fois (autant qu'il y a d'élément dans l'arbre), on retire le plus petit des éléments restant dans l'arbre et on le replace dans le tableau (en utilisant de préférences les fonctions déjà programmées, cela rend les choses plus lisibles) :

```
let tri v =
  let arbre = ref (ajouts_successifs v) in
  for i = 0 to Array.length v - 1 do
    v.(i) <- minimum !arbre;
    arbre := supprime_minimum !arbre
  done;
```

17. Le coût en temps des fusions est directement lié au nombre de comparaisons qui sont effectuées entre les racines des arbres à fusionner. On peut donc dénombrer les comparaisons lors des fusions des différents arbres.

Pour deux arbres de hauteur h et h', le nombre de comparaisons est inférieur à $h + h'$.

On effectue, en posant $n = 2^h$:

- $n/2 = 2^{h-1}$ fusions d'arbres de hauteur inférieure à 1, donc avec au plus 2 comparaison ;
- $n/4 = 2^{h-2}$ fusions d'arbres de hauteur inférieure à 2, donc avec au plus 4 comparaisons ;
- ...
- $n/2^k = 2^{h-k}$ fusions d'arbres de hauteur inférieure à k, donc avec au plus $2k$ comparaisons ;
- ...
- 1 fusion d'arbre de hauteur inférieure à $\log_2(n)$, donc avec au plus $2h$ comparaisons.

On effectue donc moins de

$$\sum_{k=1}^{\log_2(n)} k2^{h-k} = 2^h \sum_{k=1}^{\log_2(n)} \frac{k}{2^k} \quad \text{comparaisons}$$

Pour tout p, $\displaystyle\sum_{k=1}^{p} \frac{k}{2^k} \leqslant 2$, donc on a bien $O\big(2^h\big) = O(n)$ comparaisons.

18. On construit simplement l'arbre t_k^0 grâce à une fonction récursive `foo` de signature `int -> int -> arbre`, qui utilise les relations fournies. Plutôt que de mettre en paramètre j et k, on passe j et $n = 2^k$. Il est ainsi plus simple d'effectuer le premier appel, puisqu'il n'est plus nécessaire de calculer $\log_2(n)$. Et pour $k' = k - 1$, il suffit de prendre $n' = n/2$.

```
let construire t =
  let rec foo j = function
    | 1 -> N (E, t.(j), E)
    | n -> fusion (foo (2*j) (n/2)) (foo (2*j+1) (n/2))
  in foo 0 (Array.length t)
```

19. Il n'y a pratiquement rien à changer si ce n'est mettre un arbre vide plutôt qu'une feuille si on sort du tableau (**c'est beaucoup plus facile que de mettre des valeurs spéciales et d'essayer ensuite de les retirer !**). Attention quand même à la valeur initiale du second paramètre : on prend deux fois la longueur du tableau moins 1 pour correspondre à $k = \lceil \log_2(n) \rceil$:

```
let construire t =
  let n = Array.length t in
  let rec foo j = function
    | 1 -> if j >= n then E else N (E, t.(j), E)
    | n -> fusion (foo (2*j) (n/2)) (foo (2*j+1) (n/2))
  in foo 0 (2*n-1)
```