DS 3 d'option informatique - MP* - Arbres combinatoires (sujet X 2012) - Correction

Question 1 – Soit A l'arbre combinatoire de l'exemple (1) :

$$A = 0 \to (1 \to (2 \to \bot, \top), \top), (1 \to \top, (2 \to, \bot, \top))$$

On a :

$$\begin{aligned} &S(2 \to \bot, \top) = \{\{2\}\}, \\ &S(1 \to (2 \to \bot, \top), \top) = \{\{1\}, \{2\}\}, \\ &S(1 \to \top, (2 \to, \bot, \top)) = \{\varnothing, \{1, 2\}\}. \end{aligned}$$

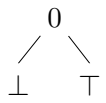
Finalement :

$$\boxed{S(A) = \{\{0\}, \{0, 1, 2\}, \{1\}, \{2\}\}}$$

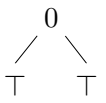
Remarque. Pour déterminer $S(A)$, on a utilisé la définition récursive de l'énoncé. Voici une autre manière de décrire $S(A)$: chaque élément $s \in S(A)$ correspond à un chemin entre la racine de A et une feuille étiquetée par $\top$. L'ensemble s contient toutes les étiquettes rencontrées juste avant de se diriger dans un sous-arbre droit.

Question 2 – Les trois arbres demandés sont :

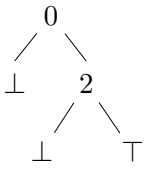
Pour $\{\{0\}\}$:



Pour $\{\varnothing, \{0\}\}$:



Pour $\{\{0, 2\}\}$:



Question 3 – On le montre par récurrence sur $h \in \mathbb{N}$ la hauteur de A :

Initialisation. Si $h = 0$ alors A est réduit à une feuille. Comme $A \neq \bot$, on a $A = \top$, d'où le résultat.

Hérédité. Soit $h > 0$. On suppose que tout arbre de hauteur $h' < h$ contient au moins une feuille $\top$ et on montre la propriété pour un arbre A de hauteur h. Comme $h > 0$, l'arbre A possède un sous-arbre droit A_2 dont la hauteur h_2 vérifie $h_2 < h$. Par la propriété (suppression), on a $A_2 \neq \bot$. En appliquant l'hypothèse de récurrence à A_2, on conclut que A_2 et A contiennent une feuille $\top$.

Question 4 – Pour tout $k \in [\![0; n]\!]$, soit T_k l'ensemble des arbres combinatoires dont les noeuds sont étiquetés par des éléments de $[\![k; n-1]\!]$. Il s'agit de calculer $|T_0|$. On a :

$$\begin{cases} T_n = \{\top, \bot\} \\ T_k = T_{k+1} \uplus \left\{(k \to A_1, A_2) \mid A_1 \in T_{k+1}, A_2 \in T_{k+1} \setminus \{\bot\}\right\} \quad \forall k \in [\![0; n-1]\!] \end{cases}$$

Ainsi, si on note $t_k = |T_k|$, on obtient :

$$\begin{cases} t_n = 2 \\ t_k = t_{k+1} + t_{k+1}(t_{k+1} - 1) = (t_{k+1})^2 \quad \forall k \in [\![0; n-1]\!] \end{cases}$$

Par itération finie sur $k \in [\![0; n]\!]$, on obtient $t_k = 2^{2^{n-k}}$.

$$\boxed{\text{Le nombre d'arbres combinatoires distincts est } t_0 = 2^{2^n}}$$

Remarque. La fonction suivante convient pour la fonction `cons` évoquée dans l'énoncé :

```
let cons i a1 a2 =
  if a2 = Zero then failwith "cons: a2 = Zero";
  Comb (i, a1, a2);;
```

Question 5 – On choisit de renvoyer l'ensemble qui correspond à la feuille en bas à droite de l'arbre (cette feuille est nécessairement $\top$ puisque l'arbre n'est pas vide).

```
let rec un_elt a = match a with
  | Zero -> failwith "Ensemble vide"
  | Un -> []
  | Comb(i, a1, a2) -> i :: un_elt a2;;
```

Lorsqu'on appelle la fonction `un_elt` sur un arbre de hauteur h, il y a 0 ou 1 appel récursif sur un arbre de hauteur strictement inférieure à h. Ainsi, le nombre d'appels récursifs est en $\mathcal{O}(h)$. De plus, mis à part l'appel récursif, la fonction `un_elt` s'exécute en temps constant. Finalement, la complexité est bien en $\mathcal{O}(h)$ avec h la hauteur de A.

Question 6 – Il suffit de créer un arbre où :

→ Les étiquettes sont les éléments de s.

→ Pour tout noeud, le sous-arbre gauche est $\bot$.

→ La feuille en bas à droite est $\top$.

```
let rec singleton li = match li with
  | [] -> Un
  | i :: q -> cons i Zero (singleton q);;
```

La complexité est bien en $\mathcal{O}(n)$ car la taille de la liste diminue de 1 à chaque appel récursif.

Question 7 –

```
let rec appartient e a = match e,a with
  | _, Zero -> false
  | _, Un -> e = []
  | [], Comb(i2, a1, a2) -> appartient [] a1
  | i1 :: q, Comb(i2, a1, a2) ->
     (i1 = i2 && appartient q a2) || (i1 > i2 && appartient e a1);;
```

Pour la complexité, on remarque qu'au plus un appel récursif est effectué (grâce à l'évaluation paresseuse de l'opérateur `&&`). Ainsi, la complexité est bien en $\mathcal{O}(n)$ car la hauteur de l'arbre diminue d'au moins 1 à chaque appel récursif.

Question 8 – Il suffit de compter le nombre de feuilles égales à $\top$.

```
let rec cardinal_Q8 a = match a with
  | Zero -> 0
  | Un -> 1
  | Comb(_, a1, a2) -> cardinal_Q8 a1 + cardinal_Q8 a2;;
```

Question 9 – Soit A l'arbre combinatoire de l'exemple (1) :

$$A = 0 \to (1 \to (2 \to \bot, \top), \top), (1 \to \top, (2 \to, \bot, \top))$$

Alors :

$$\begin{aligned}\mathcal{U}(A) = \Big\{ & A; \\ & (1 \to (2 \to \bot, \top), \top); (2 \to \bot, \top); \bot; \top; \top; \\ & (1 \to \top, (2 \to, \bot, \top)); \top; (2 \to, \bot, \top); \bot; \top \Big\}\end{aligned}$$

Donc $T(A) = |\mathcal{U}(A)| = 6$ (il faut penser à supprimer les doublons). Finalement :

La taille de l'arbre combinatoire de l'exemple (1) est 6

Question 10 – On crée une fonction auxiliaire `aux` dans laquelle on applique le principe de mémorisation. Lors d'un appel à « `aux a` », il y a deux étapes :

→ Si l'arbre `a` n'est pas associé à une valeur dans la table, on calcule le cardinal de `a` avec deux appels récursifs et on ajoute cette valeur dans la table.

→ On renvoie la valeur associée à `a` dans la table (`a` est nécessairement associé à une valeur grâce à l'étape précédente).

```
1  let cardinal_Q10 a0 =
2    let t = cree1() in
3    let rec aux a =
4      if not (present1 t a) then begin
5          match a with
6          | Zero -> ajoute1 t a 0
7          | Un -> ajoute1 t a 1
8          | Comb(_, a1, a2) -> ajoute1 t a (aux a1 + aux a2)
9        end;
10     trouve1 t a
11   in
12   aux a0;;
```

Pour justifier la complexité, on fixe un arbre A représenté par une variable `a` et on compte le nombre d'appels à la fonction `aux` lors de l'exécution de « `cardinal a` ».

On remarque d'abord que chaque élément de $\mathcal{U}(A)$ est ajouté au plus une fois dans la table. De plus, à chaque fois qu'un élément est ajouté dans la table, on effectue deux appels récursifs à la fonction `aux`. Ainsi, le nombre d'appels à la fonction `aux` est au plus $2T(A) + 1$ (le "$2T(A)$" correspond aux appels récursifs ligne 8 et le "+1" correspond à l'appel principal ligne 12). Mis à part les appels récursifs, un appel à `aux` s'exécute en temps constant.

Finalement, la complexité est bien en $\mathcal{O}(T(A))$.

Question 11 – Pour éviter de faire plusieurs fois les mêmes calculs, on utilise une table d'association indexée par des couples d'arbres combinatoires. Cette table associe à un couple (A_1, A_2) l'arbre combinatoire représentant leur intersection.

```
1  let inter a1 a2 =
2    let t2 = cree2() in
3    let rec aux (b: ac) (c: ac): ac =
4      if present2 t2 (b,c) then trouve2 t2 (b,c)
5      else begin
6          let res =
7            match b,c with
8            | Zero, _ | _, Zero -> Zero
9            | Un, Un -> Un
10           | Un, Comb(_, c1, _) -> aux b c1
11           | Comb(_, b1, _), Un -> aux b1 c
12           | Comb(i, b1, b2), Comb(j, c1, c2) when i = j ->
13              let d1 = aux b1 c1 and
14                  d2 = aux b2 c2 in
15              if d2 = Zero then d1 else cons i d1 d2
16           | Comb(i, _, _), Comb(j, c1, _) when i > j -> aux b c1
17           | Comb(_, b1, _), Comb(_) -> aux b1 c
18         in
19         ajoute2 t2 (b,c) res;
20         res
21       end;
22   in
23   aux a1 a2;;
```

Question 12 – Soient A_1 et A_2 deux arbres combinatoires. On exécute la fonction `inter` avec A_1 et A_2 en arguments et on s'intéresse à l'ensemble des appels à la fonction `aux`. Soit E_1 l'ensemble des couples d'arbres combinatoires (B, C) donnés au moins une fois en arguments à `aux`. Soit E_2 les images des couples $(B, C) \in E_1$:

$$E_2 = \Big\{ \texttt{aux}(B, C) \mid (B, C) \in E_1 \Big\}.$$

En étudiant les 7 cas du filtrage de la fonction `aux`, on remarque que :

→ Si deux arbres combinatoires (B, C) sont donnés en arguments de `aux`, et si un appel récursif est effectué sur deux arbres (B', C') (ligne 10, 11, 13, 14, 16 ou 17), alors B' est un sous-arbre de B et C' est un sous-arbre de C. Étant donné que lors du premier appel à la fonction `aux` (ligne 24), les arguments sont les arbres A_1 et A_2, on en déduit que pour tout $(B, C) \in E_1$, l'arbre B est un sous-arbre de A_1 et C est un sous-arbre de A_2. Donc :

$$|E_1| \leqslant |\mathcal{U}(A_1)| \times |\mathcal{U}(A_2)| = T(A_1) \times T(A_2).$$

→ Lorsque la fonction `aux` renvoie un arbre D, il peut être de trois formes :

- `Zero` ou `Un` (ligne 9 ou 10).
- Le résultat d'un appel récursif (ligne 10, 11, 16 ou 17).
- De la forme « `cons i d1 d2` » avec `d1` et `d2` les résultats de deux appels récursifs (ligne 15).

Ainsi, une récurrence sur la hauteur de $D \in E_2$ permet de montrer que tout sous-arbre de D appartient à E_2. En particulier avec $D = \texttt{inter}(A_1, A_2)$:

$$T(\texttt{inter}(A_1, A_2)) = T(\texttt{aux}(A_1, A_2)) = |\mathcal{U}(\texttt{aux}(A_1, A_2))| \leqslant |E_2|.$$

On peut maintenant conclure :

$$T(\texttt{inter}(A_1, A_2)) \leqslant |E_2| \leqslant |E_1| \leqslant |T(A_1)| \times |T(A_2)|.$$

Question 13 – Un domino peut être placé horizontalement ou verticalement. Pour le placer horizontalement, il faut choisir la ligne (p possibilités), puis choisir sa position sur la ligne ($p-1$ possibilités) ; soit un total de $p(p-1)$ positions. De même, il y a $p(p-1)$ façons de le placer verticalement. Finalement :

Il y a $2p(p-1)$ façons différentes de placer un domino

Question 14 – On écrit d'abord une fonction intermédiaire `aucun` qui prend en entrée deux entiers i, j, et renvoie un arbre combinatoire A tel que $S(A) \in \mathcal{P}(\mathcal{P}([\![i; n-1]\!]))$ et pour tout s,

$$s \in S(A) \text{ si et seulement si il n'existe aucun } i' \in s \text{ tel que } m[i'][j] = \texttt{true}.$$

```
(* m est la variable globale définie dans l'énoncé *)
let rec aucun i j =
  if i = Array.length m then Un else
    match m.(i).(j) with
    | true -> aucun (i+1) j
    | false -> let a = aucun (i+1) j in
               cons i a a;;
```

```
(* m est la variable globale définie dans l'énoncé *)
let colonne j =
  let n = Array.length m in
  let rec aux i =
    if i = n then Zero else
      match m.(i).(j) with
      (* 'aucun' ne renvoie jamais Zero *)
      | true -> cons i (aux (i+1)) (aucun (i+1) j)
      | false -> let a = aux (i+1) in
                 if a = Zero then Zero else cons i a a
  in
  aux 0;;
```

Complexité.

→ Fonction `aucun` : à chaque appel, la valeur de `i` augmente de 1, il y a donc $\mathcal{O}(n)$ appels récursifs. La complexité de `aucun` est en $\mathcal{O}(n)$.

→ Fonction `colonne` : le point important est que pour j fixé, il y a au plus quatre entiers i tels que $m[i][j] = \texttt{true}$. Ainsi, la fonction `aucun` est appelée au plus quatre fois. De plus, le nombre d'appels à la fonction `aux` est en $\mathcal{O}(n)$ (mêmes arguments que ci-dessus). Finalement, la complexité est bien en $\mathcal{O}(n)$.

Question 15 –

```
(* Hypothèse: p >= 2, c'est à dire que m est non vide *)
let pavage () =
  let a = ref (colonne 0) in
  for j = 1 to Array.length m.(0)-1 do
    a := inter !a (colonne j);
  done;
  !a;;
```

Complexité. Pour majorer le coût de `pavage`, il faut d'abord majorer le coût de `inter`.

⋆ Fonction `inter`. Comptons le nombre d'appels à la fonction `aux` lors de l'exécution de `inter` appliquée à deux arbres A_1 et A_2. En utilisant le raisonnement de la question 12, on remarque que tous les arguments de la fonction `aux` sont de la forme (B, C) avec $B \in \mathcal{U}(A_1)$ et $C \in \mathcal{U}(A_2)$. Grâce au principe de mémorisation, les lignes 5 à 21 sont exécutées au plus une fois pour chaque $(B, C) \in \mathcal{U}(A_1) \times \mathcal{U}(A_2)$. La fonction `aux` est appelée une fois ligne 23 et effectue au plus 2 appels récursifs entre les lignes 5 et 21. Ainsi, le nombre total d'appels à la fonction `aux` est majoré par $1 + 2 \times T(A_1) \times T(A_2)$.

⋆ Fonction `pavage`. Les p^2 appels à la fonction `colonne` s'exécutent en temps $\mathcal{O}(n \times p^2)$ (voir question 14). Soit A_j l'arbre renvoyé par la fonction `colonne` appliquée à j alors :

$$T(A_j) = \mathcal{O}(n)$$

En effet, soit A un sous-arbre de A_j, alors A est égal à l'un des arbres suivants (voir le code de la fonction `colonne`) :

→ « `aucun i j` » avec `i` $\in [\![$`0; n`$]\!]$.

→ « `aux i` » avec `i` $\in [\![$`0; n`$]\!]$ où `aux` est définie dans la fonction `colonne`.

Le temps d'exécution des $p^2 - 1$ appels à `inter` est donc en $\mathcal{O}(n^2 + n^3 + n^4 + \ldots + n^{p^2})$. Or :

$$\sum_{k=2}^{p^2} n^k = n^{p^2} \frac{1 - \frac{1}{n^{p^2-1}}}{1 - \frac{1}{n}} = \mathcal{O}(n^{p^2})$$

Il reste à exprimer p en fonction de n (je ne sais pas si l'énoncé autorise à exprimer la complexité en fonction de p). Comme $n = 2p(p-1)$ (voir question 13), une résolution d'équation du second degré donne :

$$p = \frac{1 + \sqrt{1 + 2n}}{2} \qquad \text{donc} \qquad p^2 = \frac{n + \sqrt{1 + 2n} + 1}{2}$$

Le coût de `pavage` est en $\mathcal{O}(n^{p^2})$, c'est à dire en $\mathcal{O}(n^{\frac{n+\sqrt{1+2n}+1}{2}})$

Question 16 –

```
let ajoute (t: table) (k: cle) (v: valeur) =
  let h = hache k in
  t.(h) <- (k,v) :: t.(h);;
```

Question 17 –

```
let present (t: table) (k: cle) =
  let rec present_list li = match li with
    | [] -> false
    | (k0,_) :: q -> egal k k0 || present_list q
  in present_list t.(hache k);;
```

Question 18 –

```
let trouve (t: table) (k: cle) =
  let rec trouve_list li = match li with
    | [] -> failwith "Cle absente"
    | (k0,v) :: q when egal k k0 -> v
    | _ :: q -> trouve_list q
  in trouve_list t.(hache k);;
```

Question 19 – Les fonctions `present` et `trouve` s'exécutent en temps linéaire en la taille de la liste « `t.(hache k)` ». Pour espérer avoir des fonctions dont le coût est effectivement en $\mathcal{O}(1)$, les seaux doivent contenir un nombre constant d'éléments. Il faut donc que :

→ Les éléments soient bien répartis entre les différents seaux par la fonction `hache`.

→ Le nombre d'éléments dans la table soit linéaire en H. Il faut donc choisir H suffisamment grand par rapport au nombre d'éléments qui seront ajoutés à la table.

Question 20 – Il s'agit de montrer que pour tous arbres combinatoires A_1, A_2, si $\texttt{egal}(A_1, A_2)$ alors $\texttt{hache}(A_1) = \texttt{hache}(A_2)$. D'après la définition de la fonction `egal`, il y a trois cas où $\texttt{egal}(A_1, A_2)$ est vrai :

→ Si $A_1 = A_2 = \bot$. Dans ce cas, $\texttt{hache}(A_1) = \texttt{hache}(A_2) = 0$.

→ Si $A_1 = A_2 = \top$. Dans ce cas, $\texttt{hache}(A_1) = \texttt{hache}(A_2) = 1$.

→ Si $A_1 = (i_1 \to L_1, R_1)$ et $A_2 = (i_2 \to L_2, R_2)$ avec $i_1 = i_2$ et $\texttt{unique}(L_1) = \texttt{unique}(L_2)$ et $\texttt{unique}(R_1) = \texttt{unique}(R_2)$. Dans ce cas :

$$\begin{aligned}\texttt{hache}(A_1) &= (19^2 \times i_1 + 19 \times \texttt{unique}(L_1) + \texttt{unique}(R_1)) \mod H.\\ &= (19^2 \times i_2 + 19 \times \texttt{unique}(L_2) + \texttt{unique}(R_2)) \mod H.\\ &= \texttt{hache}(A_2).\end{aligned}$$

Remarque concernant la question 21. Pour tester l'égalité entre deux arbres combinatoires, les identifiants de type `unique` présents à la racine ne sont pas pris en compte. En conséquence, les arbres `Comb(u,i,a1,a2)` et `Comb(0,i,a1,a2)` sont considérés égaux par la fonction `egal`.

Ce point est utile dans la fonction `cons` pour tester si l'arbre appartient à la table d'association (c'est à dire s'il a déjà été construit dans le passé). En effet, au début de l'appel à « `cons i a1 a2` », on ne connaît pas l'identifiant « `u: unique` » de l'arbre à construire.

Question 21 –

```
(* On appelle "table_globale" la table de type table1 évoquée dans l'énoncé.
   Elle associe à chaque arbre Comb(_,i,a1,a2) son identifiant (u: unique).
   La variable globale "dernier_unique" stocke le dernier identifiant ayant été
   donné à un arbre.
   Rappel de l'énoncé: "on pose unique(Zero) = 0 et unique(Un) = 1"

   ATTENTION: la fonction 'cons' ne compile pas. Pour cela, il faudrait définir
   table_globale et adapter les fonctions present1, ajoute1 et trouve1 comme le
   suppose l'énoncé *)
```

```
let dernier_unique = ref 1;;

let cons (i: int) (a1: ac) (a2: ac) =
  if a2 = Zero then failwith "cons: a2 = Zero";
  let k = Comb(0, i, a1, a2) in
  if not (present1 table_globale k) then begin
      incr dernier_unique;
      ajoute1 table_globale k !dernier_unique;
    end;
  let u = trouve1 table_globale k in
  Comb(u, i, a1, a2);;
```

Question 22 – Soit n_i le nombre de seaux de longueur i ($n_0 = 6450$, $n_2 = 7340$, ...).

1. Dans le cas où l'arbre doit être construit pour la première fois, la fonction `egal` est utilisée lors de l'appel à la fonction `present` et lors de l'appel à la fonction `trouve`. Pour `present`, le nombre d'appels à la fonction `egal` est la taille du seau `t.(hache k)`. Si on suppose que le seau est choisi de manière uniforme, la taille moyenne d'un seau est :

$$\frac{\sum_{i=0}^{7} i \times n_i}{\sum_{i=0}^{7} n_i} = \frac{22518}{19997} = 1.126\ldots$$

   Pour `trouve`, la fonction `egal` est utilisée une fois (l'arbre est le premier élément de son seau).

Le nombre moyen d'appels à la fonction `egal` est environ 2.13

2. Dans le cas où l'arbre A apparaissait déjà dans la table, on note $p(A)$ la position de A dans son seau : $p(A) = i + 1$ où i est l'indice de A dans la liste qui représente le seau. La fonction `egal` est utilisée $p(A)$ fois dans `present` et $p(A)$ fois dans `trouve`.
   Soit m_p le nombre d'arbre A tels que $p(A) = p$:

$$m_p = \sum_{i=p}^{7} n_i$$

   Si on choisit A de manière uniforme parmi les arbres présents dans la table, alors en moyenne $p(A)$ vaut :

$$M = \frac{\sum_{p=1}^{7} p \times m_p}{22518} = \frac{\sum_{p=1}^{7} \left(p \times \sum_{i=p}^{7} n_i \right)}{22518} = \frac{\sum_{i=1}^{7} \frac{i(i+1)}{2} \times n_i}{22518}$$

   Le nombre d'appels à la fonction `egal` est donc :

$$2M = 3.111\ldots$$

Le nombre moyen d'appels à la fonction `egal` est environ 3.11