École polytechnique - Écoles normales supérieures

Corrigé de l'épreuve d'informatique 2012

MP option informatique

Partie I. Arbres combinatoires

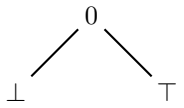
1 On obtient successivement :

$$S(2 \to \bot, \top) = \{\{2\}\},\ S(1 \to (2 \to \bot, \top), \top) = \{\{1\}, \{2\}\},\ S(1 \to \top, (2 \to \bot, \top)) = \{\emptyset, \{1, 2\}\}$$

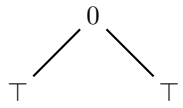
$$S(0 \to (1 \to (2 \to \bot, \top), \top), (1 \to \top, (2 \to \bot, \top))) = \{\{0\}, \{1\}, \{2\}, \{0, 1, 2\}\}$$

2 Les contraintes imposent de choisir respectivement les arbres :

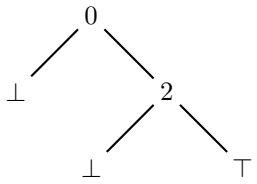
$\{\{0\}\}$



$\{\emptyset, \{0\}\}$



$\{\{0, 2\}\}$



3 Prouvons le résultat par induction sur la hauteur de l'arbre :

- si A est un arbre combinatoire de hauteur nulle, $\mathcal{A}$ est une feuille $\top$.
- soit $h \geq 0$ et supposons que la propriété soit vraie pour tout arbre de hauteur inférieure ou égale à h. Si A est un arbre combinatoire de hauteur $h+1$, il s'écrit $A = (i \to A_1, A_2)$ avec $A_2 \neq \bot$. Comme A_2 est de hauteur au plus h, il contient une feuille $\top$ qui est également une feuille de A.

4 Nous allons démontrer que l'application S est bijective de l'ensemble des arbres combinatoires sur $\mathcal{P}(\mathcal{P}(E))$. Remarquons tout d'abord que si $A = (i \to A_1, A_2)$ est un arbre combinatoire de hauteur non nulle alors $\bigcup_{s \in S(A)} s$ est non vide et son plus petit élément est i (la preuve se fait une nouvelle fois par induction sur la hauteur de A).

Pour X partie de $\mathcal{P}(E)$, nous notons $i(X)$ le minimum de l'ensemble $\bigcup_{s \in X} s$ (par convention, $i(X)$ vaut $n+1$ si cet ensemble est vide, c'est-à-dire si $X = \emptyset$ ou $X = \{\emptyset\}$). Nous allons montrer par récurrence décroissante sur i que X possède un unique antécédent par S, dont les étiquettes sont supérieures ou égales à $i(X)$.

- Si $i(X) = n+1$, $X = \emptyset$ ou $X = \{\emptyset\}$ et X possède pour seul antécédent par S l'arbre $\bot$ ou l'arbre $\top$.

- Supposons que $i(X) < n+1$ et que la propriété ait été démontrée pour tout ensemble Y tel que $i(Y) > i(X)$. Posons alors :

$$Y = \{s \in X,\ i(X) \notin s\} \text{ et } \ Z = \{s \setminus \{i(X)\},\ s \in X \text{ t.q. } i(X) \in s\}$$

Comme $i(Y) > i(X)$ et $i(Z) > i(X)$, il existe un unique couple d'arbres combinatoires (A_1, A_2) tels que $S(A_1) = Y$ et $S(A_2) = Z$.

Si X est représenté par $i \to A'_1, A'_2$, alors $i = i(X)$, $S(A'_1) = Y$ et $S(A'_2) = Z$. Ceci prouve que $A = (i(X), A_1, A_2)$ est le seul antécédent possible pour X : il reste donc à montrer que cet arbre est un arbre combinatoire et qu'il représente X.

Comme Z est non vide (par définition de $i(X)$), $A_2 \neq \perp$. D'autre part, $i(Y) > i(X)$ et $i(Z) > i(X)$ donc A_1 et A_2 ne contiennent pas d'élément j avec $j \leq i(X)$. Ainsi A est un arbre combinatoire et

$$S(A) = S(A_1) \cup \{\{i(X)\} \cup s,\ s \in S(A_2)\} = X.$$

On en déduit qu'il existe 2^{2^n} arbres combinatoires.

Remarque : il est plus simple de calculer ce cardinal par induction sur n, mais la bijectivité de S est une propriété essentielle. En notant a_n le nombre d'arbres combinatoires sur l'ensemble $\{0, 1, \ldots, n-1\}$, nous avons $a_0 = 2$ et $a_{n+1} = a_n(a_n - 1) + a_n = a_n^2$ pour tout $n \geq 0$. Cette formule s'obtient en séparant les arbres combinatoires dont la racine est 0 des arbres combinatoires sur l'ensemble $\{1, 2, \ldots, n\}$.

Partie II. Fonctions élémentaires sur les arbres combinatoires

5-8 On obtient sans problèmes les fonctions :

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

let rec singleton = function
   [] -> Un
   |i::l -> Comb(i,Zero,singleton l);;

let rec appartient s a = match s,a with
   j::l,Comb(i,a1,a2) -> if j<i then
                            false
                         else
                            if i=j then
                               appartient l a2
                            else
                               appartient s a1
  |j::l,_ -> false
  |[],Comb(i,a1,a2) -> appartient [] a1
  |_,_ -> a=Un;;

let rec cardinal = function
   Zero -> 0
   |Un -> 1
   |Comb(_,a1,a2) -> (cardinal a1)+(cardinal a2);;
```

Remarque : la complexité de cette dernière procédure n'est pas de l'ordre du cardinal de $S(A)$ (erreur d'énoncé, comme indiqué dans le rapport de jury).

Partie III. Principe de mémorisation

9 L'arbre A de l'exemple (1) est de taille 6 : il contient les sous-arbres $A_1 = \bot$, $A_2 = \top$, $A_3 = 2 \to \bot, \top$, $A_4 = 1 \to (2 \to \bot, \top), \top$, $A_5 = 1 \to \top, (2 \to \bot, \top)$ et $A_6 = A$.

10 L'analyse est la suivante : on crée une table vide t, et on ajoute à t les entrées $t(\bot) = 0$ et $t(\top) = 1$. Une fonction récursive `card` permet alors de remplir la table tout en renvoyant le cardinal d'un arbre b, selon la méthode suivante :

- si $b = \bot$ ou $\top$;
- si $b = i \to b_1, b_2$, elle renvoie $t(b)$ si cette entrée est déjà définie ; sinon, on applique récursivement `card` à b_1, b_2 : on additionne les résultats obtenus, ce qui donne une valeur c, que l'on associe à b dans la table t et que l'on renvoie.

Il reste ensuite à appliquer la fonction `card` au paramètre de la fonction `cardinal`.

```
let cardinal a =
   let t = cree1() in
   ajoute(t,Zero,0);
   ajoute(t,Un,0);
   let rec card b = match b with
      Zero -> 0
      |Un -> 1
      |Comb(_,a1,a2) -> if present1(t,b) then
                           trouve1(t,b)
                        else
                           begin
                           let c= (card a1)+(card a2) in
                              ajoute1(t,b,c);
                              c
                           end in
   card a;;
```

Quand on applique la fonction `cardinal` à un arbre qui n'est pas une feuille, la fonction `present1` est appliquée $T(A) - 2$ [1] fois avec la réponse `false`, puisque chaque sous-arbre autre que $\bot$ et $\top$ va être ajouté une et une seule fois à t ; dans chacun de ses cas, on effectuera une somme et un appel à la fonction `ajoute1`. Comme chaque appel `present1` avec un paramètre b' qui reçoit la réponse `true` est suivi de l'affectation dans la table t du cardinal du père de b', le nombre total de ces appels ne peut dépasser $2T(A)$, puisque chaque nœud interne possède deux fils. On en déduit que la fonction `cardinal` est de complexité $O(T(A))$.

11 On utilise, comme à la question 10, une table t qui stocke les résultats (on initialise t en associant au couple $(\top, \top)$ l'arbre $\top$) et une fonction récursive auxiliaire `inte`, qui calcule, quand on l'applique à deux arbres combinatoires b_1 et b_2, l'arbre associé à l'intersection des parties représentées par b_1 et b_2. Son analyse est la suivante :

- si l'un des arbres b_i est $\bot$, elle renvoie $\bot$;
- si le résultat cherché est déjà dans la table t, il suffit de le lire et de renvoyer le résultat (on utilise la symétrie de l'intersection) ;
- sinon, on distingue selon les cas :

1. ou $T(A) - 1$ fois si l'arbre ne contient pas de feuille $\bot$.

* si $b_1 = \top$ et $b_2 = (i \to b_{21}, b_{22})$, on applique `inte` à b_1 et à b_{21}, puisque l'ensemble vide appartient à b_2 si et seulement s'il appartient à b_{21} ; le résultat obtenu est alors associé à l'entrée (b_1, b_2) dans la table t, puis renvoyé ;

* si $b_2 = \top$, le traitement est symétrique ;

* si $b_1 = (i \to b_{11}, b_{12})$ et $b_2 = (i \to b_{21}, b_{22})$, on note e_1 et e_2 les resultats des instructions `inte` b_{11} b_{21} et `inte` b_{12} b_{22} : le résultat cherché est alors $(i \to e_1, e_2)$ si $e_2 \neq \bot$ et e_1 sinon : ce résultat est associé à l'entrée (b_1, b_2) dans la table t, puis renvoyé ;

* si $b_1 = (i \to b_{11}, b_{12})$ et $b_2 = (j \to b_{21}, b_{22})$ avec $i \neq j$, le résultat cherché est `inte` b_{11} b_2 si $i < j$, ou `inte` b_1 b_{21} si $i > j$ et on termine le calcul comme dans les cas précédents.

```
let inter a1 a2 =
   let t = cree2() in
   ajoute2(t,Un,Un,Un);
   let rec inte b1 b2 = match b1,b2 with
     Zero,_ -> Zero
     |_,Zero -> Zero
     |_,_ -> if present2(t,b1,b2) then
               trouve2(t,b1,b2)
             else if present2(t,b2,b1) then
               trouve2(t,b2,b1)
             else
               begin
               match b1,b2 with
                 Un,Comb(_,b21,_) -> let c = inte Un b21 in
                                     ajoute2(t,b1,b2,c);
                                     c
               | Comb(_,b11,_),Un -> let c = inte Un b11 in
                                     ajoute2(t,b1,b2,c);
                                     c
               | Comb(i,b11,b12),Comb(j,b21,b22) ->
                    if i=j then
                      begin
                      let c1,c2 = inte b11 b21,inte b12 b22 in
                      if c2 = Zero then
                        begin
                        ajoute2(t,b1,b2,c1);
                        c1
                        end
                      else
                        begin
                        ajoute2(t,b1,b2,Comb(i,c1,c2));
                        Comb(i,c1,c2)
                        end
                    else if i<j then
                      begin
                      let c = inte b11 b2 in
                      ajoute2(t,b1,b2,c);
                      c
                      end
                    else
                      begin
                      let c = inte b1 b21 in
```

```
                    ajoute2(t,b1,b2,c);
                    c
                    end
            end in
inte a1 a2;;
```

Remarque : le temps de calcul n'est pas un $O(T(\texttt{inter}(A_1, A_2))$ (erreur d'énoncé, comme indiqué dans le rapport du jury).

12 Pour tout arbre combinatoire B, notons $SA(B)$ l'ensemble de ses sous-arbres.

Chaque sous-arbre de $\texttt{inter}(A_1, A_2)$ est stocké au moins une fois dans la table t (sauf si ce sous-arbre est égal à $\bot$, pour éviter de mettre en mémoire des entrées inutiles, mais il serait possible d'ajouter l'instruction `ajoute2(t,b1,b2,Zero)` avant de renvoyer `Zero` aux lignes 5 et 6) et cette table est indexée par des couples de sous-arbres de A_1 et A_2. À la fin du calcul, la table t définit une fonction surjective :

$$\begin{array}{rccl} t : & SA(A_1) \times SA(A_2) & \longrightarrow & SA(\texttt{inter}(A_1, A_2)) \\ & (A'_1, A'_2) & \longmapsto & t(A'_1, A'_2) \text{ si } \texttt{trouve}(t, A'_1, A'_2) \end{array}$$

ce qui prouve que $T(\texttt{inter}(A_1, A_2)) \leq T(A_1) \times T(A_2)$.

Partie IV. Application au dénombrement

13 Il y a autant de manière de poser un domino horizontalement que verticalement et, sur chacune des p lignes, il existe $p-1$ façons différentes de poser un domino horizontalement : il existe donc $n = 2p \times (p-1)$ manière de poser un domino sur l'échiquier de dimension $p \times p$.

14 Nous allons utiliser une fonction auxiliaire récursive `constuire: int -> bool -> ac` qui, appliquée à un indice de ligne i_0 et à un booléen b, renvoie un arbre combinatoire codant les parties s contenues dans $\{i_0, i_0+1, \ldots, n-1\}$ telles que $\{i \in s,\ \texttt{m}[i][j] = \texttt{true}\}$ est de cardinal 1 si $b = \texttt{true}$ et de cardinal 0 sinon. La méthode est simple :

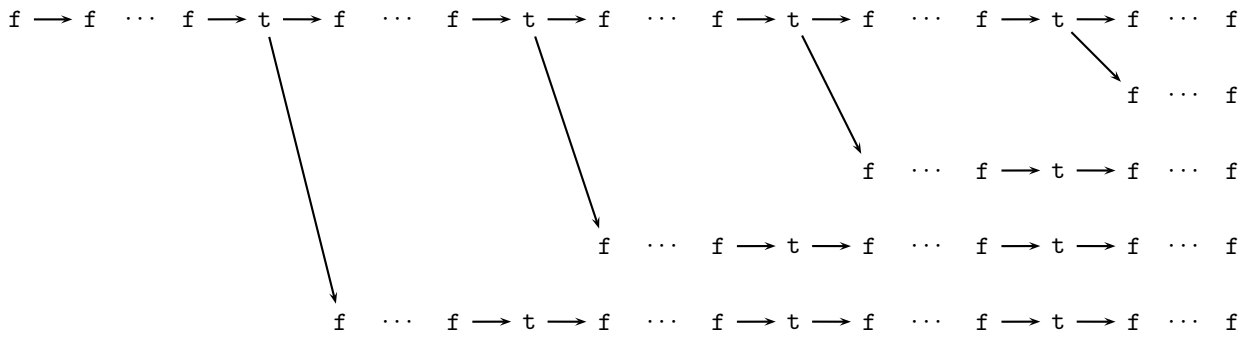
- si $i_0 = n$, on renvoit $\bot$ si $b = \texttt{true}$ et $\top$ sinon ;
- si $i_0 \leq n-1$ et $b = \texttt{false}$, on distingue deux cas :
  - si $\texttt{m}[i_0][j] = \texttt{true}$, on renvoie `construire` (i_0+1) `true` puisqu'il ne faut pas mettre la valeur i_0 dans les parties recherchées ;
  - si $\texttt{m}[i_0][j] = \texttt{false}$, on calcule $B =$ `construire` (i_0+1) `false` et on renvoie $(i_0 \to B, B)$ (en remarquant que B est nécessairement distinct de $\bot$) ;
- si $i_0 \leq n-1$ et $b = \texttt{true}$, on distingue à nouveau deux cas :
  - si $\texttt{m}[i_0][j] = \texttt{true}$, on calcule $B_1 =$ `construire` (i_0+1) `true` et $B_2 =$ `construire` (i_0+1) `false` et on renvoie $(i_0 \to B_1, B_2)$: l'arbre B_1 donnera les parties cherchées ne contenant pas i_0 et l'arbre B_2 donnera, une fois ajouté i_0, les parties cherchées qui contiennent i_0 (une nouvelle fois, B_2 est distinct de $\bot$) ;
  - si $\texttt{m}[i_0][j] = \texttt{false}$, on calcule $B =$ `construire` (i_0+1) `true` : on renvoie $(i_0 \to B, B)$ si $B \neq \bot$ et $\bot$ sinon.

L'instruction `construire` 0 `true` donnera alors le résultat final.

```
let colonne j =
  let rec construire i b =
```

```
 if i=n then
   begin
   match b with
     false -> Un
     |true -> Zero
   end
 else
   begin
   match b,m.(i).(j) with
     false,true -> construire (i+1) false
     |false,false -> let B=construire (i+1) false in Comb(i,B,B)
     |true,true -> Comb(i,(construire (i+1) true),(construire (i+1) false))
     |true,false -> let B=construire (i+1) true in
                         if B=Zero then Zero else Comb(i,B,B)
   end in
construire 0 true;;
```

Le temps de calcul de `colonne j` et la taille de l'arbre renvoyé sont des $O(n)$. En effet, il y a au plus quatre fois la valeur `true` sur une colonne (il y a au plus quatre façons de poser un domino pour recouvrir une case donnée), donc le double appel récursif se fera au maximum quatre fois. Le nombre total d'appels récursifs à la fonction `construire` générés par l'instruction `construire 0 true` est donc majoré par $5n$, valeur qui majore aussi le nombre de sous-arbres de l'arbre renvoyé (un appel à `construire` donne la construction d'au plus un sous-arbre). Le petit schéma qui suit (où la j-ème colonne est représentée en ligne) représente les appels récursifs successifs, dans le cas où la colonne j contient quatre `true` : chaque "virage vers le bas" correspond au passage du booléen de la valeur `true` à la valeur `false`.



15 L'arbre $A_j =$ `colonnej` code toutes les façons de poser des dominos sur l'échiquier de sorte que la case j soit couverte par un et un seul domino : nous obtenons donc l'arbre A cherché en intersectant tous les A_j. Cela donne :

```
let pavage () = let A = ref (colonne 0) in
  for j=1 to (p*p-1) do
    A := inter (!A) (colonne j)
  done;
  !A;;
```

Comme la complexité de l'appel `inter` A_1 A_2 est un $O(\texttt{inter}(A_1, A_2))$, donc un $O(T(A_1) \times T(A_2))$, nous obtenons un temps total $O(n \times p^2)$ pour le calcul des p^2 arbres `colonne j` et un temps de l'ordre de $n^2 + n^3 + \ldots + n^{p^2}$ pour le calcul des différentes intersections. Nous avons donc une complexité pour `pavage` qui est un $O(n^{2p^2}) = O\left((2p(p-1))^{2p^2}\right)$, puisque $n \times p^2 \ll n^2 + n^3 + \ldots + n^{p^2} = \dfrac{n^{p^2+1} - 1}{n-1} - 1 - n \sim n^{p^2+1}$.

Remarque : ce temps de calcul, un peu plus grand que n^n, est très petit devant 2^{2^n}, qui est le cardinal de l'espace de travail.

Partie V. Tables de hachage

16-18 Ces fonctions ne présentent pas de problème particulier :

```
let ajoute t k v = let c=hache k in t.(c)<- (k,v)::t.(c);;

let rec present_liste k = function
   [] -> false
   |(k1,v)::q -> if egal k k1 then true else present_liste k q;;

let present k t = present_liste k t.(hache k);;

let rec trouve_liste k = function
   [] -> failwith "La cl\'e est absente"
   |(k1,v)::q -> if egal k k1 then v else trouve_liste k q;;

let trouve k t = trouve_liste k t.(hache k);;
```

19 La fonction `ajoute` est de complexité constante, mais les fonctions `present` et `trouve` sont de complexité de l'ordre de la longueur maximale des seaux. Il faut donc que la fonction `hache` renvoie des valeurs "bien réparties" dans $[0, K-1]$ et que le rapport entre le nombre total de couples stockés et K reste majoré par une constante. L'idéal est donc d'utiliser une fonction `hache` qui simule une loi uniforme sur $[0, K-1]$, en augmentant la valeur de K dès que le taux de remplissage des seaux est trop grand.

Partie VI. Constructions des arbres combinatoires

20 Soient A_1 et A_2 deux arbres combinatoires tels que $\texttt{egal}(A_1, A_2)$. Si $A_1 = \bot$ ou $\top$, $A_1 = A_2$ et $\texttt{hache}(A_1) = \texttt{hache}(A_2)$. Sinon, on peut écrire $A_1 = (i_1 \to L_1, R_1)$ et $A_2 = (i_2 \to L_2, R_2)$. L'hypothèse s'écrit alors $i_1 = i_2$, $\texttt{unique}(L_1) = \texttt{unique}(L_2)$ et $\texttt{unique}(R_1) = \texttt{unique}(R_2)$, ce qui donne bien $\texttt{hache}(A_1) = \texttt{hache}(A_2)$.

21 Nous définissions les fonctions suivantes, qui permettent de récupérer les différentes valeurs associées à un arbre combinatoire :

```
let unique = function
  Zero -> 0
  | Un -> 1
  | Comb(c,_,_,_) -> c;;

let hache = function
  Zero -> 0
  | Un -> 1
  | Comb(_,i,a1,a2) -> (19*19*i+19*(unique a1)+(unique a2)) mod H;;

let egal a1 a2 = match a1,a2 with
   Comb(_,i1,l1,r1),Comb(_,i2,l2,r2) ->
```

```
                                i1=i2 && (unique l1=unique l2) && (unique r1=unique r2)
   | _,_ -> a1=a2;;
```

Nous supposons qu'une table `t1` de type `table1` est initialisée. Si l'arbre que l'on souhaite construire est déjà dans `t1`, il suffit de le trouver et de le renvoyer ; sinon, on doit créer l'arbre et l'ajouter à `t1`. Le seul problème est dans le choix du champ `unique` : pour que (3) soit vérifié, il faut et il suffit que la valeur choisie soit différente de tous les champs `unique` des arbres déjà construits. Il faut donc utiliser un compteur, associé à la table `t1`. On peut utiliser le type :

```
type table1 = {comp1: mutable int; tab1: ac list vect};;
```

et les fonctions associées :

```
let init1() = let t= make_vect H [] in
   t.(0) <- [Zero];
   t.(1) <- [Un];
   {compt1=2;tab1=t};;

let ajoute1 k t1 = let c=hache k in
   t1.compt1<- t1.compt1+1;
   t1.tab1.(c)<- k::t1.tab1.(c);;

let rec present1_liste k = function
   [] -> false
   |k1::q -> if egal k k1 then true else present1_liste k q;;

let present1 k t1 = present1_liste k t1.tab1.(hache k);;

let rec trouve1_liste k = function
   [] -> failwith "La cl\'e est absente"
   |k1::q -> if egal k k1 then k1 else trouve1_liste k q;;

let trouve1 k t1 = trouve1_liste k t1.tab1.(hache k);;
```

Le code de la fonction `cons` s'écrit alors facilement (on suppose que la table `t1` est une variable globale déjà initialisée). On remarquera que la recherche de l'arbre fonctionne grâce à la propriété (2).

```
let cons i a1 a2 =  let A=Comb(t1.compt1,i,a1,a2) in
   if present1 A t1 then
      trouve1 A t1
   else
      begin
      ajoute1 A t1;
      A
      end;;
```

Nous obtenons par exemple :

```
# let H=19997;;
H : int = 19997
```

```
# let t1 = init1();;
t1 : table1 =
 {compt1 = 2;
  tab1 =
   [|[Zero]; [Un]; []; []; []; []; []; []; []; []; []; []; []; []; [];
     []; []; []; []; []; []; []; []; []; []; []; []; []; []; []; []; [];
     []; []; []; []; []; []; []; []; []; []; []; []; []; []; []; []; [];
     []; []; []; []; []; []; []; []; []; []; []; []; []; []; []; []; [];
     []; []; []; []; []; []; []; []; []; []; []; []; []; []; []; []; [];
     []; []; []; []; []; []; []; []; []; []; []; []; []; []; []; []; [];
     []; []; ...|]}

# cons 0 (cons 1 (cons 2 Zero Un) Un) (cons 1 Un (cons 2 Zero Un));;

- : ac = Comb(5, 0, Comb (4, 1, Comb (2, 2, Zero, Un), Un),
                 Comb (3, 1, Un, Comb (2, 2, Zero, Un)))
```

Notre arbre s'est vu attribué le numéro 5 et sa construction a nécessité celles des sous-arbres $\underbrace{(2 \to \bot, \top)}_{\text{n}^0\,2}$, $\underbrace{(1 \to \top, (2 \to \bot, \top))}_{\text{n}^0\,3}$ et $\underbrace{(1 \to (2 \to \bot, \top), \top)}_{\text{n}^0\,4}$.

22 Pour $i \in \{0, \ldots, 7\}$, notons n_i le nombre de seaux de longueur i.

1. Si l'arbre est construit pour la première fois, on le place dans un seau dont la longueur (avant ajout) est exactement le nombre d'appels à la fonction `egal` réalisés par l'appel à la fonction `cons`. En faisant l'hypothèse que le résultat de la fonction `hache` renvoie une valeur uniformément répartie dans $[0, 19996]$, on obtient le nombre moyen :

$$T_1 = \frac{\sum_{i=0}^{7} i \times n_i}{\sum_{i=0}^{7} n_i} = \frac{22518}{19997} \simeq 1,13$$

On peut remarquer que ce nombre est égal au quotient du nombre d'arbres stockés dans la table par le nombre de seaux.

2. Supposons maintenant que l'arbre est déjà présent. Le nombre d'appels à la fonction `egal` associé à cet arbre est son rang dans le seau qui le contient. En notant $\mathcal{A}$ l'ensemble des arbres déjà construits et $r(A)$ le rang de A dans son seau, nous devons donc calculer :

$$T_2 = \frac{\sum_{A \in \mathcal{A}} r(A)}{\text{Card}(\mathcal{A})} = \frac{\sum_{r=1}^{7} r \times N_r}{\sum_{i=0}^{7} i \times n_i}$$

où N_r est le nombre d'arbres qui se trouvent en r-ème position dans leur seau. Nous avons :

$$\forall\, r \in \{1, \ldots, 7\},\ N_r = \sum_{j=r}^{7} n_j$$

ce qui donne :

$$T_2 = \frac{\sum_{r=1}^{7}\left(r\sum_{j=r}^{7} n_j\right)}{\sum_{i=0}^{7} i \times n_i} = \frac{\sum_{j=1}^{7}\left(\sum_{r=1}^{j} r\right) n_j}{\sum_{i=0}^{7} i \times n_i} = \frac{\sum_{j=1}^{7} \frac{j(j+1)}{2}\, n_j}{\sum_{i=0}^{7} i \times n_i} = \frac{35037}{22518} \simeq 1,56$$