Devoir surveillé 1 c

X-ENS 2012

Option informatique MP1, MP2 & MP3

Arbres combinatoires

On étudie dans ce problème des outils pour la combinatoire, qui peuvent être utilisés en particulier pour répondre à des questions telles que :
combien existe-t-il de façons de paver un échiquier 8×8 *par* 32 *dominos de taille* 2×1 *?*"

Les parties peuvent être traitées indépendamment. Néanmoins, chaque partie utilise des notations et fonctions introduites dans les parties précédentes.

La complexité, ou le coût, d'un programme P (fonction ou procédure) est le nombre d'opération élémentaires (addition, soustraction, multiplication, division, affectation, etc...) nécessaires à l'exécution de P. Lorsque cette complexité dépend d'un paramètre n, on dira que P a une complexité $O(f(n))$, s'il existe $K > 0$ tel que la complexité de P est au plus $Kf(n)$, pour tout n. Lorsqu'il est demandé de garantir une certaine complexité, le candidat devra justifier cette dernière si elle ne se déduit pas directement de la lecture du code.

À retenir

Dans l'ensemble de ce problème, on se fixe une constante entière $n \geq 1$.
On note E l'ensemble $\{0, 1, \ldots, n-1\}$.

I Arbres combinatoires

Dans cette partie, on étudie les arbres combinatoires, une structure de données pour représenter un élément de $\mathcal{P}\big(\mathcal{P}(E)\big)$, c'est à dire un ensemble de parties de E.
Un arbre combinatoire est un arbre binaire dont les nœuds sont étiquetés par des éléments de E et les feuilles par $\bot$ et $\top$.

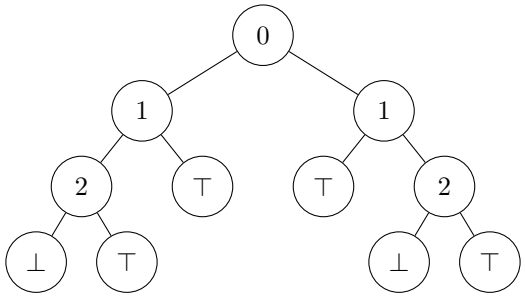


FIGURE 1 – Exemple (1) d'arbre combinatoire

Un nœud étiqueté par i, de sous-arbre gauche g et de sous-arbre droit d sera noté $i \to g, d$. L'arbre ci-dessus peut donc également s'écrire sous la forme

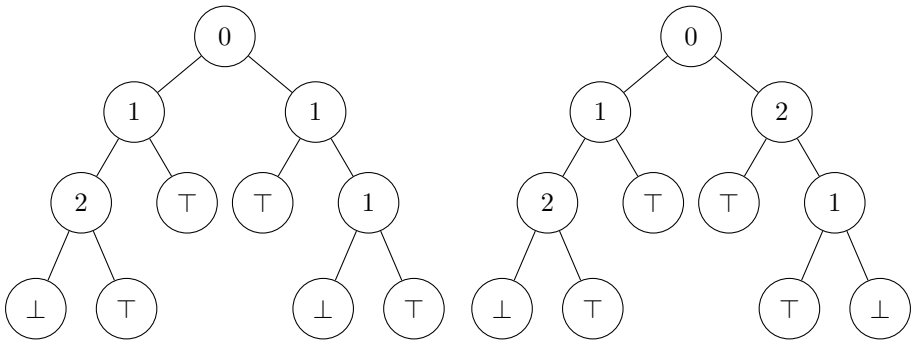
$$0 \to (1 \to (2 \to \bot, \top), \top), (1 \to \top, (2 \to \bot, \top)) \tag{1}$$

Dans ce sujet, on impose les propriétés suivantes sur tout (sous-)arbre combinatoire dont la forme est $i \to g, d$:

Définition 1 : Conditions

g et d ne contiennent pas d'élément j avec $j \leq i$ ..... *(ordre)*
$d \neq \bot$ ..... *(suppression)*

Ainsi, les deux arbres



ne correspondent pas à des arbres combinatoires, car celui de gauche ne vérifie pas la condition *(ordre)* et celui de droite ne vérifie pas la condition *(suppression)*.

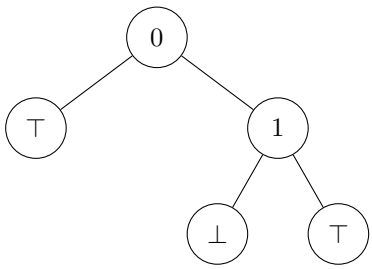
Définition 2 : Détermination de $S(a)$

A tout arbre combinatoire a on associe un ensemble de parties de E, noté $S(a)$, par

$$S(\bot) = \emptyset$$
$$S(\top) = \{\emptyset\}$$
$$S(i \to g, d) = S(g) \cup \big\{\{i\} \cup s \ ; \ s \in S(d)\big\}$$

L'interprétation d'un arbre a de la forme $i \to g, d$ est donc la suivante :

- i est le plus petit élément appartenant à au moins un ensemble de $S(a)$
- $S(g)$ est le sous-ensemble de $S(a)$ des ensembles qui ne contiennent pas i
- $S(d)$ est le sous-ensemble de $S(a)$ des ensembles qui contiennent i auxquels on a enlevé i

Ainsi, l'arbre 

est interprété comme l'ensemble $\{\emptyset, \{0, 1\}\}$.

Question I.1
Donner l'ensemble défini par l'arbre combinatoire de l'exemple (1).

Question I.2
Donner les arbres combinatoires correspondant aux ensembles $\{\{0\}\}$, $\{\emptyset, \{0\}\}$ et $\{\{0, 2\}\}$.

Question I.3
Soit a un arbre combinatoire différent de $\bot$.
Montrer que a contient au moins une feuille $\top$.

Question I.4
Combien existe-t-il d'arbres combinatoires distincts (en fonction de n) ? On justifiera soigneusement la réponse.

II Fonctions élémentaires sur les arbres combinatoires

On se donne le type suivant pour représenter les arbres combinatoires.

```
type ac = Zero | Un | Comb of int * ac * ac;;
```

Le constructeur `Zero` représente $\bot$ et le constructeur `Un` représente $\top$.

Dans les questions suivantes, une partie E est représentée par la liste de ses éléments, triée par ordre croissant.
On définit, provisoirement, le constructeur par

```
let cons i a b = Comb(i, a, b);;
```

Question II.1
Écrire une fonction qui prend en argument un arbre combinatoire a, supposé différent de $\bot$, et qui renvoie *un* ensemble $s \in S(a)$ arbitraire.
On garantira une complexité au plus égale à la hauteur de a.

```
un_elt : ac -> int list
```

Question II.2
Écrire une fonction qui prend en argument un ensemble $s \in \mathcal{P}(E)$ et qui renvoie l'arbre combinatoire représentant le singleton $\{s\}$. On garantira une complexité linéaire.

```
singleton : int list -> ac
```

Question II.3
Écrire une fonction qui prend en argument un ensemble $s \in \mathcal{P}(E)$ et un arbre combinatoire a et qui teste si s appartient à $S(a)$. On garantira une complexité linéaire.

```
appartient : int list -> ac -> bool
```

Question II.4
Écrire une fonction qui prend en argument un arbre combinatoire a et qui renvoie $\left|S(a)\right|$, le cardinal de $S(a)$. On évaluera sa complexité.

```
cardinal : ac -> int
```

III Principe de mémorisation

On définit l'ensemble des sous-arbres d'un arbre combinatoire a, noté $\mathcal{U}(a)$, par

$$\begin{array}{rcl} \mathcal{U}(\bot) & = & \{\bot\} \\ \mathcal{U}(\top) & = & \{\top\} \\ \mathcal{U}(i \to g, d) & = & \{i \to g, d\} \cup \mathcal{U}(g) \cup \mathcal{U}(d) \end{array}$$

La taille d'un arbre combinatoire a, notée $T(a)$, est définie comme le cardinal de $\mathcal{U}(a)$, c'est à dire comme le nombre de ses sous-arbres *distincts*.

Question III.1
Quelle est la taille de l'arbre combinatoire de l'exemple (1) ?

Pour écrire efficacement une fonction sur les arbres combinatoires, on va mémoriser tous les résultats obtenus par cette fonction, de manière à ne pas refaire deux fois le même calcul. Pour cela, on suppose donnée une structure de table d'association indexée par des arbres combinatoires. Plus précisément, on suppose donné un type `table1` représentant une table associant à des arbres combinatoires des valeurs d'un type quelconque et les quatre fonctions suivantes :

- `cree1()` renvoie une nouvelle table, initialement vide
- `ajoute1 t a v` ajoute l'association de la valeur v à l'arbre a dans la table t
- `present1 t a` renvoie un booléen indiquant si l'arbre a est associé à une valeur dans la table t
- `trouve1 t a` renvoie la valeur associée à l'arbre a dans la table t, en supposant qu'elle existe

On suppose que les trois fonctions `ajoute1`, `present1` et `trouve1` ont toutes un coût constant.
On suppose de même l'existence d'un type `table2` représentant des tables d'association indexées par des couples d'arbres combinatoires et quatre fonctions similaires `cree2`, `ajoute2`, `present2` et `trouve2` également de coût constant.
Les parties V et VI expliqueront comment de telles tables peuvent être construites.

Question III.2
Réécrire la fonction `cardinal` de la question 8 à l'aide du principe de mémorisation pour garantir une complexité linéaire en la taille de l'arbre.

Question III.3
Écrire une fonction qui prend en argument deux arbres combinatoires a_1 et a_2 et qui renvoie l'arbre combinatoire représentant leur intersection,
c'est à dire l'arbre a tel que $S(a) = S(a_1) \cap S(a_2)$.

```
inter : ac -> ac -> ac
```

Question III.4
Montrer que, pour tous arbres combinatoires a_1 et a_2, on a

$$T(\texttt{inter}\ a_1\ a_2) \leq T(a_1) \times T(a_2)$$

IV Application au dénombrement

On en vient maintenant au problème de dénombrement évoqué dans l'introduction.
Soit p un entier pair supérieur ou égal à 2. On cherche à déterminer le nombre de façons de paver un échiquier de dimension $p \times p$ avec $\frac{p^2}{2}$ dominos de taille 2×1. Pour cela, on va construire un arbre combinatoire a tel que le cardinal de $S(a)$ est exactement le nombre de pavages possibles.

Question IV.1
Combien existe-t-il de façons différentes de placer *un* domino 2×1 sur l'échiquier ?

Dans ce qui suit, on suppose que n est égal à la réponse à la question précédente, et que chaque élément $i \in E$ représente un placement possible de domino. Chaque case de l'échiquier est représentée par un entier j tel que $0 \leq j < p^2$, les cases étant numérotées de gauche à droite, puis de haut en bas. On se donne une matrice de booléens `m` de taille $n \times p^2$. Le booléen `m.(i).(j)` vaut `true` si et seulement si la ligne i correspond à un placement de domino qui occupe la case j. (On suppose avoir rempli ainsi la matrice `m`, qui est une variable globale.)

Un élément s de $\mathcal{P}(E)$ représente un ensemble de lignes de la matrice m. Il correspond à un pavage si et seulement si chaque case de l'échiquier est occupée par exactement un domino, *i.e.* si et seulement si pour toute colonne j, il existe une unique ligne $i \in s$ telle que `m.(i).(j)= true`.
On parle alors de **couverture exacte** de la matrice `m`.

Question IV.2
Écrire une fonction qui prend en argument un entier j avec $0 \leq j < p^2$, et qui renvoie un arbre combinatoire a tel que, pour tout s, $s \in S(a)$ si et seulement si il existe un unique $i \in s$ tel que `m.(i).(j)= true`.
On garantira une complexité linéaire.

```
colonne : int -> ac
```

Question IV.3
En déduire une fonction qui renvoie un arbre combinatoire a tel que le cardinal de $S(a)$ est égal au nombre de façons de paver l'échiquier.

```
pavage : unit -> ac
```

Majorer le coût de `pavage` en fonction de n.

V Tables de hachage

Dans cette partie, on explique comment réaliser les structures de données `table1` et `table2`, qui ont notamment permis d'obtenir des fonctions `inter` et `cardinal` efficaces. L'idée consiste à utiliser des *tables de hachage*.

On abstrait le problème en considérant qu'on cherche à construire une structure de table d'association pour des clés d'un type `cle` et des valeurs d'un type `valeur`, ces deux types étant supposés déjà définis. On se donne un entier $H > 1$ et on suppose l'existence d'une fonction `hache` de coût constant, des clés vers les entiers, telle que pour toute clé k, $0 \leq$ `hache k` $< H$.
L'idée consiste alors à utiliser un tableau de taille H et à stocker dans la case i les entrées correspondant à des clés k pour lesquelles `hache k` vaut i. Chaque case du tableau est appelée un *seau*. Comme plusieurs clés peuvent avoir la même valeur par la fonction `hache`, un seau est une liste d'entrées, c'est à dire une liste de couples *(clé,valeur)*.
Un table sera donc de type `(cle * valeur)list vect` ; pour alléger l'écriture, on simplifiera ce type par `table` dans les signatures.
On suppose par ailleurs qu'on peut comparer deux clés à l'aide d'une fonction `egal` à valeurs dans les booléens, également de coût constant, qui vérifie la propriété

Définition 3 : Propriété d'égalité

Ppour toutes clés k_1 et k_2,
`egal k1 k2` vaut `true` si et seulement si `hache k1 = hache k2`.

Question V.1
Écrire une fonction qui prend en argument une table de hachage t, une clé k et une valeur v, et ajoute l'entrée (k, v) à la table t. On ne cherchera pas à tester si l'entrée (k, v) existe déjà dans t et on garantira une complexité constante.

```
ajoute : table -> cle -> valeur -> unit
```

Question V.2
Écrire une fonction qui prend en argument une table de hachage t et une clé k, et qui teste si la table t contient une entrée pour la clé k.

```
present : table -> cle -> bool
```

Question V.3
Écrire une fonction qui prend en argument une table de hachage t et une clé k, et qui renvoie la valeur associée à la clé k dans, en supposant qu'elle existe.

```
trouve : table -> cle -> valeur
```

Question V.4
Sous quelles hypothèses sur la valeur de H et la fonction `hache` peut-on espérer que le coût des fonctions `ajoute`, `present` et `trouve` soit effectivement $O(1)$?

VI Construction des arbres combinatoires

Il reste enfin à expliquer comment réaliser une fonction de hachage, une fonction d'égalité et une fonction `cons` sur les arbres combinatoires, qui soient toutes les trois de complexité $O(1)$.

L'idée consiste à associer un entier unique à chaque arbre combinatoire a, noté `unique a`, et à garantir la propriété

Définition 4 : Propriété d'unicité

Pour tous arbres combinatoires, a_1 et a_2 :
$a_1 = a_2$ si et seulement si `unique a1 = unique a2`.

Pour cela, on pose `unique Zero = 0` et `unique Un = 1`.
Pour un arbre a de la forme $i \to g, d$, on choisira pour `unique a` une valeur arbitraire supérieure ou égale à 2, stockée dans le nœud de l'arbre. On modifie donc ainsi la définition du type `ac` :

```
type ac = Zero | Un | Comb of int * int * ac * ac;;
```

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

On propose alors la fonction `hache` suivante sur les arbres combinatoires :

- `hache Zero` vaut 0
- `hache Un` vaut 0
- Si $a = i \to g, d)$, `hache a` vaut $19^2 \times i + 19 \times$ `unique g` + `unique d` mod H

Le choix de cette fonction, et du coefficient 19 en particulier, relèvent de considérations pratiques uniquement.

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

De même, on propose la fonction `egal` suivante sur les arbres combinatoires :

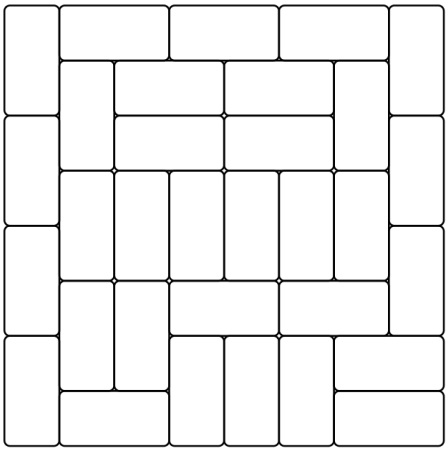
- `egal Zero Zero` vaut `true`
- `egal Un Un` vaut `true`
- Si $a_1 = i_1 \to g_1, d_1)$, $a_2 = i_2 \to g_2, d_2)$ alors egal a1 a2 prend la valeur $i_1 = i_2$ et `unique g1 = unique g2` et `unique d1 = unique d2`
- `egal a b` vaut `false` dans les autres cas.

```
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;;
```

Question VI.1
Montrer que les fonctions `hache` et `egal` vérifient bien la propriété d'égalité page 6.

Question VI.2
Proposer un code pour la fonction de construction `cons` qui garantisse la propriété d'unicité (page 6), en supposant que les arbres combinatoires sont exclusivement construits à partir de `Zero`, `Un` et de la fonction `cons`.
On garantira un coût $O(1)$ en utilisant une table globale de type `table1` contenant les arbres combinatoires déjà construits.
On suppose que le type `table1` et ses opérations ont été adaptés au nouveau type `ac`.

On veut résoudre le problème de pavage pour l'exemple ci-dessous.



On construit au total 22518 arbres combinatoires. Si on prend $H = 19997$ et la fonction de hachage proposée ci-dessus, la longueur des seaux dans la table utilisée pour `cons` n'excède jamais 7. Plus précisément, les arbres se répartissent dans cette table de la manière suivante :

longueur du seau	0	1	2	3	4	5	6	7
nombre de seaux de cette longueur	6450	7340	7080	1617	400	96	11	3

Question VI.3
Quel est, dans cet état, le nombre moyen d'appels à la fonction `egal` réalisés par un nouvel appel à la fonction `cons`

- dans le cas où l'arbre doit être construit pour la première fois ;
- dans le cas où l'arbre apparaissait déjà dans la table ?

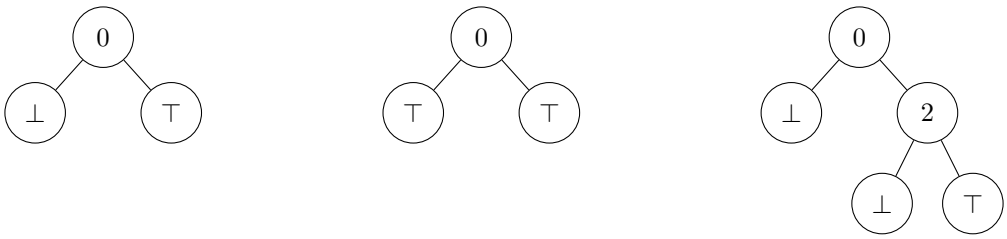
À retenir

La solution au problème du pavage est obtenue en quelques secondes avec la technique proposée ici ; on trouve 12988816. L'intérêt de cette technique est qu'elle s'applique facilement à d'autres problèmes de combinatoire. Par ailleurs, le problème de la couverture exacte peut être attaqué par d'autres techniques, telles que les "liens dansants" de Knuth.

Solutions

Solution de la question I.1
L'arbre combinatoire de l'exemple (1) définit $\big\{\{0\},\{1\},\{2\},\{0,1,2\}\big\}$.

Solution de la question I.2
$\{\{0\}\}$ est représenté par $0 \to \bot, \top$
$\{\emptyset,\{0\}\}$ est représenté par $0 \to \top, \top$
$\{\{0,2\}\}$ est représenté par $0 \to \bot, (2 \to \bot, \top)$



Solution de la question I.3
On montre par récurrence sur la hauteur h d'un arbre combinatoire que, si $h \geq 1$, le fils droit contient au moins une feuille $\top$.

- Si $h = 1$ le fils droit est une feuille qui ne peut être $\bot$ en raison de la règle de suppression, c'est donc $\top$: le fils droit contient une feuille $\top$.
- On suppose que le fils droit des arbres combinatoires de hauteur $h-1$ au plus contient au moins une feuille $\top$. Soit $a = i \to a_1, a_2$ de hauteur h ;
  - — si a_1 est une feuille alors, comme ci-dessus, c'est $\top$
  - — si a_1 est un arbre combinatoire de hauteur h' avec $1 \leq h' \leq h-1$ alors, d'après l'hypothèse de récurrence, son fils droit contient au moins une feuille $\top$ donc a_1 contient au moins une feuille $\top$

  Dans tous les cas a_1 contient une feuille $\top$.

Solution de la question I.4
On note C_k le nombre d'arbres combinatoires dont les éléments appartiennent à un ensemble de cardinal k.
On a $C_0 = 2$ et, en regroupant les arbres dont la racine est étiquetée par i dans le cas $E = \{0,1,\dots,n-1\}$, $C_n = 2 + \sum_{i=0}^{n-1} C_{n-1-i}(C_{n-1-i}-1)$. En effet si $a = i \to a_1, a_2$ alors les étiquettes de a_1 et de a_2 sont dans l'ensemble $\{i+1,\dots,n-1\}$ et a_1 n'est pas $\bot$.
On a donc $C_n = 2 + \sum_{i=0}^{n-1} C_i(C_i-1)$ d'où $a_1 = 4$ et, pour $n \geq 2$,
$C_n = 2 + \sum_{i=0}^{n-2} C_i(C_i-1) + C_{n-1}(C_{n-1}-1) = C_{n-1} + C_{n-1}(C_{n-1}-1) = C_{n-1}^2$ donc $a_n = 2^{2^n}$.

On pouvait aussi prouver que S définit une bijection de l'ensemble des arbres combinatoires sur E vers $\mathcal{P}\big(\mathcal{P}(E)\big)$ en déterminant son inverse.
On a facilement $S^{-1}(\emptyset) = \bot$ et $S^{-1}(\{\emptyset\}) = \top$.
Si $F \subset \mathcal{P}(E)$ contient un élément non vide, on choisit le plus petit entier appartenant à un des ensemble de F et on définit $F_1 = F \cap \mathcal{P}\big(\{i+1,\dots,n-1\}\big)$ et
$F_2 = \big\{X \cap \{i+1,\dots,n-1\}\ ;\ X \in F,\ i \in X\big\}$.
On définit alors $S^{-1}(F) = i \to S^{-1}(F_1), S^{-1}(F_2)$ par récurrence descendante sur i.
Il faut alors prouver que S^{-1} est bien définie et est l'inverse de S, ce qui n'est pas si facile.

Solution de la question II.1
Pour trouver un élément de $S(a)$, c'est-à-dire une partie de E on va prendre les éléments des racines et les ajouter aux racines des fils droits.
On est ainsi certain de parvenir à un dernier fils droit égal à $\top$.

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

L'ordre croissant de la liste est respecté car la racine a une étiquette inférieure à celles des nœuds de ses fils.
La complexité est la hauteur de la descendance par la droite, elle est majorée par la hauteur de l'arbre.

Solution de la question II.2
On fabrique l'arbre en descendant, c'est la réciproque de la question précédente.

```
let rec singleton = function
 |[] -> Un
 |a::l -> cons  a Zero (singleton l);;
```

On n'introduit pas de feuille $\bot$ à droite donc la condition de suppression est respectée.
La liste est croissante donc les étiquettes croissent vers le bas, la condition d'ordre est respectée.
La complexité est le nombre d'éléments de la liste donc majorée par n.

Solution de la question II.3
L'ensemble vide est recherché dans les fils gauches.
Pour un ensemble non vide le plus petit élément doit être supérieur ou égal à l'étiquette de la racine.
En cas d'égalité on recherche le reste à droite, si le plus petit élément est strictement supérieur, on cherche l'ensemble à gauche.

```
let rec appartient l a =
   match a, l with
   |Zero, l -> false
   |Un, [] -> true
   |Un, _ -> false
   |Comb(_, g, _), [] -> appartient [] g
   |Comb(i, g, d), k::ll ->
       if i > k then false
                else if i = k then appartient ll d
                              else appartient l g;;
```

Chaque appel descend la hauteur de l'arbre passé en paramètre d'au moins 1, la complexité est majorée par la hauteur de l'arbre donc par n.

Solution de la question II.4
Si $a = i \to g, d$ alors $S(a) = S(g) \cup \big\{\{i\} \cup s \ ; \ s \in S(d)\big\}$ donne $\big|S(a)\big| = \big|S(g)\big| + \big|S(d)\big|$ car les deux ensembles sont disjoints et l'ajout de i définit une bijection.

```
let rec cardinal arbre =
   match arbre with
   |Zero -> 0
   |Un -> 1
   |Comb(i, f, d) -> cardinal g + cardinal d;;
```

La complexité est majorée par le nombre de nœuds de l'arbre, il y en a au plus 2^n.

Solution de la question III.1
L'arbre de l'exemple (1) est de taille 6 car il contient les 6 sous-arbres distincts
$\bot \quad \top \quad 2 \to \bot, \top \quad 1 \to (2 \to \bot, \top), \top \quad 1 \to \top, (2 \to \bot, \top) \quad a$

Solution de la question III.2
On remplit un tableau avec les valeurs calculées de manière récursive et il suffit de lire la valeur de a.

```
let cardinal a =
   let t = cree1() in
   let rec card a =
      if not(present1 t a)
      then begin match a with
           |Zero -> ajoute1 t Zero 0
           |Un -> ajoute1 t Un 1
           |Comb(i, g, d) -> ajoute1 t a (card g + card d) end;
      trouve1 t a in
   card a;;
```

La fonction auxiliaire `card` n'est appelée que pour les arbres non encore calculés c'est-à-dire $T(A)1$ fois : sa complexité est en $\mathcal{O}\big(T(A)\big)$.

Solution de la question III.3
L'arbre représentant l'intersection de a_1 et a_2 se calcule selon les valeurs des arbres.

- Si `a1` ou `a2` vaut `Zero` alors $S(a_1) \cap S(a_2) = \emptyset$ et `inter a1 a2` renvoie `Zero`
- Si `a1` vaut `Un` alors $S(a_1) = \{\emptyset\}$ et l'intersection est égale à $\{\emptyset\}$ ou $\emptyset$ selon que $\{\emptyset\}$ appartient ou non à $S(a_2)$ ce qui se voit dans le fils gauche de `a2`.
  De même si `a2` vaut `Un`
- Si $a_1 = i \to g_1, d_1$ et $a_2 = j \to g_2, d_2$ avec $i < j$ alors les éléments associés à $S(d_1)$ contiennent i alors qu'aucun élément de $S(a_2)$ ne contient i d'où `inter a1 a2` est la même chose que `inter g1 a2`
  De même si on a $i > j$.
- $a_1 = i \to g_1, d_1$ et $a_2 = j \to g_2, d_2$ avec $i = j$ alors les éléments de $S(a_1) \cap S(a_2)$ qui ne contiennent pas i sont ceux de $S(g_1) \cap S(d_1)$ et ceux qui contiennent i sont associés aux éléments de $S(d_1) \cap S(d-2)$. Le résultat de `inter a1 a2` pourrait être `Comb (i, inter g1 g2, inter d1 d2)`
  Il faut cependant exclure le cas où `inter d1 d2` en raison de la condition de suppression.

De plus, bien que cela ne soit pas indiqué, il semble indispensable de ne pas calculer plusieurs fois les intersections déjà calculées. On utilisera une méthode semblable à la précédente.

```
let inter a1 b1 =
   let t = cree2() in
   let rec intersect a b =
      if present2 t (a,b)
      then trouve2 t (a,b)
      else match (a,b) with
           |Zero, _ -> ajoute2 t (a, b) Zero;
                       Zero
           |_, Zero -> ajoute2 t (a, b) Zero;
                       Zero
           |Un, Un -> ajoute2 t (Un, Un) Un;
                      Un
           |Un, Comb(_, bg, _) -> let ab = intersect Un bg in
                                  ajoute2 t (Un, b) ab;
                                  ab
           |Comb(_, ag, _), Un -> let ab = intersect Un ag in
                                  ajoute2 t (a, Un) ab;
                                  ab
           |Comb(i, ag, ad), Comb(j, bg, bd) when  i < j
               -> let ab = intersect ag b in
                  ajoute2 t (a, b) ab;
                  ab
           |Comb(i, ag, ad), Comb(j, bg, bd) when  i > j
               -> let ab = intersect a bg in
                  ajoute2 t (a, b) ab;
                  ab
           |Comb(i, ag, ad), Comb(j, bg, bd)
               -> let g = intersect ag bg
                  and d = intersect ad bd in
                  let ab = if d = Zero
                           then g
                           else cons i g d in
                           ajoute2 t (a, b) ab;
                           ab
   in intersect a b;;
```

La complexité est le nombre d'appel qui est égal au nombre de sous-arbres, $T(\texttt{inter}(a_1, a_2))$.

Solution de la question III.4
Dans la question ci-dessus $T(\texttt{inter}(a_1, a_2))$ est le nombre de cases définies dans le tableau `t`. Or chacun des appels qui définit une case se fait à partir d'un couple de sous-arbres. Ainsi le nombre d'appel est majoré par le nombre de sous-arbres de a_1 multiplié par le nombre de sous-arbres de a_2, ce qui majore ne nombre de sous arbre de `inter a1 a2`.

Solution de la question IV.1
Pour placer un domino horizontalement il suffit de placer la partie gauche dans une ligne quelconque et une colonne distincte de la dernière, il y a ainsi $p(p-1)$ positions.
Il y en a, par symétrie autant pour le placer verticalement.
Il existe donc $2p(p-1)$ façons différentes de placer un domino 2×1 sur un échiquier $p \times p$.

Solution de la question IV.2
On construit l'arbre combinatoire en s'intéressant aux éléments de E (les placements de dominos) par ordre décroissants. Pour chaque i on maintient deux arbres :

- l'arbre a_0 des parties de $\{i, i+1, \ldots, n-1\}$ qui ne contiennent aucun indice k tel que `m.(k).(j)` vaut `true`, ce sont les ensembles auxquels on peut ajouter un élément rencontrant j

- l'arbre a_1 des parties de $\{i, i+1, \ldots, n-1\}$ qui ne contiennent un indice unique k tel que `m.(k).(j)` vaut `true`, ce sont les ensembles qui appartiennent à l'arbre recherché.

Le résultat final est l'arbre a_1 lorsqu'on a intégré 0.
Si on suppose construits ces deux arbres pour $i+1$, on a deux cas pour passer à l'indice suivant.

- Si `m.(i).(j)` vaut `true`, a_0 est inchangé, mais on peut ajouter dans a_1 les parties de a_0 en ajoutant i donc a_1 est remplacé par $i \to a_1, a_0$.
- Si `m.(i).(j)` vaut `false`, on peut ajouter dans a_0 les parties de a_0 augmentées de i donc a_0 est remplacé par $i \to a_0, a_0$. De plus on peut aussi ajouter i dans les parties de a_1 donc a_1 est remplacé par $i \to a_1, a_1$, à condition que a_1 ne soit pas vide (`Zero`).

On commence par $a_0 = \top$, l'ensemble vide ne contient pas de placement rencontrant j, par contre $a_1 = \bot$.

```
let colonne j =
   let rec col i a0 a1 =
      if i < 0
      then a1
      else if m.(i).(j)
           then col (i-1) a0 (cons i a1 a0)
           else if a = Zero
                then col (i-1) (cons i a0 a0)
                else col (i-1) (cons i a0 a0) (cons i a1 a1)
   in col (n-1) Un Zero;;
```

L'appel récursif, qui n'ajoute que des calculs à temps constant, se fait pour chaque entier de 0 à $n-1$. La complexité est donc linéaire en n.

Solution de la question IV.3
Les pavages correspondent à un ensemble des positions de dominos I tel que, pour toute case j du damier il existe une unique position $i \in I$ telle que `m[i][j_0]=true`. On doit donc avoir I dans `colonne j` pour tout j. On calcule donc l'intersection des `colonne j`.

```
let pavage () =
   let dim = Array.length m.(0);
   let rec pave k a =
      if k < 0
      then a
      else inter a (colonne k) in
    pave (dim-2) (colonne (dim-1));;
```

Il faut initialiser par un ensemble non vide d'où la dernière ligne.
La valeur de `dim` est p^2 ; on fait donc p^2 fois appel à `colonne` ce qui donne une complexité de $np^2 = \%_{00}p^4$.
On fait p^2 fois appel à `inter` ; si on note a_k les différentes intersections calculées la complexité d'un calcul est de l'ordre de $T(a_{k-1} = T(\texttt{inter ak (colonne k)}) \leq T(a_k) \times T(\texttt{colonne k})$.
On a ainsi $T(a_{k-1}) \leq 2nT(a_k)$ puis $T(a_k) \leq (2n)^{p^2-k}$ donc la complexité des appels à `inter` est majorée par $\mathcal{O}\big((2n)^{p^2}\big) = \mathcal{O}\big((4p^2)^{p^2}\big) = \mathcal{O}\big((2p)^{2p^2}\big)$
Ce dernier terme est un infiniment grand par rapport au premier :
la complexité totale est en $\mathcal{O}\big((2p)^{2p^2}\big)$.

Solution de la question V.1

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

Solution de la question V.2
On cherche k dans les premiers éléments des paires que constitue la liste `hache k`.

```
let present t k =
 let rec pres = function
  |[] -> false
  |(k1,v)::ll -> if egal k k1
                 then true
                 else pres ll in
 pres t.(hache k);;
```

Solution de la question V.3
C'est le même principe

```
let trouve t k =
 let rec haha = function
  |[] -> failwith "Clé non présente"
  |(k1,v)::ll -> if egal k k1
                 then v
                 else haha  ll in
 haha t.(hache k);;
```

Solution de la question V.4
`ajoute` est à temps constant car `hache` l'est.
`present` et `trouve` explorent une liste, les autres opérations étant à temps constant. Pour que la complexité soit constante il faut donc que les seaux aient une taille limitée par un entier K. Pour cela il faut que KH soit supérieur au nombre d'objets à classer et que la fonction de hachage répartisse régulièrement les objets dans les seaux.

Solution de la question VI.1
`hache` est calculée, dans le cas d'un arbre distinct de `Un` et `Zero`, par les valeur de i, a_1 et a_2 donc le cas d'égalité par `egal` implique l'égalité des clé de hachage.
On ne teste pas l'égalité par la valeur de `unique` de l'arbre car dans la suite on calcule celle-ci après avoir calculé la valeur de `hache`.

Solution de la question VI.2
L'idée est de donner une valeur provisoire à `unique` puis de voir si l'arbre a déjà été construit (d'où l'intérêt de ne pas utiliser cette valeur) et de changer la valeur de `unique` si elle existait déjà.
On commence par les initialisations :

```
let t_unique = Array.make h [];;
ajoute t_unique Zero 0;;
ajoute t_unique Un 1;;
let compteur = ref 2;;
```

On travaille ensuite avec ces objets globaux :

```
let cons i a b =
 if present t_unique Comb(0,i,a,b)
  then let uni = trouve t_unique Comb(0,i,a,b) in
         comb(uni,i,a,b)
  else (let uni = !compteur in
         incr compteur;
         ajoute t-unique Comb(uni,i,a,b) uni;
         Comb(uni,i,a,b)
       );;
```

Solution de la question VI.3

- Si l'arbre est construit pour la première fois on n'utilise `egal` que dans l'instruction `present` et on doit parcourir la liste associée en entier parce qu'on ne le trouve pas. Dans la situation où est le tableau la longueur moyenne d'une liste est

$$\frac{0.6450 + 1.7340 + 2.4080 + 3.1617 + 4.400 + 5.96 + 6.11 + 7.3}{6450 + 7340 + 4080 + 1617 + 400 + 96 + 11 + 3} \sim 1,13$$

- Si l'arbre apparaissait déjà dans la table on le trouve dans la liste et la fonction `egal` est appelée deux fois, dans `present` et dans `trouve`.

  On note n_k le nombre de seaux de longueur k, il y a au total $\sum_{k=1}^{7} k.n_k$ arbres déjà trouvés.

  Le nombre de recherches pour les arbres dont l'image appartient à un seau de longueur k est $\sum_{i=1}^{k} i.n_k = \frac{k(k+1)}{2}$ car il y a n_k arbre qu'on trouve en 1 recherche, autant en deux recherches etc

  Ainsi la moyenne du nombre d'appels à égal est

$$2\frac{\frac{2.1}{2}.7340 + \frac{3.2}{2}.4080 + \frac{4.3}{2}.1617 + \frac{5.4}{2}.400 + \frac{6.5}{2}.96 + \frac{7.6}{2}.11 + \frac{8.7}{2}3}{1.7340 + 2.4080 + 3.1617 + 4.400 + 5.96 + 6.11 + 7.3} \sim 3,11$$