# Arbres combinatoires (X-ENS 2012)

Durée : 4 heures

---

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* $8 \times 8$ *par* 32 *dominos de taille* $2 \times 1$ *?*

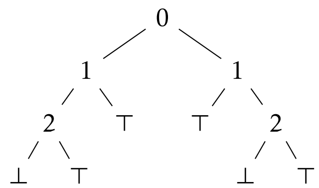
La partie I introduit la structure d'arbre combinatoire, qui permet de représenter un ensemble d'ensembles d'entiers. La partie II étudie quelques fonctions élémentaires sur cette structure. La partie III propose ensuite un principe de mémorisation, pour définir des fonctions plus complexes sur les arbres combinatoires. La partie IV utilise les résultats précédents pour répondre au problème de dénombrement ci-dessus. Enfin, les deux dernières parties expliquent comment construire et manipuler efficacement des arbres combinatoires, à l'aide tables de hachage.

Les parties peuvent être traitées indépendamment. Néanmoins, chaque partie utilise des notations et fonctions introduites dans les parties précédentes. Les tableaux sont indexés à partir de 0 et la notation $t[i]$ est utilisée dans les questions pour désigner l'élément d'indice $i$ du tableau $t$.

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

## Partie I. Arbres combinatoires

Dans cette partie, on étudie les arbres combinatoires, une structure de données pour représenter un élément de $\mathscr{P}(\mathscr{P}(\mathrm{E}))$, c'est à dire un ensemble de parties de E. Un arbre combinatoire est un arbre binaire dont les noeuds sont étiquetés par des éléments de E et les feuilles par $\perp$ et $\top$. Voici un exemple d'arbre combinatoire :



Un noeud étiqueté par $i$, de sous-arbre gauche $\mathrm{A}_1$ et de sous-arbre droit $\mathrm{A}_2$ sera noté $i \to \mathrm{A}_1, \mathrm{A}_2$. L'arbre ci-dessus peut donc également s'écrire sous la forme

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

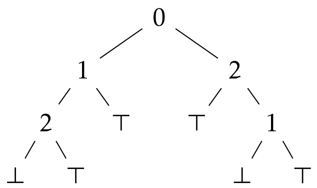
Dans ce sujet, on impose la propriété suivante sur tout (sous-)arbre combinatoire dont la forme est $i \to \mathrm{A}_1, \mathrm{A}_2$ : d'une part

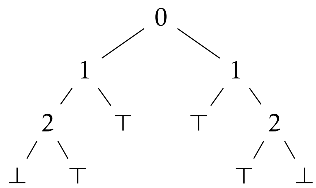
$$\mathrm{A}_1 \textit{ et } \mathrm{A}_2 \textit{ ne contiennent pas d'élément } j \textit{ avec } j \leqslant i \qquad \text{(ordre)}$$

et d'autre part

$$\mathrm{A}_2 \neq \perp \qquad \text{(suppression)}$$

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).

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

$$\begin{aligned} \mathrm{S}(\bot) &= \emptyset \\ \mathrm{S}(\top) &= \{\emptyset\} \\ \mathrm{S}(i \to \mathrm{A}_1, \mathrm{A}_2) &= \mathrm{S}(\mathrm{A}_1) \cup \Big\{ \{i\} \cup s \;\Big|\; s \in \mathrm{S}(\mathrm{A}_2) \Big\} \end{aligned}$$

L'interprétation d'un arbre A de la forme $i \to \mathrm{A}_1, \mathrm{A}_2$ est donc la suivante : $i$ est le plus petit élément appartenant à au moins un ensemble de S(A), $\mathrm{A}_1$ est le sous-ensemble de S(A) des ensembles qui ne contiennent pas $i$ et $\mathrm{A}_2$ est le sous-ensemble de S(A) des ensembles qui contiennent $i$ auxquels on a enlevé $i$. Ainsi, l'arbre

```
     0
    / \
   ⊤   1
      / \
     ⊥   ⊤
```

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

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

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

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

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

## Partie II. Fonctions élémentaires sur les arbres combinatoires

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

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

Le constructeur **Zero** represente $\bot$ et le constructeur **Un** représente $\top$. On se donne une fonction **cons** pour construire un noeud de la forme $i \to \mathrm{A}_1, \mathrm{A}_2$.

```
cons : int -> ac -> ac -> ac
```

Cette fonction suppose que les propriétés (ordre) et (suppression) sont vérifiées. On suppose que cette fonction a un coût O(1).

Dans les questions suivantes, une partie E est représentée par la liste de ses éléments, triée par ordre croissant. On note *ensemble* le type correspondant, c'est à dire

```
type ensemble == int list ;;
```

**Question 5.** Écrire une fonction **un_elt** qui prend en argument un arbre combinatoire A, supposé différent de $\bot$, et qui renvoie *un* ensemble $s \in \mathrm{S(A)}$ arbitraire. On garantira une complexité au plus égale à la hauteur de A.

```
un_elt : ac -> ensemble
```

**Question 6.** Écrire une fonction **singleton** qui prend en argument un ensemble $s \in \mathscr{P}(\mathrm{E})$ et qui renvoie l'arbre combinatoire représentant le singleton $\{s\}$. On garantira une complexité $\mathrm{O}(n)$.

```
singleton : ensemble -> ac
```

**Question 7.** Écrire une fonction **appartient** qui prend en argument un ensemble $s \in \mathscr{P}(\mathrm{E})$ et un arbre combinatoire A et qui teste si $s$ appartient à S(A). On garantira une complexité $\mathrm{O}(n)$.

```
appartient : ensemble -> ac -> bool
```

**Question 8.** Écrire une fonction **`cardinal`** qui prend en argument un arbre combinatoire A et qui renvoie card(S(A)), le cardinal de S(A).

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

## Partie III. Principe de mémorisation

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

$$\begin{aligned}\mathcal{U}(\bot) &= \{\bot\}\\ \mathcal{U}(\top) &= \{\top\}\\ \mathcal{U}(i \to \mathrm{A}_1, \mathrm{A}_2) &= \{i \to \mathrm{A}_1, \mathrm{A}_2\} \cup \mathcal{U}(\mathrm{A}_1) \cup \mathcal{U}(\mathrm{A}_2)\end{aligned}$$

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

**Question 9.** Quelle est la taille de l'arbre combinatoire de l'exemple (1) ?

**Principe de mémorisation.** 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 O(1). 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 10.** Réécrire la fonction **`cardinal`** de la question 8 à l'aide du principe de mémorisation pour garantir une complexité O(T(A)). Justifier soigneusement la complexité du code proposé.

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

**Question 11.** Écrire une fonction **`inter`** qui prend en argument deux arbres combinatoires $\mathrm{A}_1$ et $\mathrm{A}_2$ et qui renvoie l'arbre combinatoire représentant leur intersection, c'est à dire l'arbre A tel que $\mathrm{S}(\mathrm{A}) = \mathrm{S}(\mathrm{A}_1) \cap \mathrm{S}(\mathrm{A}_2)$.

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

**Question 12.** Montrer que, pour tous arbres combinatoires $\mathrm{A}_1$ et $\mathrm{A}_2$, on a

$$\mathrm{T}(\mathtt{inter}(\mathrm{A}_1, \mathrm{A}_2)) \leqslant \mathrm{T}(\mathrm{A}_1) \times \mathrm{T}(\mathrm{A}_2)$$

## Partie 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 $2 \times 1$. On donne un exemple d'un tel pavage pour $p = 8$ sur la figure 1.
Pour cela, on va construire un arbre combinatoire A tel que le cardinal de S(A) est exactement le nombre de pavages possibles.

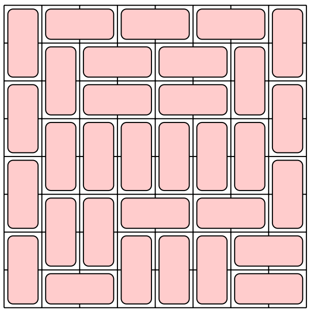


Figure 1 – Un exemple de pavage pour $p = 8$

**Question 13.** Combien existe-t-il de façons différentes de placer *un* domino $2 \times 1$ sur l'échiquier ?

Dans ce qui suit, on suppose que $n$ est égal à la réponse à la question précédente, et que chaque élément $i \in \mathrm{E}$ représente un placement possible de domino. Chaque case de l'échiquier est représentée par un entier $j$ tel que $0 \leqslant j < p^2$, les cases étant numérotées de gauche à droite, puis de haut en bas. On se donne une matrice de booléens **m**$m$ 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 $\mathscr{P}(\mathrm{E})$ 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 14.** Écrire une fonction **colonne** qui prend en argument un entier $j$ avec $0 \leqslant j < p^2$, et qui renvoie un arbre combinatoire A tel que, pour tout $s$,

$$s \in \mathrm{S}(\mathrm{A}) \text{ si et seulement si il existe un unique } i \in s \text{ tel que } m[i][j] = \texttt{true}$$

On garantira une complexité $\mathrm{O}(n)$.

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

**Question 15.** En déduire une fonction **pavage** 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
```

# Partie 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 *clé* et des valeurs d'un type *valeur*, ces deux types étant supposés déjà définis. On se donne un entier $\mathrm{H} > 1$ et on suppose l'existence d'une fonction **hache** de coût constant, des clés vers les entiers, telle que pour toute clé $k$

$$0 \leqslant \texttt{hache}(k) < \mathrm{H}$$

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 $\texttt{hache}(k) = i$. 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)*. On adopte donc le type suivant :

```
type table == (clé * valeur) list vect ;;
```

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, telle que pour toutes clés $k_1$ et $k_2$,

$$\text{si } \texttt{egal}(k_1, k_2) \text{ alors } \texttt{hache}(k_1) = \texttt{hache}(k_2) \tag{2}$$

**Question 16.** Écrire une fonction **ajoute** 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 $O(1)$.

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

**Question 17.** Écrire une fonction **present** 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 -> clé -> bool
```

**Question 18.** Écrire une fonction **trouve** 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 -> clé -> valeur
```

**Question 19.** 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 $O(1)$ ?

# Partie 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 suivante pour tous arbres combinatoires $A_1$ et $A_2$ :

$$A_1 = A_2 \text{ si et seulement si } \mathtt{unique}(A_1) = \mathtt{unique}(A_2) \tag{3}$$

Pour cela, on pose **unique**(**Zero**) = 0 et **unique**(**Un**) = 1. Pour un arbre A de la forme $i \to A_1, A_2$, on choisira pour **unique**(A) une valeur arbitraire supérieure ou égale à 2, stockée dans le noeud de l'arbre. On modifie donc ainsi la définition du type *ac* :

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

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

$$\begin{aligned}
\mathtt{hache}(\bot) &= 0 \\
\mathtt{hache}(\top) &= 1 \\
\mathtt{hache}(i \to A_1, A_2) &= 19^2 \times i + 19 \times \mathtt{unique}(A_1) + \mathtt{unique}(A_2) \bmod H
\end{aligned}$$

(Le choix de cette fonction, et du coefficient 19 en particulier, relève de considérations pratiques uniquement.) De même, on propose la fonction **egal** suivante sur les arbres combinatoires :

$$\begin{aligned}
\mathtt{egal}(\bot, \bot) &= \mathtt{true} \\
\mathtt{egal}(\top, \top) &= \mathtt{true} \\
\mathtt{egal}((i_1 \to L_1, R_1), (i_2 \to L_2, R_2)) &= i_1 = i_2 \text{ et } \mathtt{unique}(L_1) = \mathtt{unique}(L_2) \\
&\quad \text{et } \mathtt{unique}(R_1) = \mathtt{unique}(R_2) \\
\mathtt{egal}(A_1, A_2) &= \mathtt{false} \text{ sinon}
\end{aligned}$$

**Question 20.** Montrer que les fonctions **hache** et **egal** ci-dessus vérifient bien la propriété (2).

**Question 21.** Proposer un code pour la fonction **cons** qui garantisse la propriété (3), 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**.)

```
cons : int -> ac -> ac -> ac
```

Pour résoudre le problème de pavage de la partie IV, on construit au total 22 518 arbres combinatoires. Si on prend $H = 19\,997$ 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 22.** Quel est, dans cet état, le nombre moyen d'appels à la fonction **`egal`** réalisés par un nouvel appel à la fonction **`cons`**

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

On donnera les valeurs avec deux décimales, en les justifiant soigneusement.

> ***Note*** *: la solution au problème du pavage est obtenue en quelques secondes avec la technique proposée ici ; on trouve* 12 988 816. *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.

♦ ♦

♦