Arbres combinatoires
(Adapté d'un sujet d'informatique X 2012)

Introduction

On étudie dans ce problème des outils pour la combinatoire, qui peuvent être utilisés par exemple pour répondre à des questions telles que « combien existe-t-il de façons de paver un échiquier de taille $n \times n$ (n pair) par $n^2/2$ dominos de taille 2×1 »?

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 utilise cette structure our répondre au problème de dénombrement ci-dessus. La partie IV développe des outils pour une structure de dictionnaire dont les clés sont des arbres combinatoires. La partie V utilise ces dictionnaires pour améliorer les performances de l'algorithme de dénombrement grâce à la mémoïsation.

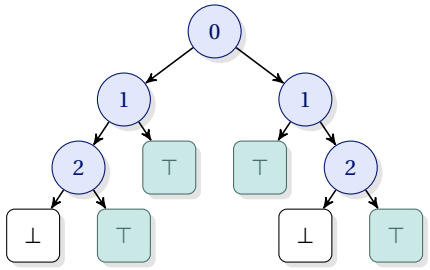
Les parties peuvent être traitées de façon largement indépendantes. Néanmoins, chaque partie utilise des notations et des fonctions introduites dans les parties précédentes. Le langage utilisé est le langage OCaml. **On pensera à commenter les fonctions proposées, et justifier leur complexités.**

1 Arbres combinatoires

Dans tout ce devoir, on note $\mathrm{E}_p = [\![0\,..\,p-1]\!]$ l'ensemble des entiers de 0 à p. Une partie p de E_p est un ensemble non ordonné d'éléments (distincts) de E_p. Par exemple, l'ensemble vide $\emptyset$, l'ensemble $\{2\}$, l'ensemble $\{1,3\}$ ou l'ensemble $\{0,1,2,3,4\}$ soit quatre exemples de parties de E_5. On note $\mathcal{P}(\mathrm{E}_p)$ l'ensemble des parties de E_p.

De même, on peut considérer des parties de $\mathcal{P}(\mathrm{E}_p)$, en d'autres termes des éléments de $\mathcal{P}(\mathcal{P}(\mathrm{E}_p))$. Par exemple, l'ensemble $\{\emptyset,\{2\},\{1,3\},\{0,1,2,3,4\}\}$ est une partie des parties de E_5.

Dans cette première partie, on introduit les *arbres combinatoires*, une structure de données permettant de représenter un élément de $\mathcal{P}(\mathcal{P}(\mathrm{E}_p))$ où p est un entier fixé. Un arbre combinatoire est un arbre binaire strict où les nœuds ont étiquetés par des entiers de $\mathrm{E}_p = [\![0\,..\,p-1]\!]$ et les feuilles par $\bot$ ou $\top$. Voici un exemple d'arbre combinatoire :



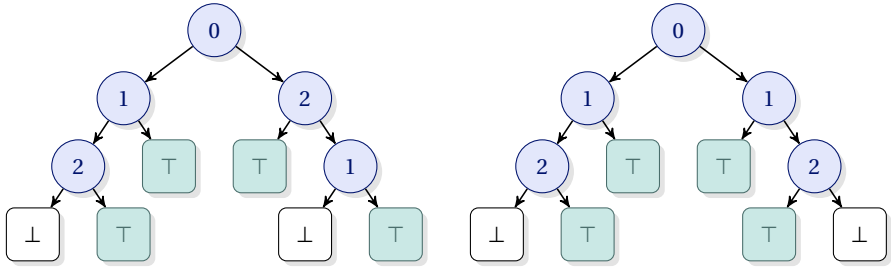
Un nœud interne étiqueté par i, de sous-arbres gauche A_1 et de sous-arbre droit A_2 sera noté $i \to \mathrm{A}_1, \mathrm{A}_2$. L'arbre ci-dessus peut donc également s'écrire sous la forme

$$0 \to (1 \to (2 \to \bot, \top), \top), (1 \to \top, (2 \to \bot, \top)).$$

Dans le sujet, on impose la double propriété suivante pour tout (sous-)arbre combinatoire de la forme $i \to \mathrm{A}_1, \mathrm{A}_2$:

- (ordre) A_1 et A_2 ne contiennent pas de nœud interne étiqueté j avec $j \leqslant i$;
- (suppression) $\mathrm{A}_2 \neq \bot$.

Ainsi, les deux arbres

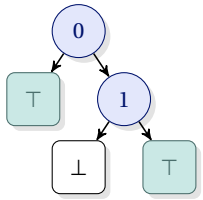


ne correspondent pas à des arbres combinatoires, car celui de gauche ne vérifie pas la condition (ordre) et celui de droite ne vérifie pas la condition (suppression). Tous les arbres combinatoires fournis à des fonctions sont supposés vérifier ces deux conditions. Il tient à vous que tous les arbres renvoyés par vos fonctions les vérifient également.

À tout arbre combinatoire A, on associe un ensemble de parties de E_p, noté $\mathrm{S}(\mathrm{A})$, défini par

$$\begin{cases} \mathrm{S}(\bot) = \emptyset \\ \mathrm{S}(\top) = \{\emptyset\} \\ \mathrm{S}(i \to \mathrm{A}_1, \mathrm{A}_2) = \mathrm{S}(\mathrm{A}_1) \cup \{\{i\} \cup s \mid s \in \mathrm{S}(\mathrm{A}_2)\} \end{cases}$$

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 $\mathrm{S}(\mathrm{A})$, $\mathrm{S}(\mathrm{A}_1)$ est le sous-ensemble de $\mathrm{S}(\mathrm{A})$ des ensembles qui ne contiennent pas i, et $\mathrm{S}(\mathrm{A}_2)$ le sous-ensemble de $\mathrm{S}(\mathrm{A})$ des ensembles qui contiennent i auxquels on a enlevé i. Ainsi, l'arbre ci-dessous est interprété comme l'ensemble d'ensembles $\{\emptyset,\{0,1\}\}$.



1. Donner l'ensemble d'ensembles défini par l'arbre combinatoire de l'exemple (1).

2. Donner les arbres combinatoires correpondant à chacun des trois ensembles suivants : $\{\{0\}\}$, $\{\emptyset,\{1\}\}$, et $\{\{2,3\}\}$.

3. Soit A un arbre combinatoire distinct de $\bot$. Montrer que A contient au moins une feuille étiquetée $\top$.

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

2 Fonctions sur les arbres combinatoires

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

```
type ct =
  | T                       (* feuille ⊤ *)
  | F                       (* feuille ⊥ *)
  | Comb of int * ct * ct   (* arbre i → A1,A2 *)
```

Dans cette définition, le constructeur `T` représente $\top$, le constructeur `F` représente $\bot$.

Dans les questions suivantes, une partie de $[\![0\,..\,p-1]\!]$ sera représentée par la liste de ses éléments, **triée par ordre croissant**. On note `set` le type correspondant, c'est-à-dire

```
type set = int list
```

5. Écrire une fonction `one_set` de signature `ct -> set` qui prend en argument un arbre combinatoire A différent de $\bot$ et renvoie *un* ensemble $s \in S(A)$ arbitraire. On garantira une complexité en $O(h(A))$.

6. Écrire une fonction `singleton` de signature `set -> ct` qui prend en argument un ensemble $s \in \mathcal{P}(E_p)$ et renvoie l'arbre combinatoire représentant le singleton $\{s\}$. On garantira une complexité $O(|s|)$.

7. Écrire une fonction `member` de signature `set -> ct -> bool` prenant en argument un ensemble $s \in \mathcal{P}(E_p)$ et un arbre combinatoire A et qui teste si s appartient à $S(A)$. On garantira une complexité $O(p)$.

8. Écrire une fonction `equals` de signature `ct -> ct -> bool` qui prend en argument deux arbres combinatoires A_1 et A_2 et renvoie un booléen indiquant si les ensembles $S(A_1) = S(A_2)$. On justifiera sa correction. Quelle est sa complexité ?

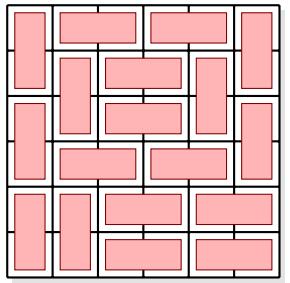
9. Écrire une fonction `cardinal` de signature `ct -> int` qui prend en argument un arbre combinatoire A et renvoie $|S(A)|$, le cardinal de $S(A)$.

10. Écrire une fonction `intersect` de signature `ct -> ct -> ct` qui prend en argument deux arbres combinatoires A_1 et A_2 et renvoie un arbre combinatoire A tel que $S(A) = S(A_1) \cap S(A_2)$ (l'arbre résultat A doit ainsi représenter l'intersection des parties de parties de E_p représentés par A_1 et A_2).

3 Application au dénombrement

On en vient au problème de dénombrement évoqué dans l'introduction. Soit n un entier pair strictement positif. On chrche à déterminer le nombre de façons de paver un échiquer de dimensions $n \times n$ avec $n^2/2$ dominos de taille 2×1.

Voici un exemple d'un tel pavage pour $n = 6$:



Pour ce faire, nous allons construire un arbre combinatoire A tel que $|S(A)|$ corresponde exactement au nombre de pavages possibles.

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

Dans ce qui suit, on note p la réponse à la question précédente, et on suppose que chaque élément $i \in [\![0\,..\,p-1]\!]$ représente un placement possible de domino sur l'échiquier.

Chaque case de l'échiquier et représentée par un entier j tel que $0 \leqslant j < n^2$, les cases étant numérotées de gauche à droite, puis de haut en bas (la case en haut à droite est, par exemple, numérotée $n-1$).

On suppose disposer d'une fonction `gen_mat` de signature `int -> bool array array` qui prend en argument la dimension n (supposée paire) d'un échiquier et retourne une matrice `m` de taille $p \times n^2$ (en temps $O(p \times n^2)$) telle que $m_{i,j}$ vaut `true` si et seulement si le placement i du domino couvre la case j.

Un élément $s \in \mathcal{P}(E_p)$ peut être associé à un ensemble de lignes de la matrice m. Il correspond à un pavage si et seulement si chaque case de l'échiquier et 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 $m_{i,j}$ contient `true`. On parle alors de *couverture exacte* de la matrice m.

12. Proposer une fonction `column` de signature `bool array array -> int -> ct` qui prend en argument une matrice m et un entier j, avec $0 \leqslant j < n^2$, et 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 $m_{i,j}$ contient `true`. On garantira une complexité $O(p)$, et on prendra soin de détailler l'algorithme proposé (note : cette question est difficile).

13. En déduire une fonction `pavage` de signature `int -> ct` qui prend en argument la dimension n d'un échiquier et renvoie un arbre combinatoire A tel que le cardinal de $S(A)$ est égal au nombre de façons de paver l'échiquier.

On peut alors théoriquement obtenir le nombre de façons de recouvrir un échiquier de taille $n \times n$ en écrivant « `cardinal (pavage n)` ». Malheureusement, la fonction est présentement coûteuse

en temps et en espace, car elle doit effectuer de très nombreux calculs, notamment d'intersections d'ensembles.

Cependant, il y a un espoir : beaucoup de ces calculs sont répétés un grand nombre de fois. Or si la première fois que l'on fait appel à `inter a1 a2` on est bien obligé d'effectuer le calcul, si ultérieurement on a besoin de la *même* intersection, si l'on a pris la peine de mémoriser le résultat, on peut s'épargner de refaire le calcul ! C'est le principe de la *mémoïzation*.

4 Dictionnaires

Pour mémoriser les résultats de certains appels, nous allons utiliser des dictionnaires, dont les clés (et les valeurs) seront des arbres combinatoires. Dans cette partie, nous allons implémenter notre propre structure de dictionnaire.

Pour construire un tel dictionnaire, nous aurons besoin d'une fonction permettant de tester l'égalité deux clés, donc de deux arbres combinatoires. Deux arbres combinatoires sont considérés égaux s'ils représentent le même élément de $\mathcal{P}(\mathcal{P}(\mathrm{E}_p))$. Malheureusement, déterminer si deux arbres sont égaux est une opération dont le coût n'est pas négligeable, comme cela a été établi pour la fonction `equals` dans la partie II.

Pour pouvoir déterminer si deux arbres combinatoires sont égaux *en temps constant*, on va associer à chaque arbre combinatoire A que l'on manipule un entier naturel, noté $\texttt{uniq}(\mathrm{A})$, et on va garantir la propriété suivante :

$$\mathrm{A}_1 = \mathrm{A}_2 \quad \text{si et seulement si} \quad \texttt{uniq}(\mathrm{A}_1) = \texttt{uniq}(\mathrm{A}_2) \tag{1}$$

Pour cela, on pose $\texttt{uniq}(\bot) = 0$, $\texttt{uniq}(\top) = 1$ et pour un arbre A quelconque distinct de $\bot$ et $\top$, on choisira pour $\texttt{uniq}(\mathrm{A})$ une valeur arbitraire supérieure ou égale à 2 (nous étudierons plus loin comment ce nombre peut être choisi). Cette valeur $\texttt{uniq}(\mathrm{A})$ sera stockée dans le nœud de l'arbre.

Pour représenter en OCaml un arbre $\mathrm{A} = i \to \mathrm{A}_1, \mathrm{A}_2$ avec ces informations supplémentaires, on va donc dorénavant utiliser un type légèrement modifié, noté `ut`, où dans le cas d'un noeud de la façon suivante :

```
type uniq = int

type ut =
  | UT                              (* feuille ⊤ *)
  | UF                              (* feuille ⊥ *)
  | UComb of uniq * int * ut * ut   (* i → A1,A2 *)
```

14. Proposer une fonction `uniq_ut` de signature `ut -> uniq` (ou `ut -> int`, `uniq` étant un simple alias vers un entier) prenant en argument un arbre combinatoire A représenté par un objet de type `ut`, et renvoyant $\texttt{uniq}(\mathrm{A})$.

On envisage une fonction `equals_ut` sur des paires d'arbres combinatoires définie par les relations suivantes :

$$\begin{cases} \texttt{equals\_ut}(\top, \top) = \texttt{true}, \\ \texttt{equals\_ut}(\bot, \bot) = \texttt{true}, \\ \texttt{equals\_ut}((i_1 \to \mathrm{L}_1, \mathrm{R}_1), (i_2 \to \mathrm{L}_2, \mathrm{R}_2)) = (i_1 = i_2) \wedge \big(\texttt{uniq}(\mathrm{L}_1) = \texttt{uniq}(\mathrm{L}_2)\big) \wedge \big(\texttt{uniq}(\mathrm{R}_1) = \texttt{uniq}(\mathrm{R}_2)\big), \\ \texttt{equals\_ut}(\mathrm{A}_1, \mathrm{A}_2) = \texttt{false} \quad \text{sinon} \end{cases}$$

15. Justifier que la fonction `equals_ut` permet bien de tester si deux arbres combinatoires sont égaux en temps constant.

16. Proposer une implémentation de la fonction `equals_ut` de signature `ut -> ut -> bool` prenant en argument deux arbres combinatoires dont les nœuds internes portent les informations « `uniq` » et renvoie un booléen indiquant s'ils sont égaux en temps constant ($O(1)$).

Pour construire notre dictionnaire, on utilisera le principe des tables de hachage. Une table de hachage est un tableau de taille H, dont chaque case contient des listes chaînées de couple (clé, valeur). Chaque couple (clé, valeur) stocké dans le dictionnaire sera un des éléments de la liste chaînée contenue dans la case d'index i, où $i = \texttt{hash\_ut}(\text{clé}) \mod \mathrm{H}$, `hash_ut` étant une fonction dite *de hachage* sur les clés, à valeur dans les entiers positifs. Chaque case du tableau est appelée *seau*.

On s'intéresse ici à des dictionnaires dont les clés seront des arbres combinatoires. Il nous faut donc disposer d'une fonction de hachage `hash_ut` prenant en argument des arbres combinatoires. On définit une telle fonction de la manière suivante :

$$\begin{cases} \texttt{hash\_ut}(\bot) = 0 \\ \texttt{hash\_ut}(\top) = 1 \\ \texttt{hash\_ut}(i \to \mathrm{A}_1, \mathrm{A}_2) = 19^2 \times i + 19 \times \texttt{uniq}(\mathrm{A}_1) + \texttt{uniq}(\mathrm{A}_2) \mod m \end{cases}$$

où `m` est une grande constante entière que l'on suppose préalablement définie. On pourra supposer qu'elle a été choisie aussi grande que possible mais de manière à garantir que le calcul de $19^2 \times i + 19 \times \texttt{uniq}(\mathrm{A}_1) + \texttt{uniq}(\mathrm{A}_2)$ ne débordera jamais. Le choix de la valeur 19 a été fait sur des considérations pratiques que l'on ne détaillera pas ici.

17. Proposer une fonction `hash_ut` de signature `ut -> int` prenant en argument un arbre combinatoire A (incluant les données « `uniq` » dans ses nœuds internes) et retournant $\texttt{hash\_ut}(\mathrm{A})$.

18. Justifier que si $\texttt{equals\_ut}(\mathrm{A}_1, \mathrm{A}_2)$, on a $\texttt{hash\_ut}(\mathrm{A}_1) = \texttt{hash\_ut}(\mathrm{A}_2)$. La réciproque est-elle vraie ?

Il est temps, à présent, de construire un dictionnaire. On rappelle que clés et valeurs sont des arbres combinatoires de type `ut`. On définit un type pour le dictionnaire, qui est constitué d'un tableau de taille H contenant des listes de couples (clé, valeur) :

```
type dict = { data : (ut * ut) list array; mutable n : int }
```

On impose l'invariant suivant pour le type précédent : à tout moment, n est le nombre de couples clé-valeurs présents dans le dictionnaire.

Pour simplifier, on suppose que la taille H du tableau est choisie de façon adéquate lors de la création du dictionnaire, et n'est plus changée ensuite. En outre, dans le cadre de ce problème, on ne fera qu'ajouter des couples (clé, valeur) et rechercher des valeurs associées à des clés. Onn n'envisage pas de supprimer ou de modifier des associations.

On fournit une fonction de signature `int -> dict` qui crée un dictionnaire vide :

```
let dict_create h =
  { data = Array.make h []; n = 0 }
```

19. Proposer une fonction add de signature `dict -> ut -> ut` qui prend en argument un dictionnaire (table de hachage), un arbre combinatoire servant de clé, un arbre combinatoire servant de valeur, et ajoute au dictionnaire le couple (clé, valeur) en temps constant $O(1)$. La fonction add **ne vérifie pas** si la clé est déjà présente dans le dictionnaire.

20. Proposer une fonction `mem` de signature `dict -> ut -> bool` qui prend en argument un dictionnaire et une clé sous la forme d'un arbre combinatoire, et renvoie un booléen indiquant si l'arbre combinatoire fait partie des clés mémorisées dans le dictionnaire.

21. Proposer une fonction `find` de signature `dict -> ut -> ut` qui prend en argument un dictionnaire et une clé sous la forme d'un arbre combinatoire, et renvoie la valeur (arbre combinatoire) associée à la clé si celle-ci est présente dans le dictionnaire, et lève l'exception `Not_Found` dans le cas contaire.

Après 22518 ajouts dans une table de hachage avec H = 19997, la longueur des seaux dans la table n'excède jamais 7. Plus précisément, la répartition est la suivante :

Longueur du seau	0	1	2	3	4	5	6	7
Nombre de seaux de cette longueur	6450	7340	4080	1617	400	96	11	3

22. Quel serait le nombre moyen d'appels à `equals_ut` si l'on recherche avec `find` une clé qui n'est pas présente dans le dictionnaire ? On donnera le chiffre avec deux décimales, en le justifiant soigneusement.

23. Quel serait le nombre moyen d'appels à `equals_ut` si l'on recherche la valeur associée à une clé présente dans le dictionnaire ? On donnera le chiffre avec deux décimales, en le justifiant soigneusement.

5 Mémoïsation des fonctions

L'arbre combinatoire dont le cardinal correspond au nombre de pavages contient plusieurs dizaines de millions de nœuds pour $n = 8$, mais seulement 22518 sous-arbres distincts. Pour gagner en temps et en espace, on souhaite représenter l'ensemble des sous-arbres égaux par un arbre unique. Pour ce faire, nous allons construire chaque sous-arbre avec une fonction `build` de signature `int -> ut -> ut -> ut` qui prend en argument un entier i et deux sous-arbres A_1 et A_2, et, si l'on a déjà construit précédemment un arbre $i \to A_1, A_2$ prélablement, renvoie cet arbre, sinon crée un nouvel arbre $i \to A_1, A_2$.

Dans la suite, on suppose avoir créé un dictionnaire appelé `memo_dict`, initialement vide. Cet dictionnaire est accessible depuis toutes les fonctions que l'on écrira dans cette dernière section. Les arbres déjà créés seront mémorisés dans ce dictionnaire `memo_dict` (sous forme d'un couple (clé, valeur) où l'arbre en question est à la fois la clé *et* la valeur).

24. Proposer une implémentation de la fonction `build`. On précisera comment on utilise le dictionnaire `memo_dict` et comment on choisit la valeur « uniq » mémorisée dans les nœuds afin de garantir la condition (1) présentée au début de la partie IV si *tous* les arbres sont construits par un appel à `build`.

25. *Sans l'implémenter*, expliquer comment on peut se servir du *même* dictionnaire `memo_dict` afin de constrire une fonction `inter_ut` de signature `ut -> ut -> ut` calculant l'intersection de deux arbres combinatoires de façon à ne jamais recalculer l'intersection de deux arbres combinatoires si cette intersection a déjà été calculée.

26. Justifier que $T(\texttt{inter}(A_1, A_2)) \leqslant T(A_1) \times T(A_2)$, où $T(A)$ représente le nombre de sous-arbres *distincts* d'un arbre combinatoire A, si les intersections d'arbres ont été calculées avec la fonction précédente.

On suppose avoir réécrit les différentes fonctions de la partie III pour utiliser les fonctions `build` et `inter_ut` à chaque fois que l'on construit un arbre ou que l'on calcule l'intersection de deux arbres.

27. Proposer un majorant de la complexité temporelle que l'on peut espérer pouvoir obtenir avec la fonction `pavage`.