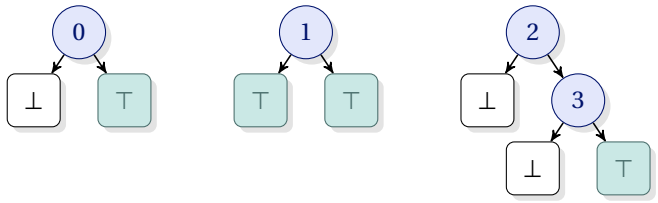
Arbres combinatoires
(Adapté d'un sujet d'informatique X 2011)

Introduction

1 Arbres combinatoires

1. On trouve exactement un ensemble pour chacune des feuilles $\top$ dans l'arbre, dont les éléments correspondent à l'ensemble des étiquettes de la branche menant à cette feuille qui ont été suivies d'une descente vers la droite. On a donc $\{\{2\},\{1\},\{0\},\{0,1,2\}\}$.

2. Les ensembles proposés correspondent aux arbres suivants :



3. Si $A \neq \bot$ est réduit à une feuille, ce ne peut être que $\top$.

Sinon, on considère la feuille que l'on atteint en descendant systématiquement vers la droite (laquelle existe, puisque l'on a un arbre binaire strict). Puisque l'arbre n'est pas réduit à une feuille, elle a nécessairement un parent dont elle est le fils droit. Par (suppression), cette feuille est $\top$.

Alternativement, on peut aussi considérer un nœud de profonde maximale (qui a donc deux feuilles pour enfants) et s'intéresser à la feuille à sa droite.

4. Il y a 2^p parties de $[\![0\,..\,p-1]\!]$, donc 2^{2^p} parties de parties de $[\![0\,..\,p-1]\!]$.

Or il y a bijection entre $\mathcal{P}(\mathcal{P}([\![0\,..\,p-1]\!]))$ et les arbres combinatoires étiquetés par $[\![0\,..\,p-1]\!]$ considérés. En effet, tout arbre combinatoire A est associé à un ensemble de parties $S(A) \in \mathcal{P}(\mathcal{P}([\![0\,..\,p-1]\!]))$ comme indiqué dans l'énoncé.

Inversement, pour tout élément $s \in \mathcal{P}(\mathcal{P}([\![0\,..\,p-1]\!]))$ on peut construire un *unique* arbre combinatoire A vérifiant $s = S(A)$. En effet, si $s \neq \{\}$ et $s \neq \{\{\}\}$ (représentés de façon unique par $\bot$ et $\top$), la racine porte nécessairement l'étiquette du plus petit entier k présent dans au moins un des ensembles de s :

- il ne peut pas être plus grand à cause de (ordre)
- il ne peut pas être plus petit à cause de (suppression), car dans ce cas le fils droit serait $\bot$

Et d'après l'interprétation que l'on fait des arbres combinatoires en terme d'ensembles, ses deux sous-arbres doivent décrire respectivement les parties $\{p \in s \mid k \notin p\}$ et $\{p \in s \mid k \in p\}$. Par induction structurelle, on peut donc construire un arbre combinatoire A tel que $s = S(A)$ et cette construction est *unique*.

Il y a donc exactement 2^{2^p} arbres combinatoires distincts.

2 Fonctions sur les arbres combinatoires

5. Comme il existe une feuille $\top$ en bas à droite, on se contente de parcourir la branche de droite en accumulant les étiquettes sur ce chemin :

```
let rec one_set = function
  | T -> []
  | F -> failwith "Impossible (suppression)"
  | Comb(i, _, rc) -> i::one_set rc
```

Il ne s'agit pas de la seule possibilité, on pourrait également descendre vers la gauche (en ne prenant pas l'étiquette) si le sous-arbre gauche n'est pas réduit à $\bot$.

6. Il s'agit, inversement, de construire un arbre-branche vers la droite, avec une seule feuille $\top$ en bas à droite, et les éléments de l'ensemble comme étiquettes sur la branche, ordonnés par ordre croissant :

```
let rec singleton = function
  | [] -> T
  | h::t -> Comb(h, F, singleton t)
```

7. Comme les éléments de l'ensemble et les étiquettes de l'arbre A sont ordonnées, on peut tester l'appartenance en descendant dans une unique branche, en se dirigeant vers la gauche si l'étiquette rencontrée n'est pas dans l'ensemble, vers la droite si elle l'est. L'ensemble se trouve dans S(A) si et seulement si on atteint une feuille $\top$ après avoir trouvé tous les éléments de l'ensemble passé en argument :

```
let rec member set = function
  | T -> set = []
  | F -> false
  | Comb(i, lc, rc)
      -> match set with
           | h::t when h=i -> member t rc
           | h::t when h<i -> false (* facultatif *)
           | _ -> member set lc
```

On notera la présence d'une ligne facultative dans la fonction précédente : si $h < i$, on sait qu'on ne trouvera h comme étiquette d'aucun des descendants (ordre), et il est inutile de continuer. Cependant, on pourrait également poursuivre la recherche (vouée à l'échec) dans le sous-arbre gauche `lc`.

La complexité est de façon évidente la hauteur de l'arbre, donc *a fortiori* $O(p)$.

8. On l'a dit, il y a bijection entre arbres et ensembles, donc $S(A_1) = S(A_2)$ si et seulement si les deux arbres sont identiques, tant sur la forme que sur les étiquettes. On peut donc écrire

```
let rec equals a1 a2 = match a1, a2 with
  | T, T -> true
  | F, F -> true
  | Comb(i1, lc1, rc1), Comb(i2, lc2, rc2)
      -> i1=i2 && equals lc1 lc2 && equals rc1 rc2
  | _ -> false
```

Mais on pourrait tout aussi bien écrire simplement

```
let equals a1 a2 = a1 = a2
```

Ou même

```
let equals = (=)
```

En effet, il faut comprendre comment l'égalité fonctionne en OCaml pour des types construits : deux objets sont égaux si et seulement si ils ont la même étiquette/constructeurs, et, dans le cas d'un constructeur avec des arguments, si ces arguments sont eux-même égaux. Ce qui est exactement ce que fait la première fonction !

On visite les deux arbres en s'arrêtant sur la première différence, donc la complexité est $O(\min(|A_1|, |A_2|))$.

9. On compte simplement les feuilles $\top$, qui représentent chacune un ensemble, comme on l'a évoqué dès la première question, ce qui donne :

```
let rec cardinal = function
  | T -> 1
  | F -> 0
  | Comb(_, lc, rc) -> cardinal lc + cardinal rc
```

10. Il y a ici plusieurs cas à étudier soigneusement :

```
let rec inter a1 a2 = match a1, a2 with
  | F, _ -> F            (* S(A1) = ∅  donc  S(A) = ∅ *)
  | _, F -> F            (* S(A2) = ∅  donc  S(A) = ∅ *)
  | T, T -> T            (* S(A1) = S(A2) = {∅}  donc  S(A) = {∅} *)
  | T, Comb(i, lc, rc) -> inter T lc
  | Comb(i, lc, rc), T -> inter lc T
  | Comb(i1, lc1, rc1), Comb(i2, lc2, rc2)
     -> if i1<i2 then inter lc1 a2 else
        if i1>i2 then inter a1 lc2 else
        match inter rc1 rc2 with  (* cas i1=i2 *)
          | F -> inter lc1 lc2
          | rc -> Comb(i1, inter lc1 lc2, rc)
```

Clarifions les trois derniers cas du filtrage principal...

- Si $S(A_1) = \{\emptyset\}$ et que A_2 contient au moins un nœud interne étiqueté par i. i n'appartient à aucune partie de $S(A_1)$, or toutes les parties construites à partir du sous-arbre droit de A_2 contiennent i. Aucune de ces parties ne figure donc dans l'intersection, on ne considère que le sous-arbre droit.
- Si A_1 contient au moins un nœud interne étiqueté par i et $S(A_2) = \{\emptyset\}$, on a le cas miroir du précédent.
- Si ni A_1 ni A_2 ne sont des feuilles, on dispose des étiquettes i_1 et i_2 dans leurs racines respectives. Si $i_1 < i_2$, i_1 n'est présent dans aucune des parties de $S(A_2)$, aussi avec la même démarche que précédemment on peut se limiter au sous-arbre gauche de A_1. Si $i_1 > i_2$, c'est i_2 qui n'est présent dans aucune des parties de $S(A_1)$. Si enfin $i_1 = i_2$, l'intersection est l'union de l'intersection des parties de $S(A_1)$ qui contiennent $i_1 = i_2$ et des parties de $S(A_2)$ qui contiennent $i_1 = i_2$, et de l'intersection des parties de $S(A_1)$ qui ne contiennent pas $i_1 = i_2$ et des parties de $S(A_2)$ qui na contiennent pas $i_1 = i_2$. Mais attention, l'énoncé impose de ne pas avoir de sous-arbre droit égal à $\perp$ (suppression), il faut donc regarder si l'intersection des deux sous-arbres droits est vide ou non !

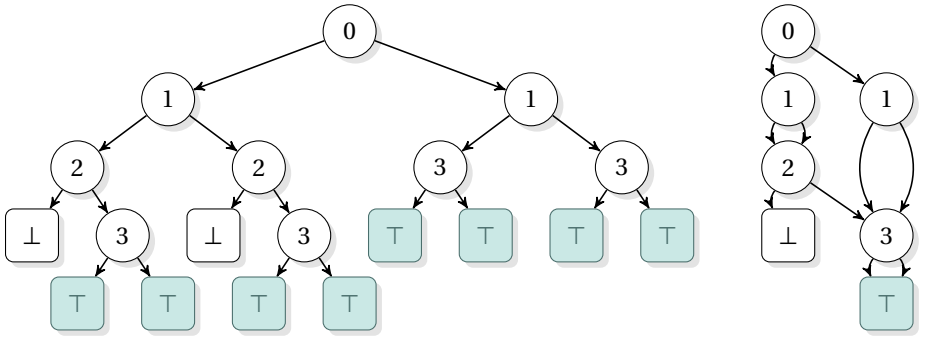
3 Application au dénombrement

11. Il existe $n \times (n-1)$ façons de poser le domino « horizontalement » (la moitié de gauche peut se situer sur n'importe quelle case de l'échiquier à l'exception de la dernière colonne). De même, il existe $n \times (n-1)$ façons de le poser « verticalement ».

On a donc au total $2n(n-1)$ placements possibles pour les dominos sur l'échiquier $n \times n$.

12. Cette question est difficile, en particulier avec la complexité attendue : elle ne permet pas de travailler naïvement avec une simple récursion, car la complexité, liée à la taille de l'arbre, serait bien plus grande que p.

Par exemple, si l'on a quatre positions de dominos et que les positions 0 et 2 couvrent la case qui nous intéresse, on voudrait construire l'arbre ci-dessous à gauche décrivant l'ensemble $\{\{0\},\{0,1\},\{0,3\},\{0,1,3\},\{1,2\},\{1,3\},\{1,2,3\},\{2\}\}$. On ne peut le construire tel quel en temps linéaire. Mais on va profiter de l'immutabilité pour réutiliser les sous-arbres identiques, et construire en fait l'« arbre » de droite :



Pour clarifier la démarche, utiliser un invarant n'est jamais une mauvaise idée. On va imposer l'invariant suivant : au début de la boucle **for**,

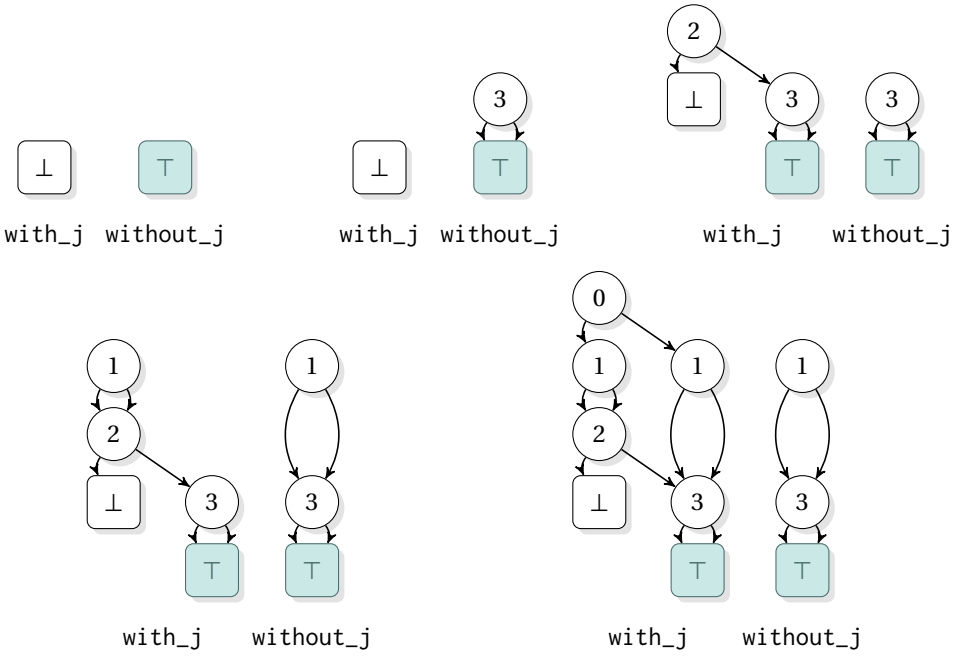
- `without_j` contient un arbre combinatoire correspondant à tous les ensembles de placements de dominos strictement supérieurs à `i` ne couvrant pas la case `j` ;
- `with_j` contient de même un arbre combinatoire correspondant à tous les ensembles de placements de dominos strictement supérieurs à `i` couvrant exactement une fois la case `j`.

Initialement, cela signifie que `with_j` doit être initialisé avec l'arbre ⊥ ($\{\}$) et `without_j` avec l'arbre ⊤ ($\{\{\}\}$ puisque $\{\}$ est un ensemble de placements qui ne couvre pas j). Ensuite, on se contente de les mettre à jour, dans l'ordre décroissant des `i` pour assurer (ordre), `m.(i).(j)` indiquant si le placement `i` couvre ou non la case `j` pour savoir comment se passe la mise à jour. On retourne `with_j` à l'issue de la boucle.

Seule subtilité : `with_j` pouvant être ⊥, il faut faire attention à la condition (suppression). Il n'y a pas ce problème avec `without_j` qui contient toujours au moins $\{\}$.

```
let column m j =
  let with_j = ref F and without_j = ref T in
  for i = (Array.length m)-1 downto 0 do
    if m.(i).(j) then with_j := Comb(i, !with_j, !without_j)
    else begin
      if !with_j <> F then with_j := Comb(i, !with_j, !with_j);
      without_j := Comb(i, !without_j, !without_j);
    end
  done;
  !with_j;;
```

Pour bien comprendre ce qui se passe, l'évolution des deux arbres, pour l'exemple proposé, est illustrée ci-dessous : on débute avec ⊥ et ⊤, puis on considère 3 qui ne couvre pas la case (puisque `with_j` est ⊥, seul `without_j` est mis à jour), 2 qui couvre la case (seul `with_j` est mis à jour), 1 qui ne couvre pas la case (les deux arbres sont modifiés), et enfin 0 qui couvre la case (seul `with_j` est modifié).



Notons que dans l'exemple précédent, les deux arbres ne sont pas indépendants (l'arbre final de droite est, notamment, une partie de l'arbre final de gauche).

Il est évident que la fonction proposée est bien en $O(p)$.

13. Il s'agit ensuite simplement, pour trouver tous les pavages, de caculer l'intersections de *tous* les résultats retournés par `column m j`.

On peut par exemple procéder avec une boucle et une référence :

```
let pavages n =
  let m = gen_mat n in
  let res = ref (column m 0) in
  for i = 1 to (n*n)-1 do
    res := inter !res (column m i)
  done;
  !res;;
```

Puisqu'il s'agit d'appliquer une intersection entre tous les éléments, cela se prête assez naturellement au repliement, aussi peut-on également écrire :

```
let pavages n =
  let m = gen_mat n in
  let trees = List.init (n*n) (column m) in
  List.fold_left inter (List.hd trees) (List.tl trees)
```

On notera que l'on commence le repliement avec l'un des arbres, car l'élément neutre pour `inter`, qui pourrait donc servir de point de « départ », serait un arbre contenant l'intégralité des parties des parties de E_p, ce qu'il nous faudrait construire (ce qui n'est pas extrêmement difficile, mais ici inutite).

La façon dont on calcule les n^2-1 intersections a d'immenses conséquences sur le temps de calcul, même si elles sont difficiles à prévoir. Pour des raisons sur lesquelles nous reviendrons, il peut être intéressant d'envisager d'effectuer les intersections deux par deux, puis de recommencer sur les résultats, et ainsi de suite, jusqu'à ce qu'il ne reste qu'un seul arbre. On pourra par exemple initialiser un tableau avec tous les arbres obtenus avec `column`, et itérativement en réduire le nombre :

```
  let pavages n =
    let m = gen_mat n in
    let t = Array.init (n*n) (column m) in
    let remaining_trees = ref (n*n) in
    while !remaining_trees > 1 do
      for i = 0 to !remaining_trees/2 - 1 do
        t.(i) <- inter t.(2*i) t.(2*i+1);
      done;
      if !remaining_trees mod 2 = 1
        then t.(0) <- inter t.(0) t.(!remaining_trees - 1);
      remaining_trees := !remaining_trees / 2;
    done;
    t.(0)
```

4 Dictionnaires

14. Rien de bien compliqué ici :

```
let uniq_ut = function
  | UT -> 1
  | UF -> 0
  | UComb (u, _, _, _) -> u
```

15. Si $A_1 = \top$, alors A1 = A2 si et seulement si $A_2 = \top$. Même chose si $A_1 = \bot$. Si ni A_1, ni A_2 ne sont $\top$ ou $\bot$, alors il sont égaux s'ils sont intégralement identiques. La condition $\texttt{uniq}(L_1) = \texttt{uniq}(L_2)$ assure $L_1 = L_2$, la condition $\texttt{uniq}(R_1) = \texttt{uniq}(R_2)$ assure $R_1 = R_2$. A_1 et A_2 sont bien égaux si et seulement si ils ont même étiquette de la racine et mêmes sous-arbres gauche et droits.

La fonction effectue un nombre fini de tests tous en temps constant $O(1)$, donc la comparaison de deux arbres se fait en temps constant.

16. On ne veut *pas* utiliser le champ `uniq` de la racine. Si on pouvait le faire, on pourrait tester l'égalité simplement avec $\texttt{uniq}(A_1) = \texttt{uniq}(A_2)$. La raison est à chercher dans la suite : on veut pouvoir savoir si un arbre est déjà dans le dictionnaire, et s'il s'y trouve, on n'a pas pu deviner la valeur du champ `uniq` avant de l'y trouver.

On peut en théorie utiliser le champ `uniq` pour tester l'égalité à $\top$ et $\bot$, *sous réserve de prendre des précautions ensuite*. On évitera ici. Pour le reste, on se content de retranscrire les règles proposées.

```
let equals_ut a1 a2 = match a1, a2 with
  | UT, UT -> true
  | UF, UF -> true
  | UComb (_, i1, l1, r1), UComb (_, i2, l2, r2)
      -> i1 = i2 && uniq_ut l1 = uniq_ut l2 && uniq_ut r1 = uniq_ut r2
  | _ -> false
```

17. Dans la mesure où le sujet nous dispense de nous inquiéter d'un quelconque débordement, la fonction est assez immédiate à implémenter :

```
let hash_ut = function
  | UT -> 1
  | UF -> 0
  | UComb (_, i, l, r) -> (uniq_ut r + 19*(uniq_ut l + 19*i)) mod m
```

18. Si $\texttt{equals\_f}(A_1, A_2)$, alors soit $A_1 = A_2 = \bot$ et les deux hachés sont 0, soit $A_1 = A_2 = \top$ et les deux hachés sont 1, soit A_1 et A_2 ne sont pas réduits à une feuille, auquels cas ils ont même étiquettes sur la racine, mêmes sous-arbres gauches (qui ont donc les mêmes valeurs `uniq`) et mêmes sous-arbres droits (même chose), donc les hachés sont, là encore, égaux.

La réciproque n'est évidemment pas vraie. L'argument le plus simple est qu'on n'a que $2^{32}-1$ valeurs possibles différentes pour le haché, et possiblement bien plus d'arbres que cela ! En effet, dès que $p \geqslant 5$, $2^{2^p} > 2^{32}-1$.

19. On se contente d'ajouter un couple dans la bonne liste chaînée. **Attention de ne pas oublier le modulo pour rester dans le tableau, le haché peut fort bien être gigantesque** ! Le modulo qui apparaît dans la définition du haché n'a rien à voir avec le modulo nécessaire

pour rester dans le tableau. On s'attachera aussi à ne pas oublier d'incrémenter le champ `.n` de la structure mémorisant le nombre d'éléments stockés dans la table. Attention, ce n'est pas une référence, on ne peut pas utiliser `incr`.

```
let add dict k v =
  let h = Array.length dict.data in
  let hsh = hash_ut k mod h in
  dict.data.(hsh) <- (k,v) :: dict.data.(hsh);
  dict.n <- dict.n + 1
```

20. On parcourt la liste chaînée correspondant au haché, là encore ramené dans le bon intervalle. Attention à comparer les clés avec `equals_ut` et non `=` car le champ `uniq` de l'arbre que l'on recherche parmi les clés du dictionnaire ne peut pas être connu !

```
let mem dict a =
  let h = Array.length dict.data in
  let hsh = hash_ut k mod h in
  let rec aux = function
    | [] -> false
    | (k,v)::t -> equals_ut k a || aux t
  in aux dict.data.(hsh)
```

21. Il s'agit pratiquement de la même fonction, seul le résultat attendu est un peu différent :

```
let find dict a =
  let h = Array.length dict.data in
  let hsh = hash_ut k mod h in
  let rec aux = function
    | [] -> raise Not_found
    | (k,v)::t -> if equals_ut k a then v else aux t
  in aux dict.data.(hsh)
```

22. Dans le cas où l'arbre est construit pour la première fois, on a une équiprobabilité de tomber dans chacun des seaux, et si le seau contient p éléments, on effectuera p comparaisons, qui vont toutes échouer. On a donc, en moyenne, le nombre de comparaisons suivant :

$$\frac{7340}{19997}\times 1+\frac{4080}{19997}\times 2+\frac{1617}{19997}\times 3+\frac{400}{19997}\times 4+\frac{96}{19997}\times 5+\frac{6}{19997}\times 6+\frac{3}{19997}\times 7\simeq 1{,}13$$

23. Dans le cas où l'arbre se trouve dans la table, c'est un peu plus compliqué. Si l'on tombe dans un seau contenant p éléments, l'arbre peut se trouver dans n'importe quelle position de manière équiprobable, donc on fera

$$\left(1+2+\cdots+p\right)\times\frac{1}{p}$$

comparaisons en moyenne, soit $\left(p+1\right)/2$ comparaisons.

Maintenant, la probabilité de tomber dans chacun des seaux n'est pas équiprobable (déjà, la probabilité de tomber dans un seau vide est nulle !) On a au total

$$7340+4080\times 2+1617\times 3+400\times 4+96\times 5+11\times 6+3\times 7=22518$$

arbres dans la table. $3\times 7=21$ d'entre eux sont dans un seau contenant 7 arbres, donc la probabilité de tomber dans un seau contenant 7 arbres est $21/22518$ (davantage que les $3/19997$ pour un arbre au hasard).

Le nombre moyen de comparaisons pour un arbre dans la table sera donc

$$\frac{7340\times 1}{22518}\times\frac{1+1}{2}+\frac{4080\times 2}{22518}\times\frac{2+1}{2}+\frac{1617\times 3}{22518}\times\frac{3+1}{2}+\frac{400\times 4}{22518}\times\frac{4+1}{2}$$
$$+\frac{96\times 5}{22518}\times\frac{5+1}{2}+\frac{11\times 6}{22518}\times\frac{6+1}{2}+\frac{3\times 7}{22518}\times\frac{7+1}{2}\simeq 1{,}56$$

Il peut apparaître surprenant que le nombre moyen de comparaisons pour un arbre déjà dans la table soit supérieur à celui pour un arbre au hasard, car on s'attend à s'arrêter plus vite. Cependant, savoir que l'arbre est dans la table a des conséquences sur les probabilités. Imaginez une table avec 19997 cases et un seul arbre. Pour un arbre au hasard, vous faites $1/19997$ comparaisons en moyenne. Pour un arbre dans la table, vous en faites 1 en moyenne, puisque vous allez au moins le comparer avec lui-même ! Donc ce résultat apparamment surprenant est normal...

5 Mémoïsation des fonctions

24. Pour pouvoir le rechercher dans le dictionnaire, on construit un arbre avec les arguments. Bien entendu, le champ `uniq` de la racine aura une valeur arbitraire, mais comme la recherche ne prend pas en compte la valeur de ce champ (ce qui explique la forme particulière de `equals_ut` !), cela se passe bien.

```
let build i a1 a2 =
  let t = UComb (0, i, a1, a2) in
  try
    find memo_dict t
  with Not_found
    -> let t = UComb (2+memo_dict.n, i, a1, a2)
       in add memo_dict t t; t
```

Pour le champ `uniq`, on a choisi le nombre d'arbres mémorisés dans le dictionnaire *plus deux*. Cela assure que chaque arbre aura bien un identifiant unique, strictement supérieur à 1.

25. Pour mémoïser la fonction calculant l'intersection de deux arbres, on souhaite mémoriser dans un dictionnaire des associations entre les arguments de la fonction (deux arbres `a1` et `a2`) et le résultat (un arbre également).

Le hic, c'est que dans le dictionnaire dont on dispose, les clés ne sont pas des couples d'arbres, mais des arbres. Qu'à cela ne tienne : on peut stocker un couple d'arbres (`a1`, `a2`) dans une clé simplement en utilisant comme clé l'arbre $2n^2 \to$ `a1, a2` ! On choisit $2n^2$ car les étiquettes des arbres déjà stockés dans le dictionnaire sont toutes entre 0 et $p-1 < 2n^2$, aussi ne pourra-t-il y avoir aucune collisions avec l'utilisation précédente du dictionnaire (il serait sans doute plus simple de choisir -1, mais il est possible que cela occasionne des problèmes avec le calcul du haché car `mod` a un comportement particulier pour les entiers négatifs).

26. Tout sous-arbre de A est nécessairement le résultat de l'intersection d'un sous-arbre de A_1 et d'un sous-arbre de A_2. Par conséquent, on a nécessairement $T(\texttt{inter}(A_1, A_2)) \leqslant T(A_1) \times T(A_2)$,.

27. La question est fort délicate (et je ne suis pas sûr de comprendre ce qu'attendait le sujet original).

Compte tenu de la façon dont sont construit les arbres avec `column`, les arbres ainsi construits contiennent au plus $2p+1$ sous-arbres indépendants.

Une intersection « naïve » de l'ensemble de ces arbres, compte tenu de la question précédente, donnerait un arbre combinatoire final ayant au plus $(2p+1)^{n^2}$ sous-arbres indépendants. Seulement, c'est clairement une surestimation, car les arbres ayant une taille maximale de $2^{p+1}-1$, on n'atteindra jamais ce chiffre.

Cependant, si on envisage l'intersection des arbres deux par deux, puis des arbres résultant de ces intersections, jusqu'à obtenir un unique arbre, on obtient une majoration du nombre de sous-arbres en $(2p+1)^{2^{\log_2 n^2}} = (2p+1)^{n^2}$, plus « raisonnable ».

La construction d'un arbre A a un coût majoré par son nombre de sous-arbres $T(A)$. Le coût de la construction du dernier arbre domine celui de tous les autres, de même que la création des n^2 arbres initiaux. Par alleurs, calculer le cardinal (de façon raisonnable) aura une complexité majorée par le nombre de sous-arbres. On a donc une complexité en $O(p^p)$.

Cependant, la croissance de la taille des arbres reste surestimée. En effet, compte tenu de leur construction, aucun arbre combinatoire ne peut avoir une taille supérieure à 2^p. Les n^2 intersections se calculent donc, dans le pire des cas, en $n^2 2^p$ opérations. On peut donc également majorer la complexité par $O(p2^p)$, qui est inférieure à la précédente. L'intérêt de la question précédente dans le sujet n'est donc pas du tout évident.

De toute façon, les deux majorations sont d'énormes surestimations : pour $n = 8$, on parle de 10^{230} pour la première, 10^{36} pour la secondes, et pourtant environ un million d'intersections d'arbres seulement sont évaluées, ce qui permet, contre toute attente lorsque l'on regarde directement les complexités, d'avoir un résultat en quelques secondes seulement !

Résultats