DS MPSI : Arbres de mots

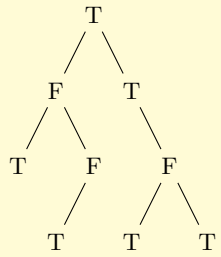
Durée : 2 heures

Question 1.

La hauteur, la taille et le fait que le sous-arbre vide est représenté une fois de plus que les sous-arbres non vides : voir cours.

Question 2.

L'arbre a_2 a pour ensemble de mots $\{00, 1, 101\}$.
$\{\varepsilon, 00, 010, 1, 110, 111\}$



Question 3.

≪ tout nœud dont les deux sous-arbres sont vides contient le booléen true ≫ : traduction avec le vocabulaire du cours : toutes les feuilles sont d'étiquette vraie.

Méthode 1 On traite à part l'ensemble vide.

L'arbre vide est réduit puisque tout nœud à deux fils qui est un sous-arbre de l'arbre vide (scoop, il n'y en a pas !) possède T pour étiquette. Or, l'arbre vide représente l'ensemble vide.

Les autres représentations de l'ensemble vide font intervenir des nœuds et des feuilles d'étiquettes fausses : elles ne sont pas réduites. D'où l'unicité de la représentaion de l'ensemble vide.

Pour les autres ensembles finis de mots, on raisonne par récurrence sur la longueur du plus long mot de E.

Si la longueur du plus long mot est nulle, c'est que $E = \{\varepsilon\}$. Alors $N(T, V, V)$ est réduit et représente E. Tout autre arbre censé représenter E aurait une hauteur non nulle. Puisque les branches se terminent toutes par T, on aurait un mot de E de longueur non vide : Absurde.

Donc l'ensemble $E = \{\varepsilon\}$ a une représentation unique par arbre réduit.

Supposons que tout ensemble dont le plus long mot est de longueur $k \leq n$ soit réprésentable de façon unique par un arbre réduit.

Soit E tel que sont plus long mot soit de longueur $n + 1$.

On note

$$G = \{m \in \Sigma^* \mid 0m \in E\}$$

et

$$D = \{m \in \Sigma^* \mid 1m \in E\}$$

Par HR, on peut représenter G et D par des arbres réduits et la représentation est unique(en effet, HR s'applique car le plus long mot de G (resp. D) est plus court que celui de E). Notons ces représentations $A(G)$ et $A(D)$.

Il est clair que $N(b, A(G), A(D))$ représente E avec b qui vaut T ou F selon que ε est dans E ou non.

Et si $N(b', L, R)$ représente aussi E, il faut que $b = b'$ puisque la racine est la seule possiblité d'exprimer l'appartenance ou non de ε à E.

Alors L est un arbre réduit qui représente les mots de l'ensemble représenté par $N(b', L, R)$ (c.a.d E) qui commencent par 0 : c'est G par HR. Il vient par HR que $L = A(G)$. De même $D = R$.

Par conséquent on peut représenter E par un arbre réduit et cette représentation est unique.

Méthode filou On pouvait être plus filou et considérer que l'existence d'un arbre réduit représentant tout ensemble de mots fini était donnée par le sujet.

Le sujet suggère en effet que tout arbre de mot possède un arbre de mot réduit qui représente le même ensemble. Or, tout ensemble fini de mots possède un arbre de mot qui le répresente. Ainsi tout ensemble est représenté par un arbre de mot réduit.

La question est alors celle de l'unicité : c'est la fin de la preuve de la méthode 1 : Soient $N(b,g,d)$ et $N(b',g',d'))$ représentant le même ensemble E et tels que g,d sont les deux uniques arbres réduits qui représentent les ensembles $\mathbb{M}(g)$ et $\mathbb{M}(d)$.

Alors b et b' sont égaux car la racine d'un arbre réduit vaut `true` si et seulement si ε est dans l'ensemble représenté. Comme g,g' représentent le même ensemble de mots (l'ensemble des mots de E qui commenecent par 0), alors l'unicité de g fait que $g=g'$. De même à droite. Et ainsi, les deux arbres sont égaux !

Question 4.

```
let zeros_puis_uns n =
  let rec aux right n =
    match right, n with
    | _,0 -> N(true,V,V)
    | false,_ -> N(false,aux false (n-1), aux true (n-1))
    | _,_ -> N(false,V, aux true (n-1))
  in  aux false n;;
```

On utilise un bouléen qui indique si on a déjà tourné à droite par le passé (donc on ne peut plus tourner à gauche dans le futur).

Pas de risque d'avoir un `N(false,V,V)` puisqu'on n'insère jamais directement un `V` qui n'a pas de père.

Question 5.

```
let rec k_uns k n =
  match k,n with
  | 0,0 -> N(true,V,V)
  | _, 0 -> V
  | 0,_ -> N(true,k_uns 0 (n-1), V)
  | _,_ -> let g = k_uns k (n-1) and
               d = k_uns (k-1) (n-1)
           in sN (false,g,d);;
```

on décrémente le nombre de 1 restant à écrire et le nombre de lettres restants. Quand tous les 1 sont utilisés, on ne peut plus que tourner à gauche.

Question 6.

```
(*on compte le nombre de T*)
let rec compter a =
  match a with
  | V ->0
  | N(true,g,d) -> 1 + (compter g) + (compter d)
  | N(_,g,d) -> (compter g) + (compter d)
```

Question 7.

```
(*a est supposé réduit*)
let  chercher a m =
  let n = Array.length m in
  let rec aux a i =
    Printf.printf "i=%d\n" i;
    match a,i with
    | V,_ -> false
    | N(false,_,_), x when x = n -> false
    | N(true,_,_), x when x = n -> true
    | N(b,g,d), _ -> Printf.printf "m.(i)=%d,b=%b\n" m.(i) b;
       if m.(i) = 0 then aux g (i+1)
       else aux d (i+1) in
  aux a 0;;
```

Question 8.

```
let rec enumerer a =
  let rec aux a acc pref =
    match a with
    | V -> acc
    | N(b,g,d) ->
       let acc1 = if b then (pref::acc) else acc
       in let acc2 = aux d acc1 (1::pref) in
          aux g acc2 (0::pref)
  in
  Array.of_list @@
    List.map
      (fun liste -> Array.of_list @@ List.rev liste)
      (aux a [] []);;
```

La fonction `aux` réalise un parcours en profondeur avec des opérations en $O(1)$ supplémentaires (ajout d'un élément en tête de liste au plus 3 fois par nœud).

La complexité de l'appel à `aux` est donc en $O(|a|)$.

La liste renvoyée par la fonction auxiliaire est celle des inverses des mots de a. Renverser chacun de ces mots a un coût proportionnel à la somme des longueurs des mots de $M(a)$: $n_2 = \sum_{m\in\mathbb{M}(a)} \ell(m)$.

Chaque mot de $M(a)$ a la même longueur qu'un chemin de nœuds non vides de a.

Comme a est réduit, toutes les branches (chemin depuis la racine à une feuille) correspondent à un mot de $M(a)$: il y a moins de branches que de mots de $M(a)$.

Or $|a|$ est inférieure à la somme des longueurs des branches (puisque certains nœuds sont alors comptés deux fois) qui est elle-même inférieure à la somme des longueurs des mots de $M(a)$ (puisqu'il y a moins de branches que de mots de $M(a)$).

Ainsi la complexité de `enumerer a` est un $O(n_2)$, ce qu'on veut.

Question 9.

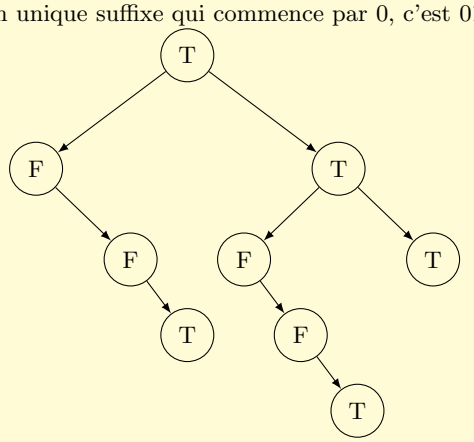
```
let rec ajouter a m =
  let m = Array.to_list m in
  let rec aux1 a m =
    match a, m with
    | N(_,g,d), [] ->  N(true,g,d)
    | N(b,g,d), x::q ->
       if x = 0 then N(b,aux1 g q,d) else N(b,g,aux1 d q)
    | V, _ ->
       let rec aux m =
         match m with
         | [] -> N(true,V,V)
         | x::q ->
            if x = 0 then N(false,aux q,V)
            else N(false,V,aux q)
       in aux m in
  aux1 a m;;
```

Question 10.

Les suffixes sont $\{\varepsilon, 1, 11, 011, 1011\}$.

L'étiquette de la racine est vraie.

Il y a un unique suffixe qui commence par 0, c'est 011. Donc il y a une seule branche à gauche.



Question 11.

La plus longue branche correspond au mot complet (c'est le seul suffixe du mot dont la longueur est celle du mot).

La définition d ehauteur prise ici est celle du plus grand nombre de nœuds non vides sur une branche

Il suffit de parcourir tout l'arbre en mémorisant le chemin pour arriver au nœud courant. Depuis ce nœud on explore les fils gauches et droits. Celui qui a la plus grande hauteur est le fils qu'il nous faut.

```
let retrouver_mot a =
  let rec aux a acc = match a with
    | V -> List.tl acc, -1
    | N(_,g,d) -> let acc1,h1 = aux g (0::acc)
                  and acc2,h2 = aux d (1::acc) in
                  if h1>h2 then acc1,h1+1 else
                    acc2, h2+1
  in let acc,h = aux a [] in
     (Printf.printf "%n;") h;
     List.rev acc;;
```

Question 12.

Identifions un nœud de $AS(m)$ avec le chemin (représenté en 0 et 1) de la racine à ce nœud. Ce chemin représente un début de branche, c'est à dire un préfixe d'un suffixe (une branche complète représente un suffixe et un seul).

Deux nœuds distinctes représentent des mots distincts puisque le chemin pour les joindre depuis la racine est différent.

Comme un préfixe de suffixe est un facteur on a que l'ensemble des nœuds (identifiés à des mots) est un sous-ensemble de celui des facteurs.

Considérons un facteur k du mot m. Alors m s'écrit ukv et donc k est un préfixe du suffixe kv. Notons A le nœud à la distance $|k|$ de la racine sur le chemin (c.a.d un début de branche) qui représente kv. Alors A représente k.

Les deux opérations décrites plus haut sont des injections réciproques l'une de l'autre : à un facteur j'associe un nœud de façon unique et réciproquement.

Ainsi $|AS(m)| = |\mathcal{F}(m)|$.

Il faut dénombrer les facteurs.

Pour un mot m, l'arbre des suffixes possède une branche de longueur $|m|+1$, puisque le mot m est son propre suffixe. Donc l'arbre des facteurs possède au moins $|m|+1$ nœuds. Un mot d'une lettre comme 0^n possède $n+1$ facteurs et donc $|AS(0^n)| = n+1$.

Majorons la taille de l'arbre :

De façon générale, le nombre de facteurs

— de longueur 1 est au plus n,

— de longueur 2 est au plus $n-1$,

— ...

— de longueur $n-1$ est au plus 2,

— de longueur n est au plus 1,

Le nombre maximal de facteurs non vides est donc :

$$\sum_{k=1}^{n}(n-k+1) = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

Il faut y ajouter le mot vide. Ainsi, le nombre maximal de facteurs est-il de :

$$\frac{n(n+1)}{2} + 1$$

C'est donc une majoration quadratique.

Il reste à trouver des mots de longeurs ℓ avec un nombre quadratique (en ℓ) de facteurs.

Essayons 0^p1^p.

Les mots de la forme 0^k1^q sont tous des facteurs distincts pour $0 \leq k \leq p$ et $0 \leq q \leq p$.

En effet $0^k1^q = 0^u1^v$ si et seulement si $k = u$ et $q = v$. Il y a ainsi au moins $p \times p$ facteurs différents. Ainsi, le nombre de facteurs est supérieur à $\frac{\ell}{2}^2$: c'est bien.

Avec 0^p1^p1, on trouve aussi au moins p^2 facteurs auxquels on ajoute tous les 0^q1^{p+1} soit $p+1$ facteurs. On obtient au moins p^2+p+1.

Or :

$$\frac{(2p+1)^2}{4} = \frac{4p^2+4p+1}{4} \leq p^2+p+1.$$

Le coefficient cherché est $\frac{1}{4}$: pour tout ℓ, il existe un mot de longueur ℓ avec au moins $\dfrac{\ell^2}{4}$ facteurs.

Question 13.

Un arbre des facteurs est réduit lorsque tous les nœuds contenant `(F)` possèdent deux fils non vide.

Les arbres a_5 et a_6 sont réduits mais pas l'arbre a_7 (3..5(F) a un fils vide.).

Les suffixes de 01001 sont $\varepsilon, 1, 01, 001, 1001$ et 01001.

Pour l'arbre a_5, on trouve de gauche à droite :

$$\varepsilon, 001, 01001, 1, 1001$$

Il manque 01 : a_5 n'est pas un arbre des facteurs réduit.

Pour l'arbre a_6, on trouve de gauche à droite :

$$\varepsilon, 001, 01, 010\ 01, 1, 1001$$

a_6 est un arbre des facteurs réduit.

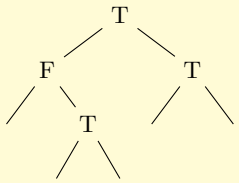
Pour l'arbre a_7, on trouve de gauche à droite :

$$\varepsilon, 001, 01, 010\ 01, 1, 10\ 01\ 01$$

a_7 n'est pas un arbre des facteurs et il n'est pas non plus réduit. Le chemin le plus long (droite gauche gauche) comporte 3 déplacement et $2+1$ lettres donc correspond à un mot de 6 lettres qui ne peut être un facteur.

Question 14.

1. Les nœuds ayant un fils vide contiennent tous `true` (ils ne peuvent contenir `False` car l'arbre est réduit).
   Or chaque nœud contenant `True` correspond à un suffixe de m.
   Comme m possède $|m|+1$ suffixes, a possède au plus $|m|+1$ arbres avec un fils vide.
   Il reste à montrer que a possède un nœud d'étiquette `true` avec deux fils non vides. Comme cela on aura montré que a possède au plus $|m|$ arbres avec un fils vide.
   m possède un zéro et un un, donc il a un suffixe qui commence par 0 et un autre qui commence par 1. Il vient que la racine (qui est d'étiquette `true`) possède deux fils.
   Ainsi a possède au plus $|m|$ nœuds avec un fils vide.
2. Les sous-arbres vides sont les extrémités des nœuds ayant au moins un fils vide. Ces derniers sont au plus $|m|$. Ils y au plus $2|m|$ sous-arbres vides. Si k est le nombre de sous-arbres non vides et k' celui des arbres vides, on a
$$|a| = k = k' - 1 \leq 2|m| - 1$$
3. On considère $m = 0^k1$ de longueur $k+1$.
   on montre que son arbre des facteurs réduits possède $2(k+1)-1 = 2k+1$ nœuds.
   C'est le cas pour 01, on obtient l'arbre des suffixes de e_1



   On construit e_k par récurrence en posant
$$e_{k+1} = \mathrm{N}(\mathrm{T}, e'_k, \mathrm{N}(\mathrm{T}, V, V))$$
   où e'_k est e_k dans lequel on a remplacé l'étiquette de la racine par `false`.
   Seul le nœud le plus à gauche possède une étiquette `false` et un fils vide.
   On transforme l'arbre en arbre des facteurs réduits :
   — remplacer le nœud le plus à gauche par `N(k,k+1,T,V,V)`.
   — ajouter `0,0` à toutes les étiquettes des autres nœuds.
   On obtient alors un arbre peigne de hauteur k et tel que chaque nœud de la branche gauche (sauf le dernier) possède un fils gauche d'étiquette `false` et un droit d'étiquette `true`. Il vient que cet arbre possède $k+1+k == 2k+ = 2|m|-1$ nœuds : ce qu'on veut.

Question 15.

Le nombre de facteurs est la taille de l'arbre de facteurs réduits plus la somme des $q-p$ pour chaque nœud `N(p,q,_,_,_)`.

```
let nb_facteurs a =
 let rec aux acc a = match a with
   | V -> acc
   | N(p,q,_,g,d)->
      let acc' = aux (acc + 1+ q-p) g in
      aux acc' d in
 aux 0 a;;
```

Question 16.

```
   let plpc m1 p q m2 i j =
  let rec aux k =
    if p + k < q && i + k < j && m1.(p + k) = m2.(i + k) then
      aux (k + 1)
    else
      k
  in
  aux 0
;;
```

Question 17.

```
let facteur a f =
  let lf = Array.length f in
  let rec aux a i =
    printf "  i=%d\n" i;
```

```
    (*i : position dans f*)
    match a, i with
    | V, _ -> printf "V\n"; false
    | _, j when j = lf -> true
    | N(p,q,_,g,d), _ ->
       let k = plpc f i lf m p q in
       printf "k=%d,i=%d,f.(i)=%d\n" k i f.(i);
       match k with
       | x when x = lf - i -> true
       | x  -> (*x < lf - i*)
          if x < q-p then false
          else
            begin
              (*x=q-p et on n'a pas fini de lire f*)
              if f.(i+x) = 0 then aux g (i+x+1)
              else aux d (i+x+1)
            end
  in aux a 0;;
```

Question 18.

```
(*oui je sais : m est censé être une variable globale, mais pour les
  tests, c'est plus pratique ainsi.
*)
let array_add acc cur p q m =
  let rec aux i acc cur =
    match i with
    | x when x = q -> cur::acc, cur
    | _ -> aux (i+1) (cur::acc) (m.(i)::cur)
  in aux p acc cur;;

  (*oui je sais : m est censé être une variable globale*)
let rec _tree_add a acc cur m = match a with
  | V -> acc
  | N(p,q,_,g,d) -> let acc1, cur1 = array_add acc cur p q m in
                    let acc2 = _tree_add g acc1 (0::cur1) m in
                    _tree_add d acc2 (1::cur1) m
;;

let facteurs a = let acc = _tree_add a [] [] m
in List.map (fun cur -> Array.of_list @@ List.rev cur) acc;;
```

Dans `_tree_add`, on fait un parcours en profondeur de `a` : ce parcours proprement dit coûte $O(|a|)$. On a vu que c'est $O(|m|)$ puisque l'arbre est réduit.

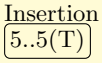
Le nombres de concaténations dans `acc` est du même ordre de grandeur que la taille de l'arbre des suffixes non réduit. En effet, les nœuds représentent des arbres chaînes de l'arbre des suffixes compactés en un seul. Donc, les concaténations dans `acc` coûtent un $O(|m|)$.

Par ailleurs, le nombre de concaténations dans `cur` (ou `cur1`) est égal au nombre d'arcs dans l'arbre des suffixes non réduit. Ce nombre est encore un $O(|a|)$ donc un $O(|m|)$.

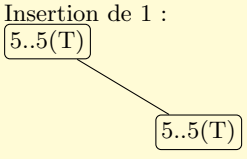
Le pacours de la liste finale `acc` par `List.map`, et l'inversion puis la conversion en tableaux des listes de caractères manipule exactement $\sum_{f\in F(m)} |f|$ caractères. La complexité de `facteurs` viens de là puisque $\sum_{f\in F(m)} |f| \geq |m|$

Question 19.

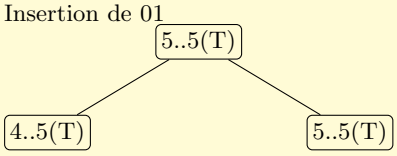
Insertion de ε :



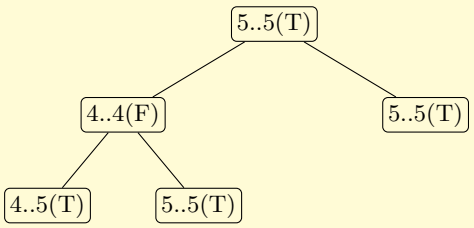
Insertion de 1 :



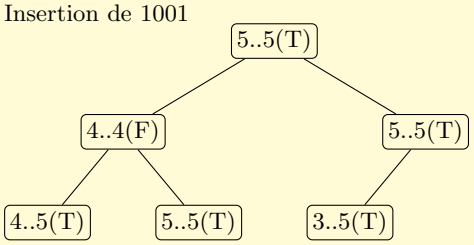
Insertion de 01



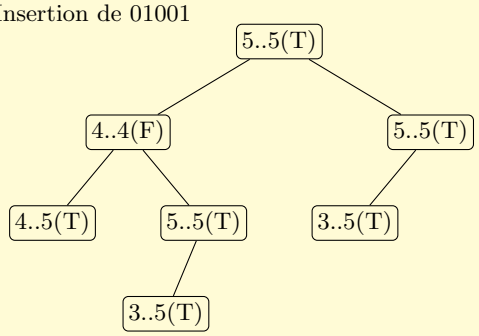
Insertion de 001



Insertion de 1001



Insertion de 01001



Question 20.

```
        (* CAS 1 *)
        begin
           match i+k with
           | x when x < n ->
              let t0 = N(i+k+1,n,true,V,V) and t1 = N(p+k+1,q,b,a0,a1)
              in if m.(i+k)=0 then
                   N(p,p+k,false,t0,t1)
                 else N(p,p+k,false,t1,t0)
           | _ -> (*i+k=n*)
              let t1 = V and t0 = N(p+k+1,q,b,a0,a1) in
              if m.(p+k)=0 then
                N(p,p+k,true,t0,t1)
              else N(p,p+k,true,t1,t0)
         end
```

```
        (* CAS 2 p+k=q*)
         begin
           match i+k with
           | x when x < n ->
              if m.(i+k)=0 then
                N(p,q,b,ajouter_suffixe (i+k+1) a0,a1)
              else N(p,q,b,a0,ajouter_suffixe (i+k+1) a1)
           | _ -> (*i+k=n*)
              N (p, q, true, a0, a1)
         end
```