1

Parcours de tableaux

« *Ordre, Permutations, Jeux.* »

2

Jouer avec les mots

« *Reconnaissance, Cosntruction, Codage.* »

2.1 Réécriture de mots

On considère les mots écrits sur l'alphabet {a, b, A, B}, tel que w =abbaBabAA par exemple. Soit ε le mot vide. On dit que deux mots u et v sont en relation si on peut réécrire des parties de u de façon à obtenir v après une suite de transformations effectuées en suivant les règles suivantes :

$$\begin{array}{rcl} \text{aA} & \to & \varepsilon \\ \text{Aa} & \to & \varepsilon \\ \varepsilon & \to & \text{aA} \\ \varepsilon & \to & \text{Aa} \\ \text{bB} & \to & \varepsilon \\ \text{Bb} & \to & \varepsilon \\ \varepsilon & \to & \text{bB} \\ \varepsilon & \to & \text{Bb} \\ \text{aab} & \to & \text{baa} \\ \text{baa} & \to & \text{aab} \\ \text{bba} & \to & \text{abb} \\ \text{abb} & \to & \text{bba} \end{array}$$

Par exemple :

$$\begin{array}{rl} & \text{aababaabAAABBB} \\ \to & \text{aababbaaAAABBBB} \\ \to & \text{aababbABBB} \\ \to & \text{aababbABBB} \\ \to & \text{aabbbaABBB} \\ \to & \text{aa} \end{array}$$

Question 1

Montrer que cette relation est une relation d'équivalence. Montrer que aa et AA commutent avec toutes les lettres.

Question 2

Montrer que tout mot w est équivalent à un mot xyz (formé de trois mots x, y et z mis bout à bout), où : x est de la forme aa...aa ou AA...AA et de longueur paire, y est de la forme bb...bb ou BB...BB et de longueur paire, z de la forme ababa...bab, ou ba...bab, ou ab...aba, ou ba...ba, c'est-à-dire alterne les lettres a et b, commençant par a ou b et finissant par a ou b. Un mot de la forme xyz est dit canonique.

Question 3

Proposer un codage des mots canoniques sous forme de triplets d'entiers.

Question 4

Écrire une fonction `forme3 (w:string) : bool` qui prend en entrée un mot w et teste si w est un mot alternant les lettres a et b, c'est-à-dire de la forme de z.

Question 5

Écrire une fonction `ajouter_a (iu, ju, ku: int * int * int) : (int * int * int)` qui prend en entrée un mot canonique u, codé par un triplet `(iu, ju, ku)`, et donne en sortie le codage `(iv, jv, kv)` du mot $v = u$a.

Question 6

Écrire une fonction `representant_canonique (w:string) : (int * int * int)` qui prend pour entrée un mot quelconque w et donne en sortie un triplet `(i, j, k)` codant un mot canonique équivalent à w.

CORRIGÉ

Question 1

Réflexivité : il suffit de prendre une suite de transformations réduite à l'ensemble vide.
Symétrie : pour chaque règle, la transformation inverse est dans l'ensemble des règles donc toute suite de transformations peut être inversée.
Transitivité : évidente.
On a donc une relation d'équivalence.

Commutation de aa avec toutes les lettres : aa commute évidemment avec a.
aaA $\to$ a $\to$ Aaa, donc aa commute avec A.
aab $\to$ baa est une règle, donc aa commute avec b.
aaB $\to$ BbaaB $\to$ BaabB $\to$ Baa, donc aa commute avec B.

Commutation de AA avec toutes les lettres : AAa $\to$ A $\to$ aAA, donc AA commute avec a.
AA commute évidemment avec A.
AAb $\to$ AAbaaAA $\to$ AAaabAA $\to$ bAA, donc AA commute avec b.
AAB $\to$ AABaaAA $\to$ AAaaBAA $\to$ BAA, donc AA commute avec B.

Remarquons que, de façon symétrique, bb et BB commutent également avec toutes les lettres.

Question 2

Pour transformer w, on commence par bouger toutes les suites de deux lettres consécutives identiques et les mettre au début du mot, les aa et AA précédant les bb et BB, et par simplifier toutes les occurrences de aA, Aa, bB ou Bb. On se retrouve avec un mot commençant par des aa et AA, continuant avec des bb et BB, et se terminant avec un mot qui alterne des a ou A avec des b ou B. On remplace alors chaque A par AAa et chaque B par BBb, puis on ramène les AA et les BB plus au début : on se retrouve avec une suite de aa et de AA, suivie d'une suite de bb et de BB, suivie d'un mot de la forme de z. En simplifiant les aaAA, les AAaa, les bbBB et les BBbb, on obtient mot xyz de la forme requise. Notons qu'il n'est pas demandé ici de montrer l'unicité de cette écriture.

Question 3

On peut coder x par un entier i, positif si x contient des aa et négatif s'il contient des AA, et de valeur absolue égale au nombre de lettres de x. De même, y peut être codé par un entier j. On peut coder z par un entier k, positif si z commence par un a et négatif si z commence par un b, et de valeur absolue égale au nombre de lettres de z.

Question 4

Supposons la longueur de w supérieure ou égale à 1. On regarde la première lettre de w, puis on teste si les suivantes alternes, en s'arrêtant dès qu'on trouve une erreur ou qu'on a parcouru tout le mot.

```
let forme3 (w:string) : bool =
  if w.[0] <> 'a' && w.[0] <> 'b' then false else

  let last = ref w.[0] and res = ref true in
  let n = String.length w in
  for i=1 to n-1 do
    if !last = 'a' && w.[i] <> 'b' || !last = 'b' && w.[i] <> 'a'
      then res := false;
    last := w.[i]
  done;
  !res
```

Question 5

La fonction n'est pas difficicle mais demande un peu d'attention pour ne pas faire d'étourderie : il est particulièrement recommandé d'écrire d'abord l'algorithme en détail. Si z est non vide et se termine par un b : `ku` strictement positif et pair ou strictement négatif et impair ; alors z s'allonge d'une lettre : `ku` augmente de 1 dans le premier cas et diminue de 1 dans le deuxième cas. Si z est non vide et se termine par un a : `ku` strictement positif et impair ou strictement négatif et pair ; alors z raccourcit d'une lettre et x change : `ku` diminue de 1 dans le premier cas et augmente de 1 dans le second cas, et `iu` augmente de 2 s'il était positif et diminue de 2 s'il était strictement négatif. Si z est vide : $\texttt{ku} = 0$; alors `ku` devient égal à 1. D'où la fonction suivante :

```
let ajouter_a (iu, ju, ku: int * int * int) : (int * int * int) =
  if ku = 0 then (iu, ju, 1) else (
    if ku > 0 && ku mod 2 = 0 then (iu, ju, ku+1)
    else if ku < 0 && ku mod 2 = 1 then (iu, ju, ku-1)
    else if ku > 0 && ku mod 2 = 1 then (
      if iu = 0 then (iu+2, ju, ku-1) else (iu-2, ju, ku-1)
    )
    else ( (* ku < 0 && ku mod 2 = 0 *)
      if iu = 0 then (iu+2, ju, ku+1) else (iu-2, ju, ku+1)
    )
  )
```

Question 6

On peut supposer qu'on dispose des fonctions `ajouter_b`, `ajouter_A` et `ajouter_B` similaires à celle de la question 5. Il est alors facile d'ajouter des lettres une par une pour construire un mot canonique pour chaque préfixe de w et finalement pour w.

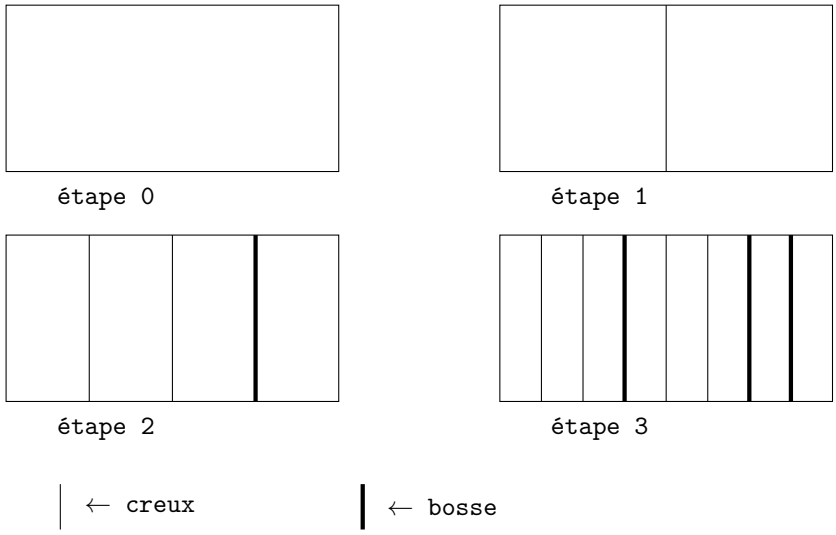
```
let representant_canonique (w:string) : (int * int * int) =
  let n = String.length w in
  let mot = ref (0, 0, 0) in
  for i=0 to n-1 do
    if w.[i] = 'a' then mot := ajouter_a !mot else
    if w.[i] = 'b' then mot := ajouter_b !mot else
    if w.[i] = 'A' then mot := ajouter_A !mot else
    mot := ajouter_B !mot
  done;
  !mot
```

La relation d'équivalence étudiée dans cet exercice correspond à un groupe donné par une permutation finie : les éléments sont les classes d'équivalence de la relation définie à la question 1, et on peut facilement vérifier que cette relation est compatible avec la concaténation. L'étude faite ici consiste en la construction d'une « structure automatique » pour le groupe, permettant de calculer un représentant distingué de la classe du produit de deux éléments donnés. L'étude de groupes automatiques est un domaine de recherche récent et actif en mathématiques.

⋆⋆⋆⋆⋆⋆⋆⋆

2.2 Pliage de papier

On prend une feuille de papier et on la plie n fois dans le sens vertical, en repliant à chaque fois la moitié droite sur la gauche. Les plis de la feuille, une fois redépliée, sont une suite de creux et de bosses. Voir figure :



Question 1

Combien y a-t-il de plis à la n-ième étape ?

Question 2

On représente chaque étape du pliage par un mot. Un creu est codé par un 0 et une bosse par un 1. Ainsi : $w_0 = \varepsilon$ (mot vide)
$w_1 = 0$
$w_2 = 001$
$w_3 = 0010011$
...
Montrer que w_i est toujours un préfixe de w_{i+1}, c'est à dire que le début de w_{i+1} coïncide avec w_i.

Question 3

Donner un algorithme de construction de w_n à partir de w_{n-1}. Écrire une fonction de calcul de w_n. On prendra comme entrée n et on renverra une liste `w` d'entiers remplie adéquatement.

Question 4

Écrire une fonction qui prend pour entrée un entier n et renvoie une liste contenant la représentation binaire de n, poids fort en tête.

Question 5

Les mots de la suite de pliage étant préfixes les uns des autres, on peut considérer le mot infini w dont ils sont tous préfixes. Proposer un algorithme qui prend pour entrée la représentation binaire de n et renvoie le n-ième bit de w.

CORRIGÉ

Question 1

Si on considère les intervalles entre les plis (y compris les côtés de la feuille), on a 1 intervalle, puis 2, puis 4, etc. ; à chaque étape, chaque intervalle est coupé en deux et le nombre d'intervalles double. Après n étapes, il y a donc 2^n intervalles, et le nombre de plis est $2^n - 1$.

Question 2

Déchirons le papier le long du pli de la première étape et débarrassons-nous de la moitié de droite : la k-ième étape de pliage du papier initial est comme la $(k-1)$-ième étape de pliage du demi-papier, et donc w_{k-1} forme la moitié gauche de w_k.

Question 3

Le pli du milieu est un creux et donc donne un 0 au milieu de w_n. Le demi-papier droit est plié, après la première étape, comme le demi-papier gauche, sauf qu'il est tourné dans l'autre sens. Soit $\mathrm{r}(w)$ le mot obtenu à partir d'un mot w en lisant les chiffres de droite à gauche, et c(w) le mot obtenu à partir de w en remplaçant chaque 0 par un 1 et chaque 1 par un 0 : on a la relation de récurrence

$$w_n = w_{n-1}\, 0\, \mathrm{c}(\mathrm{r}(w_{n-1}))$$

De cette relation, se déduit facilement la fonction permettant de construire w_n.

```
let rec motdepliage (n:int) : (int list) =
  if n = 0 then [0] else
  let pred = motdepliage (n-1) in
  let pred' = List.map (fun e -> 1-e) (List.rev pred) in
  pred @ (0::pred')
```

Question 4

Question classique et accessible à tous.

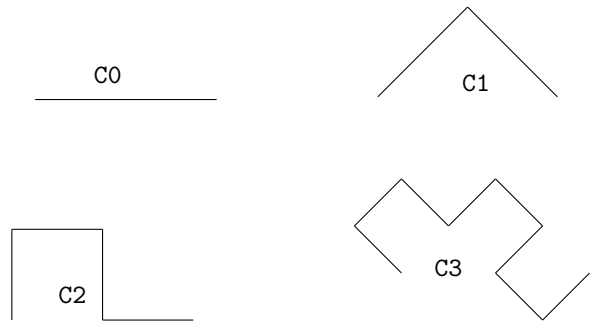
```
let binaire (n:int) : (int list) =
  let rec aux n acc =
    if n = 0 then acc
    else aux (n / 2) ((n mod 2) :: acc)
  in
  if n = 0 then [0] else aux n []
```

Question 5

On peut trouver un autre point de vue pour construire w_k par récurrence à partir de w_{k-1} : on intercale un nouveau pli entre chaque paire de plis consécutifs de w_{k-1}, et ce pli est alternativement 0 ou 1, le premier étant 0. Ainsi : $w_k = 0x1x0x...x1$, si $w_{k-1} = xxx...xx$. Donc le n-ième bit de w est facile à trouver si n est impair : c'est 0 si n est congru à 1 modulo 4 et c'est 1 si n est congru à 3 modulo 4. Si n est pair, on divise n par deux et on remarque que le n-ième bit de w_k est le $(n/2)$-ième bit de w_{k-1}, donc si $n/2$ est impair, il suffit encore une fois de tester si $n/2$ est congru à 1 ou à 3 modulo 4. Si $n/2$ est est pair, on redivise par deux, et ainsi de suite. Lorsque n est écrit en binaire, on regarde son bit le plus à droite (celui de poids le plus faible), puis le bit immédiatement à sa gauche, et ainsi de suite jusqu'à trouver un bit égal à 1. Soit x le n-ième bit de w. On a :
Si $n = 1\ 0\ 0\ ...\ 0$, alors $x = 0$.
Si $n = *\ *\ ...\ *\ 0\ 1\ 0\ 0\ ...\ 0$, alors $x = 0$.
Si $n = *\ *\ ...\ *\ 1\ 1\ 0\ 0\ ...\ 0$, alors $x = 1$.
Car un nombre se terminant par 0 1 est congru à 1 modulo 4 et un nombre se terminant par 1 1 à 3 modulo 4. Ceci donne un programme extrêmement simple.

```
let bit (nbin:int list) : int =
  let rev = List.rev nbin in
  let rec aux bin = match bin with
    | 1::1::_ -> 1
    | 1::0::_ | [1] -> 0
    | 0::q -> aux q
    | _ -> failwith "entrée invalide"
  in aux rev
```

Des études du genre de celle faite dans cet exercice peuvent servir à montrer des propriétés d'algébricité ou de transcendance du nombre étudié. C'est actuellement un domaine actif de recherche en France. On obtient une suite de courbes (C_n) approximant une courbe de Peano (c'est-à-dire une courbe qui remplit le plan), définie à partir de (w_n), en dessinant un segment de longueur $1/2^{n/2}$ pour chaque bit lu, et en tournant à droite (de $+\pi/2$) ou à gauche (de $-\pi/2$) à chaque pas, selon que le bit lu est 0 ou 1. Le premier segment est dessiné dans la direction $\pi/4$. La figure suivante illustre les premiers pas :



★★★★★★★

2.3 Plus longue sous-suite croissante

Une suite finie d'entiers $(a_i)_{1\leq i\leq n}$ est représentée par un tableau d'entiers `a` en OCaml.

On cherche la longueur de la (ou les) plus longue(s) sous-suite(s) croissante(s) (au sens large) de la suite d'entiers. Par exemple, si $n = 9$, et si les termes de la suite sont 1, 2, 6, 4, 5, 11, 9, 12, 9, la longueur cherchée est 6, et elle est atteinte par les sous-suites croissantes :

- 1, 2, 4, 5, 11, 12
- 1, 2, 4, 5, 9, 12
- 1, 2, 4, 5, 9, 9

Question 1

Écrire une fonction donnant la longueur de la plus longue sous-suite croissante de la suite `a`. Estimer le nombre de comparaisons que demande votre fonction. Pourriez-vous l'améliorer ?

Question 2

Modifiez votre fonction de manière à pouvoir afficher la plus longue (ou une des plus longues s'il y en a plusieurs de longueur maximale) sous-suites croissantes de `a`.

CORRIGÉ

Question 1

Il suffit de procéder par étapes, en ajoutant les éléments de `a` au fur et à mesure. Définissons un tableau `l` tel que `l.(p)` est la longueur de la plus longue sous-suite se terminant par `a.(p)`. Une plus longue sous-suite croissante se terminant par `a.(p)` est soit formée uniquement de `a.(p)` si pour tout $i < p$, on a `a.(i) > a.(p)`, soit obtenue en ajoutant `a.(p)` à la plus longue des plus longues sous-suites se terminant par un `a.(i)` tel que $i < p$ et `a.(i) < a.(p)`. À la fin, il ne reste plus qu'à chercher le plus grand des `l.(i)`, $1 \leq i \leq n$. Ceci nous donne la fonction suivante :

```
let longueur_sous_suite (a:int array) : int =
  let n = Array.length a in
  let l = Array.make n 0 in
  l.(0) <- 1;
  for p=1 to n-1 do
    let max = ref 0 in
    for j=0 to p-1 do
      if (a.(j) <= a.(p)) && (l.(j)+1 >= !max)
        then max := l.(j) + 1
    done;
    l.(p) <- !max
  done;
```

```
  (* Trouver le max *)
  let longueur = ref 1 in
  for i=1 to n-1 do
    if l.(i) > !longueur then
      longueur := l.(i)
  done;
  !longueur
```

On fait 2 fois $1 + 2 + ... + (n-1)$, soit $n(n-1)$ tests plus les $n-1$ derniers tests de la dernière boucle « for ». Ceci donne $(n+1)(n-1)$ tests. On peut gagner un peu de temps en classant `l` non par ordre croissant des valeurs de `p` mais par ordre croissant des valeurs de `a.(p)` et en simulant une structure dynamique pour insérer un nouvel élément sans devoir tout déplacer. Ce sera un peu lourd !

Question 2

On peut dans un premier temps créer un matrice auxiliaire `s` telle que `s.(p).(i)` contient le i-ième élément d'une plus longue sous-suite croissante se terminant par `a.(p)` (note : pas besoin d'une « marque de fin », on sait que le dernier élément est `a.(p)` et on connait la longueur d'une telle sous-suite, c'est `l.(p)`). Il est plus malin de remarquer que si l'élément qui précède `a.(p)` dans une telle sous-suite est `a.(i)`, alors la sous-suite correspondante s'obtient en ajoutant `a.(p)` à la plus longue sous-suite qui se termine par `a.(i)`. Il suffit donc de mémoriser i. On obtient alors, en déclarant un tableau `precedent` :

```
let afficher_sous_suite (a:int array) : unit =
  let n = Array.length a in
  let l = Array.make n 0 in
  let precedent = Array.make n 0 in
  l.(0) <- 1;
  for p=1 to n-1 do
    let max = ref 0 and prec = ref 0 in
    for j=0 to p-1 do
      if (a.(j) <= a.(p)) && (l.(j)+1 >= !max)
        then (
          max := l.(j) + 1;
          prec := j
        )
    done;
    l.(p) <- !max; precedent.(p) <- !prec
  done;
  (* Trouver le max *)
  let longueur = ref 1 and pmax = ref 0 in
  for i=1 to n-1 do
    if l.(i) > !longueur then (
      longueur := l.(i);
      pmax := i
    )
  done;
```

```
Printf.printf "%d" a.(!pmax);
for i=1 to !longueur-1 do
  pmax := precedent.(!pmax);
  Printf.printf " %d" a.(!pmax);
done;
```

Cette fonction affichera une plus longue sous-suite à l'envers. L'afficher à l'endroit n'est pas difficile.

⋆⋆⋆⋆⋆⋆⋆⋆

2.4 Recherche de motifs

Un mot u est une suite $(u_1, u_2, ..., u_n)$ de lettres, l'entier n est la longueur du mot u. La suite vide correspond au mot vide, noté ε, de longueur 0. La concaténation de deux mots $u = (u_1, u_2, ..., u_n)$ et $v = (v_1, v_2, ..., v_k)$, notée uv, est simplement le mot $(u_1, u_2, ..., u_n, v_1, v_2, ..., v_k)$. Il est facile de vérifier que la concaténation est une opération associative pour laquelle le mot vide est élément neutre à gauche et à droite. Le mot v est dit facteur ou motif d'un mot u s'il existe des mots g et d tels que $u = gvd$. Si le mot g (resp. d) est la suite vide, on dit que v est un préfixe (resp. suffixe) de u.

Question 1

Modéliser la situation décrite. Écrire un algorithme qui, étant donnés deux mots u et v, vérifie si v est un facteur de u. Quel est le nombre maximum de comparaisons effectuées par votre algorithme ?

Question 2

Si v n'est pas le mot vide, on note $Bord(v)$ le plus grand mot distinct de v qui soit à la fois préfixe et suffixe de v. Par exemple, si $v = abacaba$, $Bord(v) = aba$. Soit $v = (v_1, v_2, ..., v_k)$. Pour tout i entre 1 et k, on définit $B(i)$ comme étant la longueur de $Bord(v_1...v_k)$.

Utiliser cette fonction B pour écrire un nouvel algorithme vérifiant si un mot v est facteur d'un mot u. Quel est le nombre maximum de comparaisons effectuées par ce nouvel algorithme ?

Question 3

Soit $v = (v_1, v_2, ..., v_k)$ et a une lettre. On admet que $Bord(va)$ est le plus long préfixe de v qui soit dans $\{\varepsilon, Bord(v)a, Bord^2(v)a, ..., Bord^k(v)a\}$ (en posant $Bord(\varepsilon) = \varepsilon$). Déduire de ce résultat un algorithme qui, étant donné un mot $v = (v_1, v_2, ..., v_k)$, calcule successivement $B(1), ..., B(k)$.

CORRIGÉ

Question 1

La modélisation suggérée consiste à représenter les mots par des chaînes de caractères.

L'algorithme naïf, mais naturel, consiste à comparer v avec les facteurs de u de même longueur que v en commençant en posotion 0, puis 1, 2... jusqu'au succès ou à la fin de u.

```
let est_facteur (u:string) (v:string) : bool =
  let n = String.length u in
  let k = String.length v in

  try
    for i=0 to n-k do
      let sub = String.sub u i k in
      if String.equal sub v then
        raise Exit
    done;
    false
  with Exit -> true
```

Chaque appel à la fonction `String.equal` amène à au plus k comparaisons. Le nombre d'appels à `String.equal` étant d'au plus $(n-k+1)$, le nombre total de comparaisons effectuées par cet algorithme est au plus $k(n-k+1)$. Le maximum est atteint pour les mots de la forme $u = a^{n-1}b$ et $v = a^{k-1}b$.

Question 2

La primitive Caml `equal` compare les chaînes de caractère passées en argument octet par octet (caractère par caractère). On donne la code OCaml équivalent suivant :

```
let equal (s1:string) (s2:string) : bool =
  let n = String.length s1 in
  let k = String.length s2 in
  if n <> k then false else
  try
    for i=0 to n-1 do
      if s1.[i] <> s2.[i]
        then raise Exit
    done;
    true
  with Exit -> false
```

Supposons que l'appel à `String.equal` renvoie false lors de la comparaison entre u_{i+j-1} et v_j. On a alors la situation suivante :

$$\begin{array}{ccccc} u_i & u_{i+1} & ... & u_{i+j-2} & u_{i+j-1} \\ = & = & & = & \neq \\ v_1 & v_2 & ... & v_{j-1} & v_j \end{array}$$

On va donc ensuite comparer la chaîne de caractères $(u_{i+1}, u_{i+2}, ..., u_{i+j})$ à v. Pour que cet appel renvoie *true*, il faut en particulier que $v_1 = v_2$, ..., $v_{j-2} = v_{j-1}$. Ces égalités

signifient exactement que $Bord(v_1...v_{j-1}) = v_1...v_{j-2}$. Ainsi, si $Bord(v_1...v_{j-1}) \neq v_1...v_{j-2}$, il est inutile d'appeler `String.equal` avec la chaîne de caractères (u_{i+1}, u_{i+2}, ..., u_{i+j}). De façon identique, on voit que si $Bord(v_1...v_{j-1}) \neq v_1...v_{j-3}$, l'appel à `String.equal` avec la chaîne de caractères $(u_{i+2}, u_{i+3}, ..., u_{i+j+1})$ est inutile. En répétant le raisonnement, il apparaît que le premier appel qui ne soit pas voué à l'échec est celui où u_{i+j-1} est comparé à $v_{1+B(j-1)}$. Cette remarque conduit alors au nouvel algorithme suivant, où on suppose la fonction `b` correspondant à B définie :

```
let est_facteur_bis (u:string) (v:string) : bool =
  let n = String.length u in
  let k = String.length v in

  let i = ref 0 and j = ref 0 in
  while !i < n && !j < k do
    if u.[!i] = v.[!j] then (
      incr i; incr j
    ) else (
      if !j = 0 then incr i
      else i := b v (!j-1)
    )
  done;
  !j = k
```

Le nombre maximal de comparaisons peut être évalué comme suit. Appelons « positif » un test $u_i = v_j$ évalué à *true* et négatif un test $u_i = v_j$ évalué à $false$. Lors d'un test positif, i augmente, il y a donc au plus n tests positifs. Lors d'un test négatif, la quantité $i - j$ augmente : elle vaut 0 au commencement de l'algorithme et au plus n lorsque l'algorithme se termine. Ainsi, il y a également au plus n tests négatifs. En conclusion, ce second algorithme effectue donc au plus $2n$ comparaisons.

Question 3

On a bien sûr $B(1) = 0$ et $B(2) = 1$ si $v_1 = v_2$, 0 sinon.

Supposons calculés $B(1)$, ..., $B(h-1)$. D'après l'indication donnée dans l'énoncé, $B(h)$ est la longueur du plus long préfixe de $v_1...v_h$ qui soit dans

$$\{\varepsilon, Bord(v_1...v_{h-1})v_h, Bord^2(v_1...v_{h-1})v_h, ...\}$$

Il est clair que la longueur des mots $Bord^i(v_1...v_{h-1})v_h$ décroit (au sens large) lorsque i croit. Ainsi, il faut d'abord tester si $Bord(v_1...v_{h-1})v_h$ est un préfixe de $v_1...v_h$, puis si ce n'est pas le cas, si $Bord^2(v_1...v_{h-1})v_h$ est un préfixe de $v_1...v_h$, etc. En remarquant que $Bord(v_1...v_{h-1})v_h$ est un préfixe de $v_1...v_h$ si et seulement si $v_h = v_{1+B(h-1)}$, on est amenés naturellement à l'algorithme suivant, renvoyant le tableau $[B(1), B(2), ..., B(k)]$:

```
let calcul (v:string) : int array =
  let k = String.length v in
  let b = Array.make k 0 in

  for h=1 to k-1 do
    let j = ref b.(h-1) in
```

```
    let fini = ref false in
    while not !fini do
      (* v_{j+1} = v_h *)
      if v.[!j] = v.[h] then (
        fini := true;
        b.(h) <- !j+1
      )
      else if !j = 0 then (
        fini := true;
        b.(h) <- 0
      )
      else j := b.(!j-1)
    done;
  done;
  b
```

En considérant la quantité $h - j$, on obtient par un argument similaire à celui de la question 2 que le nombre de comparaisons effectuées est au plus $2k$.

L'indication de l'énoncé : $Bord(va)$ est le plus long préfixe de v qui soit dans $\{\varepsilon, Bord(v)a, Bord^2(v)a, ..., Bord^k(v)a\}$ se démontre comme suit.
On dit que w est un bord de v si w est à la fois un préfixe et un suffixe de v. Soit z un bord de va. Si $z \neq \varepsilon$, $z = z'a$, où z' est un bord de v. Ou bien $z' = Bord(v)$, ou bien z' est un bord de $Bord(v)$. Par induction, il vient immédiatement que z' est de la forme $Bord^i(v)$ pour un certain i tel que $1 \leq i \leq k$. Ainsi, z est dans l'ensemble $\{\varepsilon, Bord(v)a, Bord^2(v)a, ..., Bord^k(v)a\}$.
Réciproquement, tout mot de cet ensemble est clairement suffixe de va, donc un bord de va s'il est préfixe de va.

⋆⋆⋆⋆⋆⋆⋆⋆

2.5 Mots bien parenthésés

On considère une parenthèse ouvrante « (» et une parenthèse fermante «) ». Un mot parenthésé u est une suite de parenthèses ouvrantes et fermantes. Le nombre de parenthèses utilisées est appelée la longueur du mot. La suite vide correspond au mot vide, noté ε. La concaténation de deux mots parenthésés u et v, notée uv, est simplement le mot parenthésé obtenu en mettant u et v bout à bout. Ainsi, si $u = (((()$ et $v =))$, $uv = ((()()))$. Un mot parenthésé v est dit facteur gauche d'un mot parenthésé u s'il existe un mot w tel que $u = vw$. Un mot parenthésé u est bien parenthésé s'il contient autant de parenthèses ouvrantes que de parenthèses fermantes et si tout facteur gauche v de u contient au moins autant de parenthèses ouvrantes que de parenthèses fermantes.

Question 1

Modéliser la situation décrite et écrire un algorithme qui, étant donné un mot parenthésé, vérifie s'il est bien parenthésé ou non.

Question 2

Montrer que, étant donné un mot bien parenthésé non vide u, il existe un unique couple (v, w) de mots bien parenthésés tels que $u = (v)w$. Écrire un algorithme qui, étant donné le mot u, calcule les mots v et w.

Question 3

Soit $N \in \mathbb{N}$. Écrire un algorithme qui énumère tous les mots bien parenthésés de longueur au plus N.

CORRIGÉ

Question 1

Une solution est de modéliser un mot parenthésé par une chaîne de caractères ne contenant que des parenthèses.

Pour tester si un mot u est bien parenthésé, il suffit de vérifier que u contient autant d'ouvrantes que de fermantes et que tout facteur gauche de u contient plus de parenthèses ouvrantes que de fermantes. On peut pour cela gérer un compteur que l'on incrémente (resp. décrémente) lorsque l'on trouve une parenthèse ouvrante (resp. fermante). On suppose ici que u est bien formé (i.e ne contient que des parenthèses).

```
let test (u:string) : bool =
  let n = String.length u in
  let cpt = ref 0 in
  try
    for i=0 to n-1 do
      if u.[i] = '(' then incr cpt
      else decr cpt;   (* u.[i] = ')' *)

      if !cpt < 0 then raise Exit
    done;
    !cpt = 0
  with Exit -> false
```

Question 2

Montrons tout d'abord l'unicité de la décomposition proposée. Supposons qu'un mot u se décompose en $u = (v)w = (v')w$ avec $v \neq v'$. Il est clair que si v et v' sont de même longueur alors $v = v'$. Par symétrie, il suffit de traiter le cas où v' est de longueur strictement plus grande que v. Ainsi $v' = v)v''$ où v'' est un mot parenthésé éventuellement vide. Comme v est bien parenthésé, il contient autant d'ouvrantes que de fermantes. Ainsi, le préfixe $v)$ de v' contient une fermante de plus que d'ouvrantes, ce qui est en contradiction avec le fait que v' soit bien parenthésé. On a ainsi montré l'unicité de la décomposition.

Soit u un mot bien parenthésé. Pour montrer l'existence d'une décomposition, considérons le plus petit facteur gauche non vide u' de u qui contienne autant d'ouvrantes que

de fermantes (u' existe puisque u a cette propriété). Dans la fonction de la question précédente, u' correspond au plus petit facteur gauche non vide pour lequel la variable `cpt` prend la valeur 0. Ainsi, il est facile de voir que u' commence par une ouvrante et finit par une fermante : $u' = (v)$. En utilisant le fait que u est bien parenthésé, on vérifie aisément que v et w, où w est défini par $u = u'w$, sont bien parenthésés.

Le calcul de v et w repose sur le remarque que (v) est le plus petit facteur de u qui contienne autant d'ouvrantes que de fermantes.

```
let decompose (u:string) : (string * string) =
  let n = String.length u in
  let cpt = ref 0 and idx = ref 0 in

  for i=0 to n-1 do
    if u.[i] = '(' then incr cpt
    else decr cpt;

    if !cpt = 0 && !idx = 0 then (
      idx := i
    )
  done;

  let v = String.sub u 1 (!idx-1) in
  let w = String.sub u (!idx+1) (n - !idx - 1) in
  (v, w)
```

Question 3

L'idée est bien entendu d'utiliser la question précédente. Si l'on connaît tous les mots bien parenthésés de longueur au plus L, où L est une certaine constante, on obtient un mot bien parenthésé de longueur $L + 2$ en prenant un mot bien parenthésé u de longueur k, un mot bien parenthésé de longueur $L - k$ et en construisant le mot $(u)v$. Il est donc nécessaire de garder en mémoire tous les mots parenthésés déjà construits.

Nous allons ici utiliser un tableau de listes tel que la liste à l'indice i contient tous les mots bien parenthésés de longueur $2i$.

```
let enumere_mots (lmax:int) : string list array =
  let mots = Array.make (lmax+1) [] in
  mots.(0) <- [""];
  for l=0 to lmax-1 do
    for k=0 to l do
      List.iter (fun u ->
        List.iter (fun v ->
          let mot = Printf.sprintf "(%s)%s" u v in
          mots.(l+1) <- mot::mots.(l+1)
        ) mots.(l-k)
      ) mots.(k)
    done
  done; mots
```

Remarque : Le nombre de mots bien parenthésés de longueur $2n$ est égal à $\frac{1}{n+1}\binom{2n}{n}$, le n-ième nombre de Catalan.

⋆⋆⋆⋆⋆⋆⋆⋆

2.6 Plus longue sous-suite commune

Notation : Si X est une chaîne de caractères, on désignera par X_l le préfixe de X de longueur l.

Une sous-suite du mot $A = a_1a_2...a_n$ est un mot obtenu en supprimant certaines lettres. Par exemple, *bbcdd* est une sous-suite de *aaabbbcccddd*. Une sous-suite commune à deux chaînes de caractères A et B et de longueur maximale est appelée *PLSC* (Plus Longue Sous-suite Commune) de A et B. Par exemple, si $A = abaab$ et $B = aabb$, on a deux *PLSC*, *aab* et *abb*.

Question 1

Étant donné deux mots $A = a_1a_2...a_m$ et $B = b_1b_2...b_n$ de longueurs respectives m et n, on demande de calculer la longueur d'une *PLSC* de A et B :

Justifier l'équation récurrente (à compléter) :

$$long(i,j) = \max(long(i-1,j-1) + [a_i = b_j],\ long(i,j-1),\ long(i-1,j))$$

où $[a_i = b_j]$ vaut 1 si $a_i = b_j$ et 0 sinon et où $long(i,j)$ désigne la longueur d'une *PLSC* de A_i et B_j.

Écrire une fonction OCaml pour calculer la longueur d'une *PLSC* de A et B.

Question 2

On s'intéresse maintenant au calcul effectif d'une *PLSC* de A et B. Écrire une fonction OCaml qui effectue ce calcul. Comment la modifier pour obtenir toutes les *PLSC* de A et B ?

CORRIGÉ

Question 1

Soit $A = a_1a_2...a_i$ et $B = b_1b_2...b_j$. Notons $Z = z_1...z_k$ une de leurs *PLSC* :

- Si $a_i = b_j$ alors $z_k = a_i = b_j$ et Z_{k-1} est une *PLSC* de A_{i-1} et B_{j-1} ;
- Si $a_m \neq b_n$ alors : $(z_k \neq a_m) \implies Z$ est une *PLSC* de A_{m-1} et B ;
- Si $a_m \neq b_n$ alors : $(z_k \neq b_n) \implies Z$ est une *PLSC* de A et B_{n-1}.

d'où la relation complète :

$$long(i,j) = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0 \\ long(i-1,j-1) + 1 & \text{si } i,j > 0 \text{ et } a_i = b_j \\ \max(long(i,j-1),\ long(i-1,j)) & \text{si } i,j > 0 \text{ et } a_i \neq b_j \end{cases}$$

Pour écrire la fonction, il suffit de générer toutes les valeurs de $long(i, j)$ dans un ordre compatible avec la relation de récurrence. On écrit donc un algorithme de *programmation dynamique* qui remplit un tableau ligne par ligne. On a $long(i, j) =$ `long.(i).(j)`.

```
let longueur (a:string) (b:string) : int =
  let n = String.length a in
  let m = String.length b in
  let long = Array.make_matrix (n+1) (m+1) 0 in

  for i=1 to n do
    for j=1 to m do
      if a.[i-1] = b.[j-1] then
        long.(i).(j) <- long.(i-1).(j-1) + 1
      else
        long.(i).(j) <- max long.(i).(j-1) long.(i-1).(j)
    done
  done;
  long.(n).(m)
```

La complexité est en $O(nm)$.

Question 2

Il suffit de créer une matrice `chemin` que l'on modifie à chaque fois qu'on met à jour `long.(i).(j)`. Celle-ci permettra de déterminer « d'où on vient » et donc de reconstruire une ou toutes les *PLSC*. La fonction devient :

```
let plsc_chemin (a:string) (b:string) : string array array =
  let n = String.length a in
  let m = String.length b in
  let long = Array.make_matrix (n+1) (m+1) 0 in
  let chemin = Array.make_matrix (n+1) (m+1) "" in

  for i=1 to n do
    for j=1 to m do
      if a.[i-1] = b.[j-1] then (
        long.(i).(j) <- long.(i-1).(j-1) + 1;
        chemin.(i).(j) <- " ↖ "
      )
      else if long.(i-1).(j) >= long.(i).(j-1) then (
        long.(i).(j) <- long.(i-1).(j);
        chemin.(i).(j) <- " ↑ "
      )
      else (
        long.(i).(j) <- long.(i).(j-1);
        chemin.(i).(j) <- " ← "
      )
    done
  done;
  chemin
```

Grâce au tableau chemin, on peut donc retrouver une *PLSC* de A et B :

```
let plsc (a:string) (b:string) : unit =
  let n = String.length a in
  let m = String.length b in
  let chemin = plsc_chemin a b in

  let rec aux i j =
    if i=0 || j=0 then ()

    else if chemin.(i).(j) = " ↖ " then (
      aux (i-1) (j-1);
      print_char a.[i-1]
    )
    else if chemin.(i).(j) = " ↑ " then
      aux (i-1) j
    else
      aux i (j-1)
  in aux n m;
  print_newline ()
```

Comme le montre la figure ci-dessous représentant les matrices `longueur` et `chemin` pour $A = abcbdab$ et $B = bdcaba$, on suit les flèches (chemin grisé) et on note les lettres correspondant aux flèches obliques :

	j	0	1	2	3	4	5	6
i		b_j	b	d	c	a	b	a
0	a_i	0	0	0	0	0	0	0
1	a	0	0 ↑	0 ↑	0 ↑	1 ↖	1 ←	1 ↖
2	b	0	1 ↖	1 ←	1 ←	1 ↑	2 ↖	2 ←
3	c	0	1 ↑	1 ↑	2 ↖	2 ←	2 ↑	2 ↑
4	b	0	1 ↖	1 ↑	2 ↑	2 ↑	3 ↖	3 ←
5	d	0	1 ↑	2 ↖	2 ↑	2 ↑	3 ↑	3 ↑
6	a	0	1 ↑	2 ↑	2 ↑	3 ↖	3 ↑	4 ↖
7	b	0	1 ↖	2 ↑	2 ↑	3 ↑	4 ↖	4 ↑

Cette fonction a une complexité en $O(n+m)$ car i ou j décroît à chaque itération.

Pour afficher toutes les *PLSC*, il faudrait traiter les cas où $long(i-1,j) = long(i,j-1)$, c'est-à-dire celui où le chemin montant et celui allant à gauche permettent tous les deux d'obtenir une *PLSC* et afficher les deux chaînes correspondantes à chaque fois.

⋆⋆⋆⋆⋆⋆⋆⋆