▶ **MP & MP* – Option Informatique**
Année 2002–2003, Septième TP Caml
Vincent Simonet (`http://cristal.inria.fr/~simonet/`)

Recherche de motifs

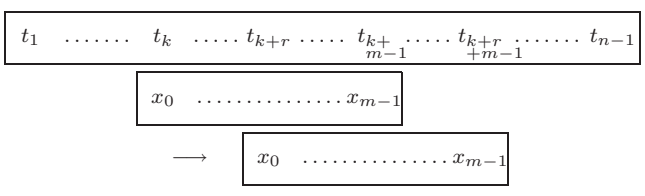
On s'intéresse au problème de la recherche de motifs à une dimension : étant donné un texte (c'est-à-dire une suite de lettres) on souhaite trouver toutes les occurences d'un motif (i.e. une autre suite de lettres) dans celui-ci. Dans ce sujet, nous allons étudier plusieurs algorithme résolvant ce problème.

Soit un alphabet Σ constitué de q lettres ($q \geq 2$). On considère deux mots sur Σ : le premier sera noté $t = t_0 \dots t_{n-1}$ et appelé *texte*, le second mot sera noté $x = x_0 \dots x_{m-1}$ et appelé *motif*. L'objectif est de trouver des algorithmes permettant de savoir si le motif x est présent dans le texte t (i.e. x est un facteur de t) et dans ce cas de déterminer la position des différentes apparitions de x dans t (i.e. trouver tous les entiers k tels que $t_k \dots t_{k+m-1} = x_0 \dots x_{m-1}$). Dans toute la suite de cet énoncé, nous conservons ces notations sans les redéfinir systématiquement.

Pour la programmation en Caml, on supposera que l'alphabet est formé de l'ensemble des valeurs de type `char`. On rappelle que ces caractères sont numérotés de 0 à 255, et on note a_i celui qui porte le numéro i. La fonction `int_of_char` de la bibliothèque standard permet d'obtenir le numéro d'un caractère. Les mots t et x seront stockés dans des objets de type `string`.

Pour les calculs de complexité, l'opération élémentaire choisie est la comparaison entre les lettres.

Les algorithmes seront du type suivant : on compare x avec un facteur $t_k \dots t_{k+m-1}$ de t, puis on décale x vers la droite de t d'un certain nombre de lettres et on compare de nouveau x avec un autre facteur de t de la forme $t_{k+r} \dots t_{k+r+m-1}$, avec $r \geq 1$. Le problème essentiel est le choix du décalage, afin de minimiser le nombre de comparaisons de lettres. On peut représenter schématiquement ces comparaisons de la manière suivante :



1 Méthode naïve

La méthode la plus immédiate consiste à systématiquement décaler x d'une lettre, c'est-à-dire à toujours prendre $r = 1$.

▶ **Question 1** *Écrivez une fonction* `compare_substrings` *prenant pour argument un mot w, un entier k, un mot w', un entier k' et un entier s. Cette fonction retournera un booléen indiquant si $w_k \cdots w_{k+s-1} = w'_{k'} \cdots w'_{k'+s-1}$. (Vous pourrez supposer que les entiers k, k' et s désignent bien des sous-mots valides de w et w'.)*

```
value compare_substrings: string -> int -> string -> int -> int -> bool
```

▶ **Question 2** *Déduisez-en une fonction* `occurences` *qui retourne la liste des occurences d'un mot x dans un texte t.*

```
value occurences: string -> string -> int list
```

▶ **Question 3** *Vérifiez le résultat retourné par votre fonction sur l'exemple* `x = "abadababa"` *et* `t = "abacabadabaabadabababadababaa"`.

▶ **Question 4** *Quelle est, en fonction de n et m, la complexité au pire de cette fonction ?*

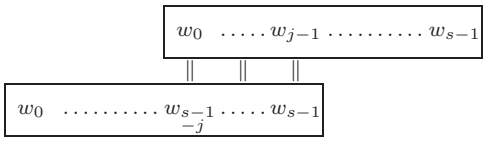
2 Algorithme de Knuth-Morris-Pratt

Dans cette section, nous étudions un algorithme qui fut inventé indépendamment par Knuth et Pratt et par Morris ; leurs travaux furent publiés conjointement en 1977. Cet algorithme a un temps d'exécution en $O(n + m)$ dans le pire des cas.

© Vincent Simonet 2002
Ce document est distribué selon les termes de la GNU Free Documentation License.

2.1 Bords d'un mot

Soit $w = w_0 \dots w_{s-1}$ un mot non vide. On appelle *bord* de w le plus long préfixe strict de w qui soit également un suffixe de w. Le bord de w pouvant éventuellement être le mot vide ε, il a donc une longueur j comprise entre 0 et $s-1$:



► **Question 5** *Écrivez une fonction* border_length *qui calcule la longueur du bord d'un mot.*

value border_length: string −> int

En fait, pour implémenter l'algorithme de Knuth-Morris-Pratt, nous aurons besoin de connaître la longueur du bord de chaque préfixe de x, c'est à dire de tous les mots de la forme $x_0 \dots x_{s-1}$. Nous allons maintenant proposer une méthode permettant d'effectuer ce calcul de manière efficace, en utilisant une idée analogue à celles que nous avons rencontrées dans le sujet « *Programmation dynamique* ».
Pour j compris entre 1 et m, notons $\beta(j)$ la longueur du bord de $x_0 \dots x_{j-1}$ (i.e. $\text{bord}(x_0 \cdots x_{j-1}) = x_0 \cdots x_{\beta(j)-1}$). Par convention, on pose $\beta(0) = -1$. Soit a une lettre. Pour j compris entre 0 et m, on définit $\beta(j)_a$ comme la longueur du bord du mot $x_0 \dots x_{j-1}a$.

► **Question 6** *Soit $j \geq 1$. Montrez que si $a = x_{\beta(j)}$ alors on a $\text{bord}(x_0 \cdots x_{j-1}a) = \text{bord}(x_0 \cdots x_{j-1})a$; et, sinon, $\text{bord}(x_0 \cdots x_{j-1}a) = \text{bord}(x_0 \cdots x_{\beta(j)-1}a)$.*

On déduit de ce résultat la relation de récurrence suivante sur β, pour $j \geq 1$:

$$\beta(j)_a = \begin{cases} \beta(j)+1 & \text{si } x_{\beta(j)+1} = a \\ \beta(\beta(j))_a & \text{si } x_{\beta(j)+1} \neq a \text{ et } \beta(j) \geq 0 \end{cases}$$

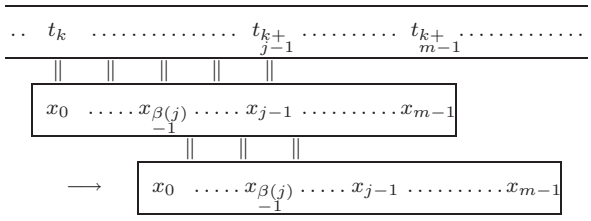
► **Question 7** *En utilisant le fait que $\beta(j+1) = \beta(j)_{x_j}$, écrivez une fonction* borders *prenant pour argument le mot* x *et retournant un tableau* beta *contenant les valeurs de $\beta(j)$ pour j allant de 0 à m.*

value borders: string −> int vect

2.2 Application à l'algorithme KMP

Le principe de l'algorithme de Knuth, Morris et Pratt est d'utiliser la comparaison de x avec $t_k \dots t_{k+m-1}$ pour déterminer le décalage à réaliser avant d'effectuer la comparaison suivante :
- Dans le cas où $t_k \neq x_0$, on décale simplement x d'une lettre vers la droite.
- Sinon, dans le cas où $t_k = x_0$, on considère j maximal tel que $t_k \dots t_{k+j-1} = x_0 \dots x_{j-1}$. On décale alors x de $j - \beta(j)$ lettres vers la droite et on commence la comparaison de x avec ce nouveau facteur de t à la lettre $x_{\beta(j)}$.

On peut représenter le décalage effectué dans le deuxième cas comme suit :



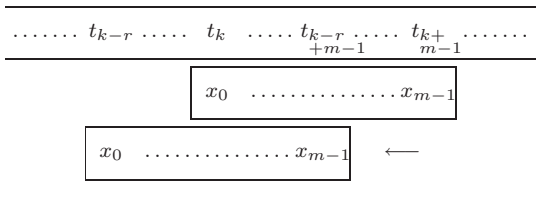
► **Question 8** *Justifiez la validité de cette méthode, i.e. montrez qu'en faisant ce décalage, on n'a pas « oublié » une apparition de x dans t.*

► **Question 9** *En utilisant la fonction* borders*, définissez une fonction* kmp *qui implémente l'algorithme de Knuth, Morris et Pratt. (Cette fonction prendra les mêmes arguments et retournera le même résultat que la fonction* occurences*.)*

value kmp: string −> string −> int list

3 Algorithme de Boyer-Moore

Si le motif x est relativement long et si l'alphabet Σ n'est pas un ensemble trop grand, l'algorithme dû à Robert S. Boyer et J. Strother Moore (1977) sera sans doute le plus efficace des algorithmes de recherche de motif. Cet algorithme intègre deux heuristiques qui lui permettent en pratique d'éviter une grande partie du travail effectué par les algorithmes de recherche de motif précédents. Dans la présentation de l'algorithme de Boyer-Moore que nous allons donner, on commencera les comparaisons de x avec des facteurs de t en commençant par la fin du texte, chaque comparaison étant quant-à-elle toujours effectuée de la gauche vers la droite :

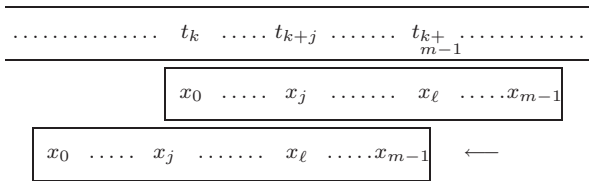


On peut considérer que les deux heuristiques agissent indépendamment et en parallèle : lorsque deux caractères ne correspondent plus, chacune propose un décalage qui ne fasse pas manquer d'occurence de x dans t, et l'algorithme choisit à chaque fois la meilleure des deux.

Supposons pour la suite que l'on compare x avec le facteur $t_k \cdots t_{k+m-1}$ de t, et notons j la longueur de la correspondance entre x et t à cette position, c'est-à-dire le plus grand entier tel que $x_0 \cdots x_{j-1} = t_k \cdots t_{k+j-1}$.

3.1 L'heuristique du mauvais caractère

L'heuristique du mauvais caractère n'agit que lorsque la comparaison de x avec le facteur de t échoue, c'est-à-dire quand $j < m$. Dans ce cas, on a $x_j \neq t_{k+j}$ et on recherche la première occurence de la lettre t_{k+j} dans x : soit ℓ le plus petit entier tel que $x_\ell = t_{k+j}$ (par convention, on pose $\ell = m$ si t_{k+j} n'apparaît pas dans x). L'heuristique du mauvais caractère propose alors de décaler x de $\ell - j$ caractères vers la gauche, de manière à faire coïncider x_ℓ avec t_{k+j}.



Notons que dans le cas où la première occurence de t_{k+j} dans x est à gauche de x_j (i.e. $\ell < j$) alors cette heuristique ne propose pas de décalage acceptable, puisque $\ell - j < 0$.
Pour appliquer cette heuristique, nous avons besoin de pré-calculer, pour chaque lettre de l'alphabet Σ, la position de sa première occurence dans x.

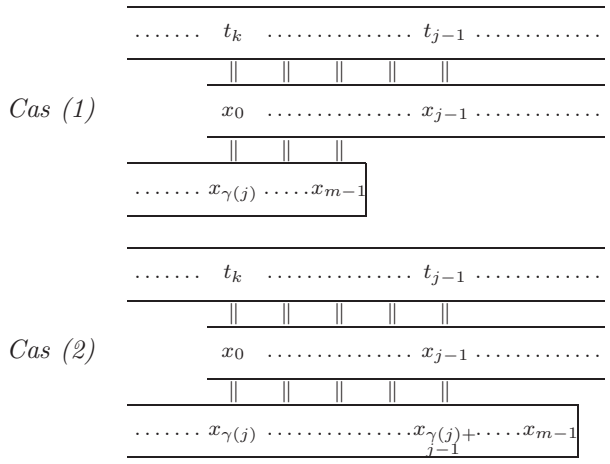
► **Question 10** *Écrivez une fonction* first_occurence *qui calcule la position de la première occurence de chaque lettre dans un mot x. Son résultat sera un tableau* o *dont la case j contiendra la position de la première occurence de a_j dans* x *(ou bien* m*, la longueur de* x*, si a_j n'apparaît pas dans* x*).*

value first_occurence: char vect −> int vect

3.2 L'heuristique du bon préfixe

L'heuristique du bon préfixe déplace le motif vers la gauche de la plus petite quantité qui préserve l'appariement des caractères du « bon préfixe » $x_0 \cdots x_{j-1}$ dans le texte avec des caractères du motif. Pour cela, l'heuristique recherche le plus petit entier $\gamma(j) \geq 1$ tel que (1) $x_{\gamma(j)} \cdots x_{m-1}$ soit un préfixe de $x_0 \cdots x_{j-1}$ ou bien (2) $x_0 \cdots x_{j-1}$ soit un préfixe de $x_{\gamma(j)} \cdots x_{m-1}$; et propose un décalage de $\gamma(j)$ caractères vers la gauche.

On peut représenter ces deux cas par les schémas respectifs suivants :



Pour appliquer cette heuristique, il faut connaître la valeur de $\gamma(j)$ pour chaque suffixe $x_0 \cdots x_{j-1}$ de x. Pour calculer efficacement les valeurs de la fonction γ, on peut remarquer que :
- Dans le cas (1), $x_{\gamma(j)} \cdots x_{m-1}$ est le bord de x, i.e. $\gamma(j) = m - \beta(m)$.
- Dans le cas (2), $x_0 \cdots x_{j-1}$ est le bord de $x_0 \cdots x_{\gamma(j)+j-1}$, i.e. $\beta(\gamma(j) + j) = j$.

► **Question 11** *En utilisant la fonction* borders*, écrivez une fonction* good_suffix *prenant pour argument un mot x et retournant le tableau des valeurs de $\gamma(j)$ (pour j compris entre 0 et m) correspondant.*

value good_suffix: 'a vect −> int vect

3.3 Application à l'algorithme BM

► **Question 12** *En utilisant les fonctions* first_occurence *et* good_suffix*, définissez une fonction* bm *qui implémente l'algorithme de Boyer et Moore. (Cette fonction prendra les mêmes arguments et retournera le même résultat que la fonction* occurences*.)*

value bm: 'a vect −> 'a vect −> int list

Il existe de nombreux autres algorithmes effectuant la recherche d'un motif dans un texte. Une première méthode classique consiste à construire un automate à partir du mot x puis à lire le texte t à l'aide de cet automate. Ce dernier est construit de telle sorte qu'à chaque occurence de x dans le texte, il passe par un état particulier. La lecture du texte s'effectue en temps linéaire, mais pas la construction de l'automate. En fait, on peut voir l'algorithme de Knuth-Morris-Pratt comme un raffinement de cette méthode où le pré-calcul sur x est lui aussi effectué en temps linéaire.
Rabin et Karp ont proposé en 1981 un algorithme basé sur des notions de théorie des nombres qui s'étend au cas de la dimension 2. Enfin l'algorithme de Galil et Seiferas (1983) effectue la recherche de motif en temps linéaire, comme l'algorithme de Knuth-Morris-Pratt, mais n'utilise qu'un espace de stockage de taille constante ($O(1)$) en plus de celui requis pour stocker le motif et le texte.

▶ **MP & MP* – Option Informatique**
Année 2002–2003, Septième TP Caml
Vincent Simonet (`http://cristal.inria.fr/~simonet/`)

Recherche de motifs

Un corrigé

▶ **Question 1** Il suffit de comparer les caractères de w et w' deux à deux.

```
let compare_substrings w k w' k' s =
  let rec aux i =
    if i = s then true
    else w.[k + i] = w'.[k' + i] && aux (i + 1)
  in
  aux 0
;;
```

Dans cette implémentation, l'itération est codée à l'aide d'une fonction récursive (aux). On peut aussi l'écrire dans un style impératif à l'aide d'une boucle **while** ou **for**.

▶ **Question 2** Notre fonction occurences teste toutes les positions possibles de x à l'aide d'une boucle **for**. Celles qui correspondent à des occurences de x dans t sont alors stockées dans la références result.

```
let occurences x t =
  let m = string_length x
  and n = string_length t in
  let result = ref [] in
  for k = 0 to n − m do
    if compare_substrings t k x 0 m
    then result := k :: !result
  done;
  !result
;;
```

▶ **Question 5** Une méthode « naïve » pour déterminer le bord de w consiste à tester tous ses préfixes stricts, en commençant par le plus long, jusqu'à en trouver un qui soit également un suffixe de w.

```
let border_length w =
  let s = string_length w in
  let rec aux i =
    if i = 0 || compare_substrings w 0 w (s − i) i
    then i else aux (i − 1)
  in
  aux (s − 1)
;;
```

▶ **Question 7** On utilise une technique proche de la « *programmation dynamique* » : les résultats intermédiaires sont stockées dans un tableau (beta). Les calculs sont effectués grâce à une fonction récursive auxilliaire : un appel aux j a retourne la valeur de $\beta(j)_a$.

```
let borders x =
  let m = string_length x in
  let beta = make_vect (m+1) (−1) in
  let rec aux j a =
    if j = 0 then 0
    else if x.[beta.(j)] = a then beta.(j) + 1
    else aux beta.(j) a
    in
  for j = 1 to m do
    beta.(j) <− aux (j−1) x.[j−1]
  done;
  beta
;;
```

▶ **Question 9**

```
let kmp x t =
  let beta = borders x in
  let m = string_length x in
  let n = string_length t in
  let k = ref 0 in
  let j = ref 0 in
  let result = ref [] in

  while !k <= n − m do
    while !j < m && x.[!j] = t.[!k + !j] do j := !j + 1 done;
    if !j = m then result := !k :: !result;
    k := !k + !j − beta.(!j);
    j := max 0 beta.(!j);
  done;

  !result
;;
```

▶ **Question 10** Pour calculer la première occurence de chaque lettre dans x, il suffit de parcourir ce mot de droite à gauche : pour chaque lettre rencontrée, on met à jour le tableau des occurences o avec la position courante dans x (j). Le mot étant parcouru dans le sens des positions décroissantes, on est ainsi assuré d'obtenir la première occurence de chaque caractère.

```
let first_occurence x =
  let o = make_vect 256 0 in
  for j = string_length x − 1 downto 0 do
    o.(int_of_char x.[j]) <− j
  done;
  o
;;
```

► **Question 11**

```
let good_suffix x =
  let m = string_length x in
  let beta = borders x in
  let gamma = make_vect (m+1) (m − beta.(m)) in
  for i = m downto 1 do
    if gamma.(beta.(i)) > i − beta.(i)
    then gamma.(beta.(i)) <− i − beta.(i)
  done;
  gamma
;;
```

► **Question 12** On commence par écrire une fonction match_length telle que match_length x t k retourne la valeur de j définie dans l'énoncée (i.e. la longueur de correspondance entre x et t à la position k).

```
let match_length x t k =
  let m = string_length x in
  let rec aux j =
    if j = m or x.[j] <> t.[k+j]
    then j
    else aux (j + 1)
  in
  aux 0
;;
```

On peut ensuite écrire la fonction implémentant l'algorithme de Boyer-Moore. Pour chaque décalage, on distingue le cas j = m où seul la seconde heuristique s'applique du cas général où les deux heuristiques sont valides.

```
let bm x t =
  let n = string_length t in
  let m = string_length x in
  let first = first_occurence x in
  let gamma = good_suffix x in

  let k = ref (n − m) in
  let result = ref [] in

  while !k >= 0 do
    let j = match_length x t !k in
    if j = m then begin
      result := !k :: !result;
      k := !k − gamma.(m)
    end
    else k := !k − (max gamma.(j) (first.(int_of_char t.[!k+j]) − j))
  done;

  !result
;;
```