Informatique *Cours*
Algorithmique des textes Ipesup

Objectifs

À l'issue de cette section, l'étudiant devra être capable de :
— ok

1 Recherche dans un texte

Dans cette section, on s'intéresse au problème de la recherche des occurrences d'une chaîne de caractères, que l'on appellera *motif*, dans une autre chaîne de caractères, que l'on appellera *texte*. Plus précisément, on va chercher à quelles positions dans le texte le motif apparaît.

Exemple

Il y a deux occurrences du motif "bra" dans le texte "abracadabra". En numérotant les positions à partir de 0, on a donc une occurrence de "bra" à la position 1 (le deuxième caractère du texte) et une autre à la position 8 (le neuvième caractère du texte).

Notre objectif est d'écrire une fonction qui affiche les positions de toutes les occurrences du motif dans le texte, sous la forme suivante,

```
occurrence à la position 1
occurrence à la position 8
```

et renvoie leur nombre total au final.
Bien entendu, on pourrait envisager plutôt de renvoyer la position de la première occurrence, le cas échéant, ou encore simplement de signaler la présence du motif avec un booléen.

Dans la suite, on note m le motif que l'on recherche et t le texte dans lequel on le recherche. On note M la longueur du motif et N la longueur du texte.

Une première remarque évidente est qu'il ne peut y avoir une occurrence de m dans t que si $M \leq N$.

Plus précisément, une occurrence de m dans t à la position i est contrainte par l'inégalité $0 \leq i \leq N - M$. En particulier, la chaîne vide "", qui a une longueur $M = 0$, apparaît à toutes les positions dans le texte t, pour $N + 1$ occurrences au total.

Il est utile de se représenter une occurrence de m dans t à la position i comme ceci :

Le programme suivante contient le code C d'une solution simple, mais assez naïve, à notre problème.

Recherche dans un texte naïvement

```c
int naive_string_search(char *m, char *t) {
  int lm = strlen(m), lt = strlen(t), count = 0;
  for (int i = 0; i <= lt - lm; i++)
    if (strncmp(m, t + i, lm) == 0) {
      printf("occurrence  la position %d\n", i);
      count++;
    }
  return count;
}
```

Ce programme considère successivement toutes les positions possibles pour une occurrence, c'est-à-dire tous les entiers entre 0 et $N - M$.
Pour une position i donnée, on utilise la fonction de bibliothèque `strncmp` pour déterminer si le motif m apparaît à la position i dans t. La fonction `strncmp` compare deux chaînes de caractères, pour l'ordre lexicographique, dans la limite d'un nombre de caractères donné, ici M. Elle renvoie -1 ou 1 si elle observe une différence entre les deux chaînes, à savoir -1 si la première est plus petite et 1 si elle est plus grande, et 0 si les deux chaînes contiennent au moins M caractères chacune, deux à deux égaux.

Le programme en reste assez naïf : dans le pire des cas, en effet, on peut être amené à systématiquement comparer tous les caractères du motif à chaque position, pour un total de $M(N - M + 1)$ comparaisons. C'est le cas si on recherche un motif formé uniquement de caractères a dans un texte également formé uniquement de caractères a.

Il existe cependant des moyens plus efficaces pour rechercher un motif dans un texte.

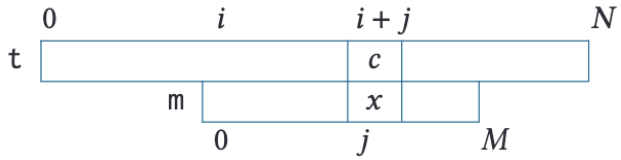
1.1 Algorithme de Boyer–Moore

L'algorithme de Boyer–Moore utilise un prétraitement du motif m à chercher dans un texte t pour accélérer la recherche. Le principe de l'algorithme est le suivant.

- On va tester l'occurrence du motif m dans le texte t à des positions i de plus en plus grandes, en partant de $i = 0$. Cela n'est pas différent pour l'instant de ce que fait le programme d'avant.
- Pour une position i donnée, on va comparer les caractères de m et de t de la droite vers la gauche, c'est-à-dire en comparant d'abord $m[M-1]$ et $t[i+M-1]$, puis $m[M-2]$ et $t[i+M-2]$, etc. Il s'agit là du sens inverse de celui utilisé avant. Le changement peut paraître anecdotique, mais en pratique il permet d'avancer plus vite dans certains cas.
- Si tous les caractères coïncident, on a trouvé une occurrence. Sinon, soit j l'indice de la première différence, c'est-à-dire le plus grand entier tel que $0 \leq j < M$ et $m[j] \neq t[i+j]$. Appelons c le caractère $t[i+j]$.

L'idée de l'algorithme de Boyer–Moore consiste à augmenter alors la valeur de i de

- la grandeur $j - k$ où k est le plus grand entier tel que $0 \leq k < j$ et $m[k] = c$, si un tel k existe (de manière à amener un caractère c de m sous le caractère $t[i+j]$),

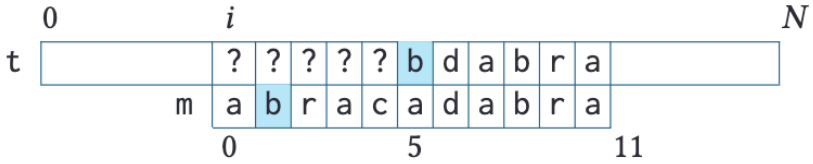


— la grandeur $j + 1$ sinon.

Plutôt que de rechercher un tel k à chaque fois, on peut précalculer une *table de décalages* contenant, à la case indexée par l'entier j et le caractère c, le plus grand entier k tel que $0 \leq k < j$ et $m[k] = c$ s'il existe, et rien sinon.

Exemple

Supposons que l'on recherche le motif m = "abracadabra". On est en train de tester l'occurrence de ce motif à une certaine position i dans le texte. On procède de droite à gauche, en commençant par la fin du motif. Supposons que les cinq premières comparaisons de caractères ont été positives, c'est-à-dire que les cinq dernières lettres du motif (dabra) coïncident avec les caractères « en face » dans le texte. Supposons enfin que le caractère suivant dans le texte ne coïncide pas, car il s'agit du caractère b alors que le motif a un caractère a à cette position. La situation s'illustre donc ainsi :



Pour une position j, avec $0 \leq j < M$, et un caractère c, la table donne le plus grand entier k tel que $0 \leq k < j$ et $m[k] = c$, s'il existe, et rien sinon.

	a	b	r	c	d
0					
1	0				
2	0	1			
3	0	1	2		
4	3	1	2		
5	3	1	2	4	
6	5	1	2	4	
7	5	1	2	4	6
8	7	1	2	4	6
9	7	8	2	4	6
10	7	8	9	4	6

On consulte alors la table de décalages, pour l'indice $j = 5$ (la position dans le motif) et pour le caractère b (le caractère du texte). La table indique la valeur 1, ce qui veut dire qu'il faut décaler le motif de $j - 1 = 4$ positions vers la droite. Cela a pour effet d'amener le caractère b en deuxième position dans le motif sous le caractère b du texte. Si en revanche le caractère du texte avait été z, alors la table n'aurait pas contenu d'entrée pour ce caractère, car il n'y a pas d'occurrence de z dans les cinq premiers caractères du motif. On aurait alors décalé le motif de $j+1 = 6$ positions vers la droite.

1.1.1 Mise en œuvre.

Écrivons un programme C qui met en œuvre l'algorithme de Boyer–Moore.

Il faut commencer par choisir une structure de données pour représenter la table de décalages. Il s'agit d'un dictionnaire à deux clés : d'une part l'indice j du caractère du motif qui diffère et d'autre part le caractère c du texte.

— Pour la première clé, c'est-à-dire l'indice j, on peut naturellement utiliser un tableau de taille M car toutes les valeurs $0, 1, \ldots, M-1$ seront présentes.
— Pour la seconde clé, en revanche, seuls les caractères apparaissant dans le motif seront des valeurs significatives. Ainsi, pour le motif $m =$ "abracadabra", seuls cinq caractères sont significatifs, à savoir a, b, r, c et d.

D'autre part, certaines entrées de la table sont vides, quand il n'y a pas de décalage possible. Du coup, chaque ligne de la table est un dictionnaire possiblement très creux : il peut être intéressant de le représenter par une table de hachage ou encore un arbre binaire de recherche.

Si l'alphabet est petit, on peut se permettre de représenter chaque ligne par un tableau, avec la valeur -1 pour toutes les cases qui ne correspondent pas à une entrée. Le choix de la valeur -1 n'est pas anodin : ainsi, le décalage à effectuer sera toujours $j - d$ où d est la valeur donnée par la table.

Le programme suivante contient un programme C qui réalise l'algorithme de Boyer–Moore avec une telle table.

Algorithme de Boyer–Moore

```
1
2  int **build_table(char *m, int lm) {
3    int **table = calloc(lm, sizeof(int*));
4    for (int j = 0; j < lm; j++) {
5      table[j] = calloc(256, sizeof(int));
6      for (int c = 0; c < 256; c++)
7        table[j][c] = -1;
8      for (int k = 0; k < j; k++) {
9        unsigned char c = m[k];
10       table[j][c] = k;
11     }
12   }
13   return table;
14 }
15
16 int boyer_moore(char *m, char *t) {
17   int lm = strlen(m), lt = strlen(t);
18   int **table = build_table(m, lm);
19   int count = 0;
20   for (int i = 0; i <= lt - lm; ) {
21     int k = 0;
22     for (int j = lm - 1; j >= 0; j--) {
23       if (t[i + j] != m[j]) {
24         unsigned char c = t[i + j];
25         k = j - table[j][c];
```

```
26          break;
27        }
28      }
29      if (k == 0) {
30        printf("occurrence a la position %d\n", i);
31        count++;
32        k = 1;
33      }
34      i += k;
35    }
36    return count;
37  }
```

La fonction `build_table` construit la table à partir du motif m et de sa longueur lm (lignes 1–13). Les tableaux sont alloués, initialisés avec la valeur -1, puis remplis avec l'affectation de la ligne 9. Comme on parcourt les indices k du plus petit au plus grand, plusieurs occurrences d'un même caractère vont donner plusieurs affectations, chacune écrasant la précédente. Ainsi, on aura bien au final dans $\texttt{table}[j][c]$ le plus grand k tel que $k < j$ et $m[k] = c$.

La fonction `boyer_moore` réalise la recherche proprement dite. Elle commence par construire la table (ligne 17) puis parcourt toutes les positions possibles (ligne 19). On va alors comparer les caractères un par un, en partant de la droite (boucle ligne 21). On retient dans une variable k le décalage trouvé, le cas échéant. Dès qu'il y a une différence entre le motif et le texte, on calcule le décalage en consultant la table (lignes 23–24) et on sort immédiatement de la boucle interne (ligne 25). On note que le décalage k vaut alors au moins 1. En effet, s'il n'y a pas d'entrée dans la table, alors $\texttt{table}[j][c]$ vaut -1 et donc $j - \texttt{table}[j][c] \geq 1$. Et s'il y a une entrée dans la table, alors $0 \leq \texttt{table}[j][c] < j$, par définition, et donc $j - \texttt{table}[j][c] \geq 1$ là encore.

Une fois sorti de la boucle interne, on a trouvé une occurrence si et seulement si $k = 0$. Le cas échéant, on la signale, on incrémente `count` et on donne à k la valeur 1. Dans tous les cas, on avance ensuite dans le texte en ajoutant la valeur de k à l'indice i. On avancera toujours d'au moins une unité, mais possiblement plus dans certains cas.

Complexité.

Quelle est l'efficacité de l'algorithme de Boyer–Moore ?

En premier lieu, il y a le coût de la construction de la table.

Pour notre programme, le coût en temps est $O(M^2)$ et le coût en espace est $O(M)$, si on considère que la taille de l'alphabet est une constante (ici 256). Avec un alphabet très grand (des caractères UTF-8 par exemple), et en optant cette fois pour une table de hachage pour chaque ligne de la table de décalages, on aurait un coût $O(M^2)$ en espace. Il peut descendre à $O(M)$ lorsque de nombreux caractères du motif sont identiques. Dans tous les cas, il est raisonnable d'estimer que la taille M du motif est bien plus petite que la taille N du texte. Dès lors, on peut espérer que le coût quadratique de la construction de la table sera négligeable devant le coût de la recherche.

Venons-en justement à la complexité de la recherche.

Dans le pire des cas, la comparaison entre le motif et le texte se fait systématiquement jusqu'au bout du motif, c'est-à-dire jusqu'à $j = 0$. C'est le cas par exemple si on recherche le mot $abbb\ldots bb$ dans le texte $bbbb\ldots bb$ (aucune occurrence) ou le mot

bbbb…*bb* dans ce même texte *bbbb*…*bb* (une occurrence à chaque position). Dans les deux cas, le nombre total de comparaisons de caractères est $M(N-M+1)$, ce qui n'est pas meilleur qu'avec la recherche naive.
Dans le meilleur des cas, en revanche, la comparaison peut être négative immédiatement, dès le premier caractère testé, c'est-à-dire pour $j = M-1$, et le décalage être aussi grand que M. C'est le cas par exemple si on recherche le mot *aaa*…*aa* dans le texte *bbb*…*bb*. Le nombre total de comparaisons sera alors N/M, car on ne compare qu'un caractère sur M. Ainsi, si on cherche les occurrences d'un motif contenant 1000 caractères a dans un texte contenant 2000 caractères b, on ne fera que deux comparaisons ! Cet exemple extrême illustre notamment l'intérêt d'avoir procédé de la droite vers la gauche.
Entre ces deux cas de figure, on trouve une multitude de situations intermédiaires, où le coût de la recherche varie beaucoup avec le motif et avec le texte.

1.2 Algorithme de Rabin–Karp

L'algorithme de Rabin–Karp repose sur deux idées.

1. La première consiste à utiliser une fonction de hachage h sur les chaînes de caractères et à comparer la valeur de $h(m)$ avec les valeurs de $h(t[i..i+M-1])$ pour toutes les positions $0 \leq i \leq N-M$. Lorsqu'il y a égalité, on a peut-être trouvé une occurrence de m, ce que l'on peut alors vérifier avec une comparaison exhaustive des caractères de m avec les caractères de t à la position i. Mais lorsque les valeurs de hachage diffèrent, on est certain qu'il ne peut y avoir une occurrence à la position i. En effet, une fonction de hachage sur les chaînes est une fonction déterministe qui ne dépend que des caractères. Dès lors, seules des chaînes différentes peuvent avoir des valeurs différentes par h.
   Présenté comme cela, l'algorithme de Rabin–Karp n'a pas l'air très efficace. Il faut a priori un temps proportionnel à M pour calculer $h(t[i..i+M-1])$, d'où une complexité totale en $M(N-M+1)$, ce qui n'est pas meilleure que la recherche naïve.
2. La seconde idée derrière l'algorithme de Rabin–Karp consiste à calculer la valeur de hachage de la sous-chaîne à la position $i+1$ en temps constant à partir de la valeur de hachage pour la sous-chaîne à la position i.
   Quand on a vu les tables de hachage, nous avons proposé pour les chaînes de caractères une fonction de hachage de la forme suivante

$$h(c_0 c_1 \dots c_{M-1}) = \sum_{0 \leq j < M} B^{M-1-j} \times c_j$$

   avec une valeur $B = 31$ choisie empiriquement. Il se trouve qu'une telle fonction possède très exactement la caractéristique qui nous intéresse ici. En effet, si on note c_i le caractère i du texte t, on a

$$h(c_{i+1} c_{i+2} \dots c_{i+M}) = B\left(h(c_i c_{i+1} \dots c_{i+M-1}) - B^{M-1} c_i\right) + c_{i+M} \tag{1}$$

   c'est-à-dire que l'on peut calculer le hachage des M caractères à la position $i+1$ en temps constant à partir du hachage des M caractères à la position i. En précalculant B^{M-1}, il reste seulement deux multiplications, une soustraction et une addition à effectuer. On note que le dernier caractère c_{i+M} est tout simplement ajouté, car son coefficient est B^0.

Bien entendu, le calcul ci-dessus est susceptible de provoquer un débordement arithmétique, comme n'importe quelle fonction de hachage. Mais l'identité n'est pas moins valable dans l'arithmétique modulaire, car on ne fait que des produits, des additions et des soustractions. Et plutôt que de calculer dans une arithmétique machine modulo 2^n, on peut avantageusement faire le calcul modulo P, pour un nombre premier P le plus grand possible.

Le programme suivante met en œuvre cette idée, en prenant comme paramètres $B = 256$ et $P = 1\,869\,461\,003$.

Algorithme de Rabin–Karp

```
1  const uint64_t B = 256;
2  const uint64_t P = 1869461003;
3
4  // le hachage des ‘len‘ premiers caracteres de ‘s‘
5  uint64_t rk_hash(char *s, int len) {
6    uint64_t h = 0;
7    for (int i = 0; i < len; i++) {
8      unsigned char c = s[i];
9      h = (h * B + c) % P;
10   }
11   return h;
12 }
13
14 int rabin_karp(char *m, char *t) {
15   int lm = strlen(m), lt = strlen(t);
16   assert(lm > 0);
17   if (lt < lm) return 0;
18   uint64_t sh = math_power_mod(B, lm - 1, P);
19   uint64_t hm = rk_hash(m, lm), ht = rk_hash(t, lm);
20   int count = 0;
21   for (int i = 0; true; i++) {
22     if (ht == hm && strncmp(t + i, m, lm) == 0) {
23       printf("occurrence  la position %d\n", i);
24       count++;
25     }
26     if (i == lt - lm) break;
27     unsigned char ci = t[i];
28     ht = (ht + P - (ci * sh) % P) % P; // on enlve t[i]
29     unsigned char c = t[i + lm];
30     ht = (ht * B + c) % P; // et on ajoute t[i+lm]
31   }
32   return count;
33 }
```

On choisit ici un nombre premier P tel que P^2 tient encore dans le type `uint64_t` des entiers 64 bits non signés. Ainsi, il n'y a pas de débordement chaque fois que l'on calcule $(x \times y) \bmod P$ dans le code (lignes 9, 28 et 30) avec x et y des entiers modulo P. Pour le calcul de B^{M-1} modulo P, on utilise la fonction `math_power_mod` (ligne 18). Ce programme suppose que $M > 0$ c'est-à-dire que le motif n'est pas vide (ligne 16). Dans le cas d'un motif vide, il suffirait de signaler une occurrence à toute position $0 \leq i \leq N$ et de renvoyer $N + 1$.

Complexité.

Dans le meilleur des cas, la comparaison des deux valeurs de hachage est toujours négative et on n'appelle jamais la fonction `strncmp`. Dans ce cas, on a une complexité en $O(N)$. C'est possiblement moins bien que l'algorithme de Boyer–Moore, qui peut être sous-linéaire dans certains cas. Mais on ne dépend pas de la taille M du motif, si ce n'est pour calculer initialement $h(m)$, ce qui se fait en $O(M)$.
Dans le pire des cas, en revanche, on peut appeler `strncmp` pour chaque position et effectuer systématiquement une comparaison entre le motif et le texte jusqu'au bout. C'est le cas par exemple si on recherche le mot $bbb\ldots bb$ dans le texte $bbbb\ldots bb$, avec une occurrence à chaque position. Le nombre total de comparaisons de caractères est alors $M(N-M+1)$, ce qui est également un pire cas de l'algorithme de Boyer–Moore et n'est pas meilleur que la recherche simple. Il s'agit là cependant d'un cas extrême.
On peut se persuader empiriquement du bien-fondé de l'algorithme de Rabin–Karp en examinant les collisions sur un exemple réaliste.

Exemple

Prenons le texte intégral du roman de Jules Verne *Le Tour du monde en quatre-vingts jours*, considérons toutes les sous-chaînes de taille M qu'il contient et cherchons les collisions par notre fonction de hachage. Pour $M = 5$, on a 67 933 sous-chaînes différentes et seulement trois collisions. Pour chacune de ces collisions, il s'agit de deux sous-chaînes seulement ayant la même valeur de hachage. Pour $M = 10$, on a 324 478 sous-chaînes différentes et seulement 28 collisions, là encore impliquant à chaque fois deux sous-chaînes seulement. Par exemple, les deux sous-chaînes "du flair q" et "quante-deu" ont la même valeur de hachage. Cela veut dire que si l'on cherche l'une des deux dans le texte, on aura un seul faux positif, écarté ensuite par l'appel à `strncmp`. Et si on cherche le mot "automobile" dans le roman, son hachage est distinct de celui de toutes les sous-chaînes de longueur 10 et on conclut que le mot n'apparaît pas sans jamais appeler `strncmp`.