Recherche d'un mot dans un texte

Quentin Fortier

February 20, 2025

Chaînes de caractères : en C

Une chaîne de caractères est un tableau dont les éléments sont des `char` (caractères) et terminé par le caractère spécial `'\0'` :

```c
char s[] = "mpi"; // \0 est ajouté automatiquement
printf("%c", s[0]); // affiche le 1er caractère
s[2] = '2'; // modifie un caractère
strlen(s); // nombre de caractères
for(int i = 0; s[i] != '\0'; i++)
    // parcourir s
```

Chaînes de caractères : en OCaml

Une chaîne de caractères est un tableau dont les éléments sont des `char` (caractères) et « null-terminated » (`'\0'`) :

```
let s = "mpi"
s.[0] (* 1er caractère *)
s.[2] <- '2' (* ERREUR : un string est immutable en OCaml *)
let n = String.length s (* nombre de caractères *)
if s = "mp2i" then ... (* comparaison en O(n) *)
for i = 0 to n - 1 do
    (* parcourir s *)
```

Recherche d'un mot dans un texte

Recherche d'un mot

Entrée : Deux mots `m` et `t`.

Sortie : `m` est-il un facteur de `t` ?

Applications :

1. Recherche d'une séquence ADN
2. Recherche dans un éditeur de t (Visual Code...)
3. ...

Algorithme naïf

Recherche naïve de CGGCAG avec fenêtre glissante :

C	A	A	T	G	T	C	T	G	C	A	C	C	A	A	C	A	C	C	G	G	C	A	G
C	G	G	C	A	G																		

Algorithme naïf

Recherche naïve de CGGCAG avec fenêtre glissante :

C	A	A	T	G	T	C	T	G	C	A	C	C	A	A	C	A	C	C	G	G	C	A	G
	C	G	G	C	A	G																	

Algorithme naïf

Recherche naïve de CGGCAG avec fenêtre glissante :

C	A	A	T	G	T	C	T	G	C	A	C	C	A	A	C	A	C	C	G	G	C	A	G
		C	G	G	C	A	G																

Algorithme naïf

Recherche naïve de CGGCAG avec fenêtre glissante :

C	A	A	T	G	T	C	T	G	C	A	C	C	A	A	C	A	C	C	G	G	C	A	G
			C	G	G	C	A	G															

Algorithme naïf

```c
bool is_substring(char* m, char* t) {
    int k = strlen(m);
    for(int i = 0; i < strlen(t) - k + 1; i++)
        for(int j = 0; t[i + j] == m[j]; j++)
            if(j == k - 1)
                return true;
    return false;
```

Algorithme naïf

```
bool is_substring(char* m, char* t) {
    int k = strlen(m);
    for(int i = 0; i < strlen(t) - k + 1; i++)
        for(int j = 0; t[i + j] == m[j]; j++)
            if(j == k - 1)
                return true;
    return false;
```

Complexité : $O(nk)$ où k est la taille de `m` et `n` la taille de `text`.

Algorithmes de recherche de facteur

On note $k = |m|$ et $n = |t|$.

Algorithme	Complexité	Prétraitement	Mémoire	Méthode
Naïf	O(nk)	0	0	Fenêtre glissante
Rabin-Karp	O(n)	0	0	Fonction de hachage
Boyer-Moore	O(nk)	O(k)	O(k)	Décalage
KMP	O(n)	O(k)	O(k)	Automate (2ème année)

La complexité de Rabin-Karp est donnée en moyenne (à cause de la complexité moyenne O(1) des tables de hachages).
Boyer-Moore est efficace en pratique.

Algorithme de Rabin-Karp

L'idée de Rabin-Karp est d'accélérer la comparaison du mot avec une fenêtre :

```
bool is_substring(char* m, char* text) {
    int k = strlen(m);
    for(int i = 0; i < strlen(text) - k + 1; i++)
        for(int j = 0; text[i + j] == m[j]; j++) // ici
            if(j == k - 1)
                return true;
    return false;
}
```

Algorithme de Rabin-Karp

L'idée de Rabin-Karp est d'accélérer la comparaison du mot avec une fenêtre :

```
bool is_substring(char* m, char* text) {
    int k = strlen(m);
    for(int i = 0; i < strlen(text) - k + 1; i++)
        for(int j = 0; text[i + j] == m[j]; j++) // ici
            if(j == k - 1)
                return true;
    return false;
}
```

Pour cela, il utilise une fonction de hachage et compare les hashs de chaque chaîne de caractères.

Algorithme de Rabin-Karp

L'idée de Rabin-Karp est d'accélérer la comparaison du mot avec une fenêtre :

```
bool is_substring(char* m, char* text) {
    int k = strlen(m);
    for(int i = 0; i < strlen(text) - k + 1; i++)
        for(int j = 0; text[i + j] == m[j]; j++) // ici
            if(j == k - 1)
                return true;
    return false;
}
```

Pour cela, il utilise une fonction de hachage et compare les hashs de chaque chaîne de caractères.
Mais il faut être capable de calculer les hashs rapidement, ce qui va être réalisé avec une « rolling hash », qui permet de déduire le hash d'une fenêtre à partir de la fenêtre précédente.

Algorithme de Rabin-Karp

L'algorithme de Rabin-Karp utilise une fonction de hachage h rapide à calculer (si possible O(1)) et, pour chaque indice i de t :

- Compare $h(m)$ et $h(t[i:i+k])$.
- Si ces deux valeurs sont égales, on compare m et $t[i:i+k]$ lettre par lettre, en O(k).

Ainsi, si $h(m) \neq h(t[i:i+k])$, on sait que $m \neq t[i:i+k]$ et on gagne du temps d'exécution.

Remarque : Comme h n'est *a priori* pas injective, il peut y avoir des collisions, c'est-à-dire $h(m) = h(t[i:i+k])$ et $m \neq t[i:i+k]$.

Algorithme de Rabin-Karp

Étant donné un mot $w = w_0 ... w_{k-1}$, un entier b (le nombre de caractères possibles) et un entier q, on définit $h(w) \in \{0, ..., q-1\}$ par :

$$h(w) = \sum_{i=0}^{n-1} \mathsf{code}(w_i) b^{n-1-i} \mod q$$

où $\mathsf{code}(w_i)$ est le code de la lettre w_i (par exemple, le code ASCII).

Algorithme de Rabin-Karp

Calcul de la fonction de hachage :

```
let hash b q s =
  let rec aux i p =
    if i = -1 then 0
    else ((Char.code s.[i])*p + aux (i-1) ((p*b) mod q)) mod q in
  aux (String.length s - 1) 1
```

Algorithme de Rabin-Karp

```
let rabin_karp text w =
  let k, n = String.length w, String.length text in
  let q = 3719 in (* prime number for modulo *)
  let b = 256 in (* number of characters (basis) *)
  let p = pow b (k - 1) q in (* maximum power of b *)
  let h_w = hash b q w in
  let rec search i h =
    if h = h_w && w = String.sub text i k then i
    else if i >= n - k then -1
    else let h_ = (b*(h - p*Char.code text.[i]) + Char.code text.[
      search (i + 1) (if h_ >= 0 then h_  else h_ + q) in
  search 0 (hash b q (String.sub text 0 k))
```

`pow a n q` renvoie $a^n \mod q$.

Algorithme de Boyer-Moore-Horspool

1. Prétraitement : On utilise un dictionnaire `d` tel que `d[c]` soit l'indice de la première apparition de `c` dans `m`.
   Exemple : si `w = "CGGCAG"` alors `d` a les associations `('A', 1), ('C', 2), ('G', 3)` et `'T'` n'est pas une clé de `d`.

Algorithme de Boyer-Moore-Horspool

1. Prétraitement : On utilise un dictionnaire `d` tel que `d[c]` soit l'indice de la première apparition de `c` dans `m`.
   Exemple : si `w = "CGGCAG"` alors `d` a les associations `('A', 1), ('C', 2), ('G', 3)` et `'T'` n'est pas une clé de `d`.
2. L'algorithme considère alors les lettres de t et les compare aux lettres de m, de droite à gauche. Dès qu'il y a une différence, on décale d'un nombre de caractères donné par `d` et on reprend la comparaison du début.

Algorithme de Boyer-Moore-Horspool

Recherche de CGGCAG :

C	A	A	T	G	~~T~~	C	T	G	T	A	C	G	G	C	A	G

C G G C A G

T n'apparaît pas dans le mot : on décale de k

Algorithme de Boyer-Moore-Horspool

Recherche de CGGCAG :

C	A	A	T	G	T	C	T	G	T	A	~~C~~	G	G	C	A	G
						C	G	G	C	A	G					
									↑							

La dernière lettre ne correspond pas : on décale de 2 pour avoir un C

Recherche de CGGCAG :

C	A	A	T	G	T	C	T	G	T	A	~~C~~	G	G	C	A	G
								C	G	G	C	A	G			

$\uparrow$

La dernière lettre ne correspond pas : on décale de 2 pour avoir un C

Recherche de CGGCAG :

C	A	A	T	G	T	C	T	G	T	A	C	~~G~~	G	C	A	G
								C	G	G	C	A	G			

L'avant-dernière lettre ne correspond pas : on décale de 2 pour avoir un G

Algorithme de Boyer-Moore-Horspool

Recherche de CGGCAG :

C	A	A	T	G	T	C	T	G	T	A	C	G	G	C	~~A~~	G
										C	G	G	C	A	G	

La denière lettre ne correspond pas : on décale de 1 pour avoir un A

Algorithme de Boyer-Moore-Horspool

Recherche de CGGCAG :

C	A	A	T	G	T	C	T	G	T	A	C	G	G	C	A	G
											C	G	G	C	A	G

On a trouvé le mot !

Algorithme de Boyer-Moore-Horspool

```
let boyer_moore_horspool t w =
    let k, n = String.length w, String.length t in
    let module M = Map.Make(Char) in
    let rec make_d i =
        if i = k then M.empty
        else make_d (i+1) |> M.add w.[k - i - 1] i in
    let d = make_d 1 in

    let rec search i j = (* teste si w[:k-j] = t[i-k:i-j] *)
        if i >= n then -1
        else if j = k then i - k + 1
        else if w.[k - j - 1] = t.[i - j] then search i (j + 1)
        else match M.find_opt t.[i - j] d with
            | Some s -> search (s + i) 0
            | None -> search (i + k - j) 0 in
    search (k - 1) 0
```

Où on a utilisé le module `Map` qui implémente un dictionnaire persistant (par arbre binaire de recherche) :

```
let module M = Map.Make(Char) (* dictionnaire dont les clés sont d
let d = M.empty (* dictionnaire vide *)
let d2 = M.add 'a' 1 d (* ajoute une clé 'a' avec valeur 1 *)
M.find_opt 'a' d2 (* renvoie Some 1 *)
M.find_opt 'b' d2 (* renvoie None *)
```

Algorithme de Boyer-Moore-Horspool : Complexité

Exercice

1. Quelle est la complexité de Boyer-Moore-Horspool dans le meilleur cas ?
2. Montrer que l'algorithme de Boyer-Moore-Horspool a une complexité en $\Theta(nk)$ dans le pire des cas.