Recherche d'un mot dans un texte

Quentin Fortier

April 3, 2022

Code Tests

Chaînes de caractères : en C

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

```c
char s[] = "mpsi"; // \0 est ajouté automatiquement
printf("%c", s[0]); // affiche le 1er caractère
s[2] = '2'; // modifie un caractère
strlen(s); // nombre de caractères
strcmp(s, "mpsi"); // 0 si les strings sont égales
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 = "mpsi"
s.[0] (* 1er caractère *)
s.[2] <- "2" (* ERREUR : un string est immutable en OCaml *)
let n = String.length s (* nombre de caractères *)
s = "mpsi" (* comparaison de 2 strings *)
for i = 0 to n - 1 do
    (* parcourir s *)
```

Recherche d'un mot dans un texte

Problème

Entrée : deux chaînes de caractères `m` et `texte`.
Sortie : un indice `i` à partir duquel `m` apparaît dans `texte`, ou `-1` si `m` n'apparaît pas.

Exemple : Si `m = "thm"` et `texte = "Un algorithme"` alors il faut renvoyer `i = 9`.

Recherche d'un mot dans un texte

Problème

Entrée : deux chaînes de caractères `m` et `texte`.
Sortie : un indice `i` à partir duquel `m` apparaît dans `texte`, ou `-1` si `m` n'apparaît pas.

Exemple : Si `m = "thm"` et `texte = "Un algorithme"` alors il faut renvoyer `i = 9`.

Applications :

1. Recherche d'une séquence ADN
2. Recherche dans un éditeur de texte (Visual Code...)
3. ...

Algorithme naïf

Recherche naîve 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

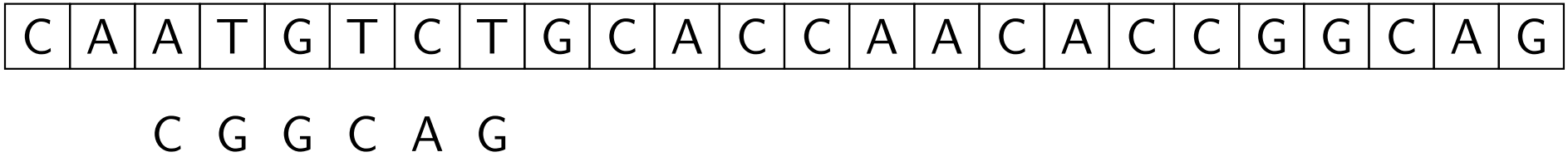
Recherche naîve de CGGCAG avec fenêtre glissante :

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

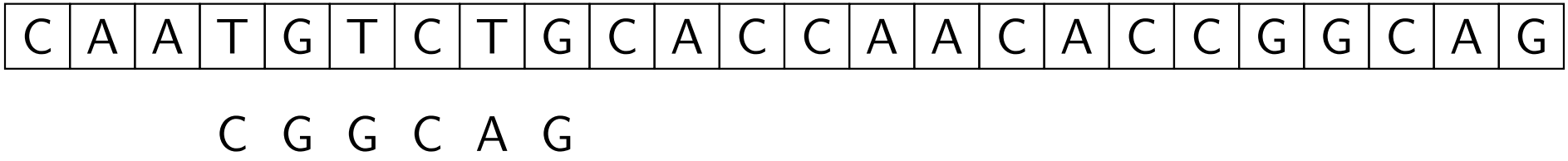
Algorithme naïf

Recherche naîve de CGGCAG avec fenêtre glissante :



Algorithme naïf

Recherche naîve de CGGCAG avec fenêtre glissante :



Algorithme naïf

```
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++)
            if(j == k - 1)
                return true;
    return false;
}
```

Algorithme naïf

```
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++)
            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 sous-mot

Pour cherche un sous-mot de longueur k dans un texte de longueur n :

Algorithme	Complexité	Prétraitement	Mémoire	Méthode
Naïf	$O(nk)$	0	0	Fenêtre glissante
Rabin-Karp	$O(n)$	0	0	Fonction de hachage
Boyer-Moore	$O(nk)$	$O(k)$	$O(k)$	Droite à gauche, 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 et peut-être amélioré en $O(n)$ (mais HP).

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

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 :

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

où $\text{code}(w_i)$ 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

1. **Prétraitement** : Calculer, dans un dictionnaire `d`, la première apparition (en partant de la fin, sans compter la denière lettre) de chaque lettre `c` dans le mot `w`.
   Exemple : si `w = "CGGCAG"` alors `d` a les associations `('A', 1), ('C', 2), ('G', 3)` et `'T'` n'est pas une clé de `d`.

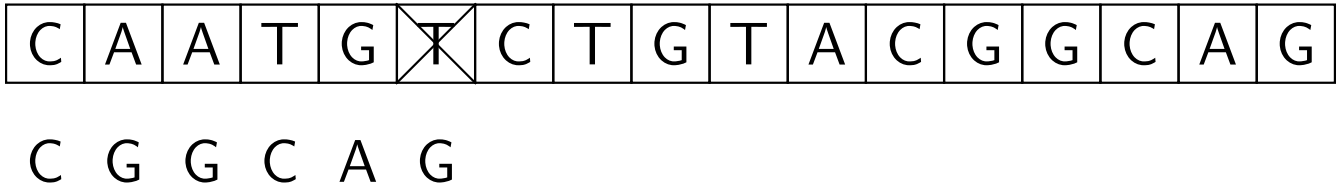
Algorithme de Boyer-Moore

1. **Prétraitement** : Calculer, dans un dictionnaire `d`, la première apparition (en partant de la fin, sans compter la denière lettre) de chaque lettre `c` dans le mot `w`.
   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. Regarder les lettres du mot et du texte une par une, en parcourant le mot de droite à gauche.
   Dès qu'il y a une différence, décaler d'un nombre de caractères donné par `d` et reprendre la comparaison du début.

Algorithme de Boyer-Moore

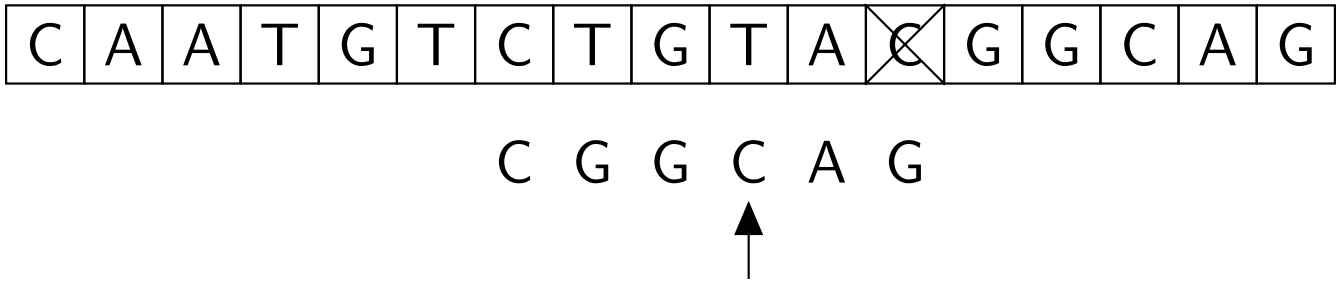
Recherche de CGGCAG :



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

Algorithme de Boyer-Moore

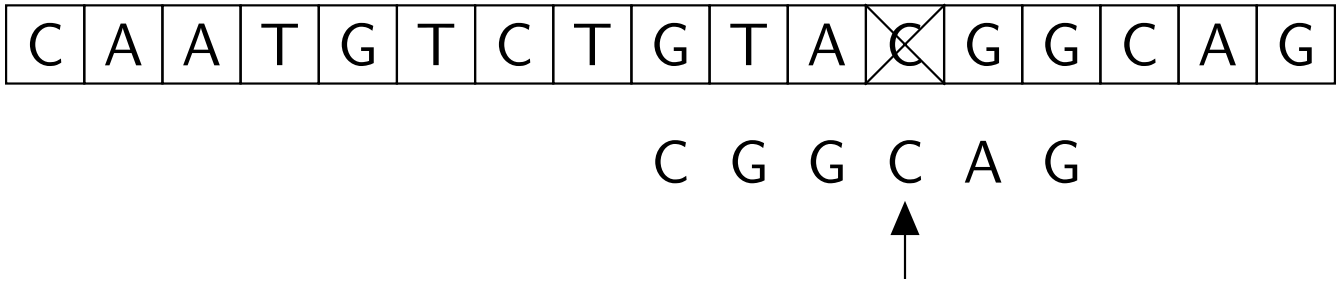
Recherche de CGGCAG :



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

Algorithme de Boyer-Moore

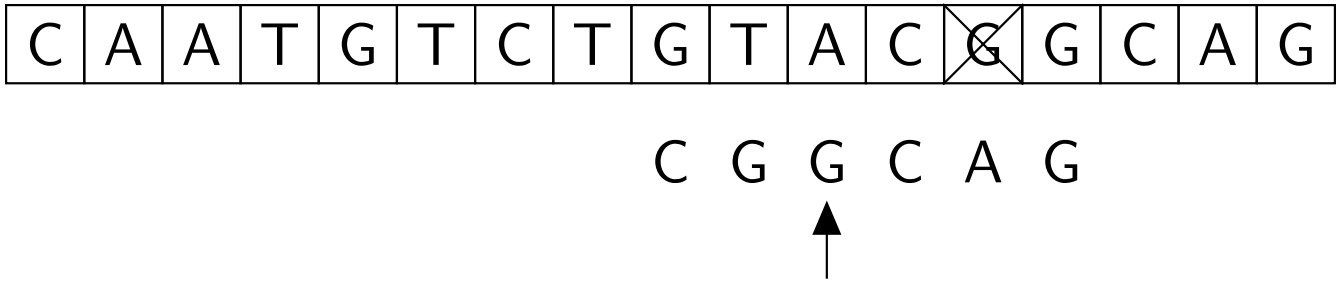
Recherche de CGGCAG :



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

Algorithme de Boyer-Moore

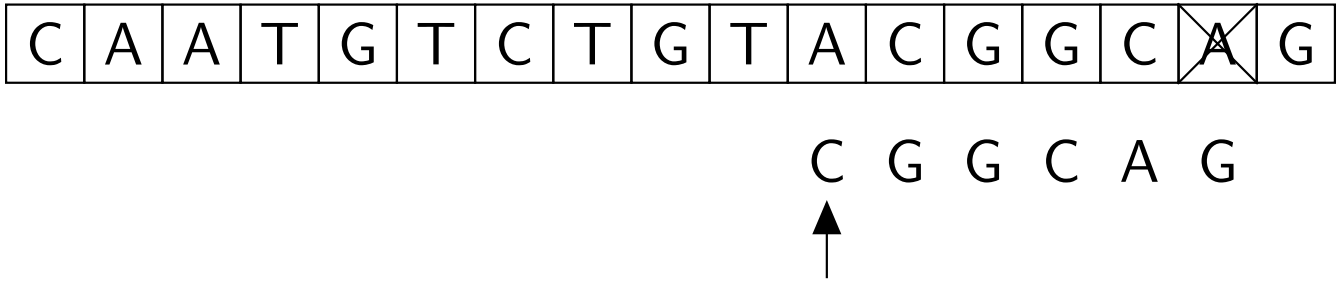
Recherche de CGGCAG :



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

Algorithme de Boyer-Moore

Recherche de CGGCAG :



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

Algorithme de Boyer-Moore

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

```
let boyer_moore text w =
  let k, n = String.length w, String.length text 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 = (* test if w[:k-j] = text[i-k:i-j] *)
    if i >= n then -1
    else if j = k then i - k + 1
    else if w.[k - j - 1] = text.[i - j] then search i (j + 1)
    else match M.find_opt text.[i - j] d with
      | Some s -> search (max (i + 1) (s + i)) 0
      | None -> search (i + k - j) 0 in
  search (k - 1) 0
```

Algorithme de Boyer-Moore : Complexité

Pour chaque indice `i` du texte, Boyer-Moore va comparer au plus les `k` caractères du mot finissant en `i`. D'où une complexité $\boxed{\mathrm{O}(nk)}$.

Algorithme de Boyer-Moore : Complexité

Pour chaque indice `i` du texte, Boyer-Moore va comparer au plus les `k` caractères du mot finissant en `i`. D'où une complexité $\boxed{\mathrm{O}(nk)}$.

Exercice

Trouver un texte et un mot tel que la complexité soit effectivement $\Theta(nk)$) (c'est-à-dire égal à nk, à une constante près).

Algorithme de Boyer-Moore : Complexité

Pour chaque indice `i` du texte, Boyer-Moore va comparer au plus les `k` caractères du mot finissant en `i`. D'où une complexité $\boxed{\mathrm{O}(nk)}$.

Exercice

Trouver un texte et un mot tel que la complexité soit effectivement $\Theta(nk)$) (c'est-à-dire égal à nk, à une constante près).

Dans le meilleur cas, la complexité est

Algorithme de Boyer-Moore : Complexité

Pour chaque indice `i` du texte, Boyer-Moore va comparer au plus les `k` caractères du mot finissant en `i`. D'où une complexité $\boxed{\mathrm{O}(nk)}$.

Exercice

Trouver un texte et un mot tel que la complexité soit effectivement $\Theta(nk)$) (c'est-à-dire égal à nk, à une constante près).

Dans le meilleur cas, la complexité est $\boxed{\mathrm{O}(\frac{n}{k})}$: on saute de k à chaque fois.