

Recherche textuelle

Il nous arrive lorsqu'on écrit un programme de vouloir modifier le nom d'une variable par exemple, (et celle ci peut apparaître dans de nombreux endroits du texte qu'est le programme en cours d'édition)

On utilise en général la touche CTRL + F (ou cmd + F) pour rechercher toutes les occurrences de cette variable dans le programme d'où le problème plus général :

Données Un texte t de longueur n et un motif m avec $m \leq n$

Résultat Toutes les occurrences (positions) du motif m dans t (c'est la position du premier caractère de m dans t lorsque m se superpose à une sous-chaine de t à cet endroit)

Exemple $t = \text{ATCATATACCGATA}$ et $m = \text{ATA}$ $m = 3$ et $n = 14$

ATCATATACCGATA

ATA

□ATA

□□ATA

□□□ATA (match)

Le motif se trouve en position 3, 5 et 15

1 Algorithme naïf

La première idée est de faire "glisser" une fenêtre de longueur la taille du motif sur le texte ,d'un et de comparer pour chaque position de la fenêtre sur le texte la sous-chaine sélectionnée par la fenêtre et le motif

En pseudo-code cela donne

On note comme en Python (slicing) $t[i : j]$ la sous chaine allant du caractère $t[i]$ au caractère $t[j-1]$

Algorithme 1 : Recherche naïve

```
positions (m,t)
Données : Un texte t et un motif m
Résultat : une liste des positions de m dans t
1 début
2   occurrences ←  $\phi$ 
3   pour  $i \leftarrow 0 ; i < \text{longueur}(t) - \text{longueur}(m) + 1 ; i++$  faire
4       si  $t[i : i + \text{longueur}(m)] = m$  alors
5           occurrences ← occurrences  $\cup \{i\}$ 
6       fin
7   fin
8   retourner occurrences
9 fin
```

En Python

```

def positions(texte, motif):
    occurrences = []
    for i in range(len(texte) - len(motif) + 1):
        if texte[i:i+len(motif)] == motif:
            occurrences.append(i)
    return occurrences

```

1.1 Complexité

La comparaison de la fenêtre `texte[i:i+len(motif)]` et du motif est en $O(\text{longueur}(\text{motif}))$
dans le pire des cas

Donc **dans le pire des cas** la recherche naïve des occurrences est en $O(\text{longueur}(\text{motif}) \times \text{longueur}(\text{texte}))$

1.2 Exemple

Le texte est la séquence génomique d'une bactérie *Vibrio cholerae* (1,1 Mo) et le motif est ATGATCAAG

1.3 Exercices

Exercice 1

Compter le nombre de comparaisons de caractères dans l'exemple

Exercice 2

Réécrire la fonction en Python `==` qui compare deux chaînes de caractères

Exercice 3

Sur la séquence génomique de 1,1 Mo de la bactérie *Vibrio cholerae* la recherche naïve du 9-mer ATGATCAAG a mis à peu près 0,28 s

1. Combien de temps mettra à peu près la recherche des occurrences du même 9-mer pour une séquence de 4,8 Mo de *SalmonellaEnterica* ?
2. Combien de temps mettra à peu près la recherche des occurrences du 4 -mer ATGA pour la séquence de 1,1 Mo ? pour la séquence de 4,8 Mo ?

2 Algorithme de Boyer-Moore

2.1 Règle du mauvais caractère

Le but est **en ayant des informations sur le motif de glisser plus vite la fenêtre sur le texte**

On procède comme pour la recherche naïve la fenêtre glisse de la gauche vers la droite
MAIS on ne délègue plus à Python la comparaison du motif et de la fenêtre

On va redéfinir cette fonction de comparaison pour pouvoir "glisser" plus vite

La comparaison de la fenêtre et du motif **se fait de la droite vers la gauche**

Regardons cela sur le même exemple

ATCATATACCGATA
ATA

Au début la fenêtre est ATC et le motif est ATA, comme on compare en allant de la droite vers la gauche dès la première comparaison puisque $C \neq A$ on va donc décaler la fenêtre **mais plus de 1 caractère** car le caractère C de la fenêtre ne se trouve pas dans le motif donc on peut décaler le motif **sur toute sa longueur**

ATCATATACCGATA
□□□ATA (match)

On obtient la première position du motif avec $i = 3$

On glisse ensuite d'un caractère, voici mis en correspondance la fenêtre et le motif

TAT
ATA

Cette fois-ci le premier caractère où il y a mismatch le caractère T **appartient au motif**, on aligne les deux T et on recommence la comparaison ce qui nous donne la deuxième occurrence pour $i = 5$

On décale la fenêtre d'un caractère et cette fois ci on a

TAC
ATA

On décale

CGA
ATA

Cette fois ci le caractère du premier mismatch G n'est pas dans le motif

D'où le décalage de la fenêtre sur la sous-chaine ATA et on trouve le dernier "match"

En tout il y a eu 14 comparaisons de caractères pour 20 précédemment pour la recherche naive

Définition 1 La dernière occurrence non finale d'un caractère x relativement au motif m est le plus grand des indices où se trouve x dans le motif **à part le dernier**

$$d_m(x) = \max\{j < \text{longueur}(m) - 1 \text{ où } t[j] = x\}$$

Si x n'appartient pas au motif $d_m(x) = -1$

Théorème 1 (règle du mauvais caractère (Boyer-Moore))

Pour un certain i tel que $0 \leq i \leq \text{longueur}(t) - \text{longueur}(m)$

la fenêtre $t[i : i + m]$ vérifie

Il existe j tel que $0 < j < \text{longueur}(m)$ tel que $t[i+j] \neq m[j]$

avec $t[i+k] = m[k]$ pour $k > j$

On pose $d = d_m(t[i + j])$

Par conséquent $t[l : l+m] \neq m$ pour tout l vérifiant $i < l < i + j - d$

Preuve

Puisque d est la plus grande position **non finale** du premier mismatch cela veut dire que tous les caractères situés dans le motif après la position d strictement sont différents de $t[i+j]$, pour peu que $j > d$ on est sur que tout décalage inférieur à $j - d$ mettra en face de $t[i+j]$ un caractère de position strictement supérieur à d donc différent de $t[i+j]$. A cause de cette différence sur un caractère il n'y aura pas match entre la fenêtre et le motif

Par contre si $d \geq j$ alors on décalera d'une unité

D'où l'algorithme de Boyer-Moore concernant la règle du mauvais caractère

Algorithme 2 : Règle du mauvais caractère

```
MC (m,t,d)
  Données : Un texte t ,un motif m, un tableau d
  Résultat : une liste des positions de m dans t
1 début
2   occurrences  $\leftarrow \phi$ 
3    $i \leftarrow 0$ 
4   tant que  $i \leq \text{longueur}(t) - \text{longueur}(m)$  faire
5      $j \leftarrow \text{longueur}(m)$ 
6     tant que  $j \geq 0$  et  $t[i + j] = m[j]$  faire
7        $j \leftarrow j - 1$ 
8     fin
9     si  $j = -1$  alors
10      occurrences  $\leftarrow$  occurrences  $\cup \{i\}$ 
11     fin
12      $i \leftarrow i + \max(1, j - d[t[i + j]])$ 
13  fin
14  retourner occurrences
15 fin
```

2.2 Application

L'algorithme de Boyer-Moore sera plus rapide que l'algorithme naïf surtout si le texte contient des caractères différents des caractères du motif en assez grande quantité, de telle sorte que l'évènement "le caractère du premier mismatch n'est pas dans le motif" sera plus fréquent

Ainsi chercher "ATGATCAAG" dans une séquence génomique ne se fera pas plus vite 'et peut-être même moins vite (voir TP)) en utilisant l'algorithme de Boyer-Moore plutôt que l'algorithme naïf

On verra en TP que la recherche du mot "maintenant" dans la nouvelle "Le scarabée d'Or" d'Edgar Poe (traduite en Français) donne comme occurrences [20795, 21675, 26241, 27333, 30364, 31241, 32044, 32423, 37889, 39033, 68660, 71043, 71316, 82003] et que le temps mis est à peu près 9 millièmes de secondes

Le temps mis par l'algorithme naïf est à eu près 2 centièmes

2.3 Exercices

Exercice 1

Donner le tableau associatif d pour les motifs suivants

1. ATATATA
2. ATCTAGGATC

Exercice 2

Ecrire une fonction Python qui retourne un dictionnaire d contenant les dernières occurrences non finales des caractères appartenant au motif (sauf le dernier caractère)

La complexité de cet algorithme est en $O(\text{longueur}(\text{motif}))$

Exercice 3

Exécuter l'algorithme de Boyer-Moore avec la règle du mauvais caractère pour la recherche du motif 'chat' dans le texte 'Il exécute un entrechat'

Exercice 4

1. Implémentation l'algorithme de Boyer-Moore avec la règle du mauvais caractère en Python
2. Ajouter à ce programme un compteur de comparaisons
3. Comparer avec la recherche naive