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 $m \leqslant n$
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-chaîne de t à cet endroit)
Exemple t = ATCATATACCGATA et m = ATA $m = 3$ et $n = 14$
ATCATATACCGATA
ATA
□ATA
□□ATA
□□□ATA (match)
****
Le motif se trouve en position 3, 5 et 15

1 Algorithme "naif"

La première idée est de faire "glisser" une fenêtre de longueur la taille du motif sur le texte 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 naive

```
positions (m,t)
Données : Un texte t et un motif m
Résultat : une liste des positions de m dans t
1 début
2 |  occurences ← φ
3 |  pour i ← 0 ; i < longueur(t)-longueur(m)+1 ;i++ faire
4 |  |  si t[i :i+longueur(m)] = m alors
5 |  |  |  occurences ← occurences ∪ {i}
6 |  |  fin
7 |  fin
8 |  retourner occurences
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 Coût en temps

La comparaison de la fenêtre texte[i :i+len(motif)] et du motif est proportionnelle à la longueur du motif **dans le pire des cas**.
Donc **dans le pire des cas** la recherche naive des occurences est proportionnelle au produit de la longueur du motif et de la longueur du texte.

1.2 Exemple

Le texte est la séquence génomique d'une bactérie Vibriocholerae (1,1 Mo) et le motif est 'ATGATCAAG'.
Le motif apparaît 17 fois dans la séquence génomique aux positions [116556, 149355, 151913, 152013, 152394, 186189, 194276, 200076, 224527, 307692, 479770, 610980, 653338, 679985, 768828, 878903, 985368]
Si 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, 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 Algorithme de Boyer-Moore (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.**
Comment ? 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 occurence 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
L'algorithme de Boyer-Moore nécessite un **pré-traitement du motif** dans un dictionnaire dans lequel sera rangé des informations concernant les caractères du motif et du texte.

Définition 1 *La dernière occurence 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***
$d_m(x) = max\{j < longueur(m) - 1 \text{ où } t[j] = x\}$
Si x n'appartient pas au motif $d_m(x) = -1$

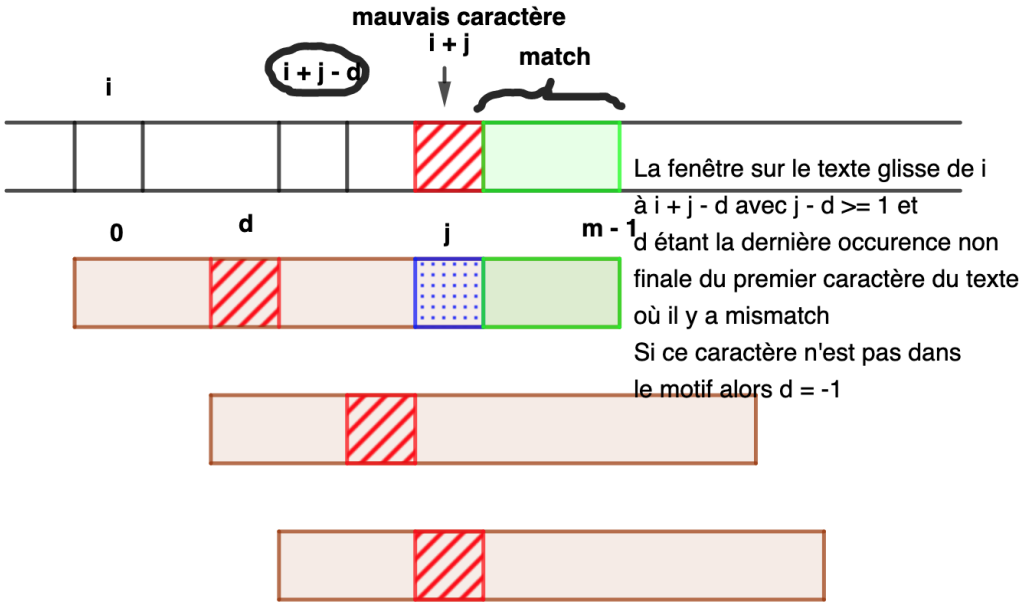
Exemple :
Si le motif est ATGATCAAG :

1. La dernière occurrence non finale de A est 7 car A apparaît aux positions 0,3,6 et 7 et la dernière occurrence apparaît à la position 7
2. La dernière occurrence non finale de G est 2 car G apparaît en positions 2 et 8, **mais il se trouve que 8 est la dernière position possible dans le motif** donc la dernière occurrence non finale de G est 2.

Ainsi pour le motif ATGATCAAG le dictionnaire résultant du pré-traitement est
d = {"A" :7, "C" : 5, "G" : 2,"T" :4}
On note `t` le texte et `m` le motif.
Le motif va glisser le long du texte en partant de la position `i = 0` jusqu'à `len(t) - len(m)-1`.
Soit `i` la position générale du premier caractère du motif :
On met en correspondance le motif et la fenêtre correspondante du texte, notée `t[i:i+longueur(m)]`, en commençant par le dernier caractère du motif.
Pour faire la comparaison on utilise une variable `j` initialisée à `len(m)-1`, puis on compare `t[i+j]` à `m[j]`.

Soit `j` la première valeur telle qu'on a `t[i+j] != m[j]`, on dit alors que le caractère `t[i+j]` est un mauvais caractère.
Il y a peut-être des indices `k` tel que :
`t[i+k] = m[k]` si `k > j`
On va alors utiliser le pré-traitement et calculer $d = d_m(t[i+j])$
Cette information nous sera utile pour savoir si le mauvais caractère se trouve ou ne se trouve pas dans le motif en amont de `j`.

1. Si le mauvais caractère ne se trouve pas entre l'indice 0 et l'indice `j` dans ce cas `d = -1` alors on peut glisser le motif de `i` à `i+ j + 1` car tout caractère dans le motif d'indice strictement inférieur à j est différent de `t[i+j]`
2. Si le mauvais caractère a pour dernière occurence non finale `d >= 0` alors il y a deux cas possibles :
   (a) soit `d < j` dans ce cas on glisse le motif de `i` à `i + j - d` car tout caractère dans le motif d'indice strictement compris entre d et j est différent de `t[i+j]`
   (b) soit `d > j` dans ce cas on va avancer le motif que d'un caractère car on ne peut rien dire des caractères du motif qui précèdent `m[j]`
   (c) On englobe les deux cas en faisant i ← i +max(1,j-d)



Théorème 1 *(Règle du mauvais caractère (Boyer-Moore))*
Pour un certain i tel que $0 \leqslant i \leqslant longueur(t) - longueur(m) - 1$
la fenêtre t[i : i + longueur(m)] vérifie :
Il existe j tel que $0 \leqslant j < longueur(m)$ *tel que* $t[i+j] \neq m[j]$
avec $t[i+k] = m[k]$ *pour* $k > j$ *(j est le lieu dans le motif du premier mismatch)*
On pose $d = d_m(t[i+j])$
Par conséquent on peut itérer en faisant i ← i +max(1,j-d)

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      occurences ← φ
3      i ← 0
4      tant que i ⩽ longueur(t) − longueur(m) faire
5          j ← longueur(m)
6          tant que j ⩾ 0 et t[ i + j] = m[j] faire
7              j ← j − 1
8          fin
9          si j = −1 alors
10             occurences ← occurences ∪ {i}
11         fin
12         i ← i + max(1, j − d[t[i + j]])
13     fin
14     retourner occurences
15 fin
```

2.1 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

2.2 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(longueur(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