

# Optimisation d'une recherche textuelle



Dans cette activité, 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. Par exemple, il y a deux occurrences du motif "bra" dans le texte "abracadabra".

Plus précisément, on va chercher à quelles positions dans le texte le motif apparaît. 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).

Objectif : Comparer les performances de plusieurs algorithmes de recherche de correspondance entre un motif et un texte.

## 1. Notations utilisées

Dans toute la suite on cherche donc la première occurrence d'un motif de longueur  $p$  dans un texte de longueur  $n$ .

À un moment donné de la recherche, on observe une fenêtre de taille  $p$  du texte complet, sur laquelle on aligne le motif, et on regarde s'il y a bien correspondance. S'il n'y a pas correspondance, on recommencera la recherche avec une fenêtre décalée vers la droite dans le texte.

Dans tous les algorithmes présentés ici, la fenêtre se déplacera toujours de gauche à droite. Nous noterons  $i$  la position de la fenêtre dans le texte : c'est l'index du premier caractère du texte qui apparaît dans la fenêtre.

Nous noterons  $j$  l'index dans le motif du caractère du motif que nous comparons avec son analogue du texte et on compare `motif[j]` avec `texte[i + j]`

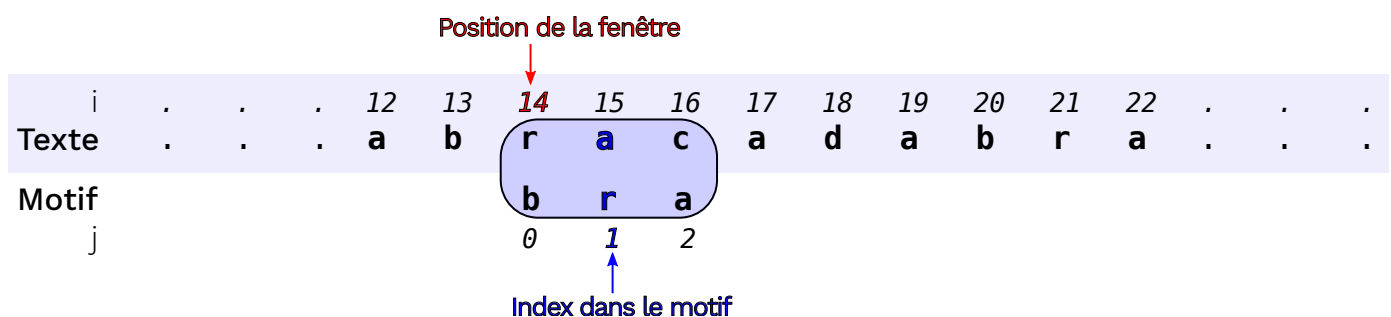


Figure 1 : Méthode de recherche à partir d'une fenêtre glissante de gauche à droite

Dans la figure ci-dessus, on représente une fenêtre à la position  $i = 14$  dans le texte, pour un motif de longueur  $p = 3$ . On compare le caractère 'r' qui figure à l'index  $j = 1$  du motif avec le caractère 'a' qui figure à l'index  $i + j = 15$  dans le texte.

Quand la fenêtre présente un défaut de correspondance entre les caractères du texte et ceux du motif, on déplace la fenêtre. Si le motif correspond parfaitement au texte dévoilé dans la fenêtre, on a trouvé une occurrence à la position  $i$ .

On écrira donc une fonction correspondance qui renvoie, pour une fenêtre en position  $i$ , un couple formé d'un booléen `correspond`, égal à `True` si il y a correspondance du motif et à `False` sinon ; et un entier `decalage` qui indique de combien on souhaite augmenter la position  $i$  de la fenêtre pour la prochaine recherche.

Il est alors facile d'écrire une fonction de recherche :

```
def cherche(texte, motif) :
    n, p = len(texte), len(motif)
    i = 0
    positions = []
    while . . . . :
        correspond, decalage = correspondance(texte, motif, i)
        if correspond :
            positions.append(i)
        i = i + decalage
    return positions
```

Programme 1 : Fonction de recherche utilisée dans cette activité

**Q1. Etablir** l'expression associée à la boucle `while` de la fonction `cherche` afin de décaler la fenêtre de recherche du début à la fin du texte sans dépassement.

**Q2. Déterminer** les valeurs valeurs renvoyées par les appels suivants :

```
>>> cherche('abracadabra', 'ra')
>>> cherche('abracadabra', 'cra')
```

## 2. Recherche naïve

L'algorithme naïf consiste simplement à comparer un à un, de gauche à droite, les caractères du texte apparaissant dans la fenêtre avec ceux du motif puis de décaler la fenêtre vers la droite d'un caractère.

On en déduit l'écriture de la fonction correspondance :

```
def correspondance(texte, motif, i) :
    p, n = len(motif), len(texte)
    assert i + p <= n
    for j in range(p):
        if texte[i + j] != motif[j]:
            return (False, 1)
    return (True, 1)
```

Programme 2 : Fonction de test de correspondance entre un texte et un motif à l'indice  $i$

**Q3. Décrire** le rôle de la deuxième instruction (`assert`) de la fonction correspondance du programme 1.



**Q4.** Dans le cas de la fonction correspondance définie comme dans le programme 1, **déterminer** le nombre de comparaisons de caractères effectués pendant le calcul de `cherche('chercher', rechercher et chercher encore', 'chercher')`

### Coût d'exécution

**Q5. Justifier** que dans ce cas le coût d'exécution de cette recherche est  $p(n-p)$  au pire des cas.

## 3.Algorithme de Boyer-Moore, version simplifiée de Horspool

L'algorithme de Boyer-Moore utilise un prétraitement du motif à chercher dans un texte pour accélérer cette recherche. Son principe est le suivant :

- x Comme dans l'étude précédente, on teste la présence du motif dans le texte à des positions de plus en plus grandes, en partant de  $i = 0$ .
- x Pour une position donnée, on compare les caractères du motif et du texte de la droite vers la gauche. Il s'agit là du sens inverse de celui utilisé précédemment.
- x Si tous les caractères coïncident, on a trouvé une occurrence. Sinon, on note  $j$  l'indice de la première différence.
- x 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 inférieur à  $j$  tel que :

$\text{motif}[k] == \text{texte}[i+j]$  s'il existe

→ la grandeur  $j + 1$  sinon.

Exemple :

Supposons que l'on recherche le motif 'abracadabra' dans un texte. Dans la situation illustrées en dessous, on est en train de tester l'occurrence de ce motif à une certaine position  $i$  dans le texte. Comme indiqué plus haut, la recherche s'effectue de droite à gauche, en commençant par la fin du motif. On voit que les cinq premiers caractères coïncident ('dabra') mais 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.

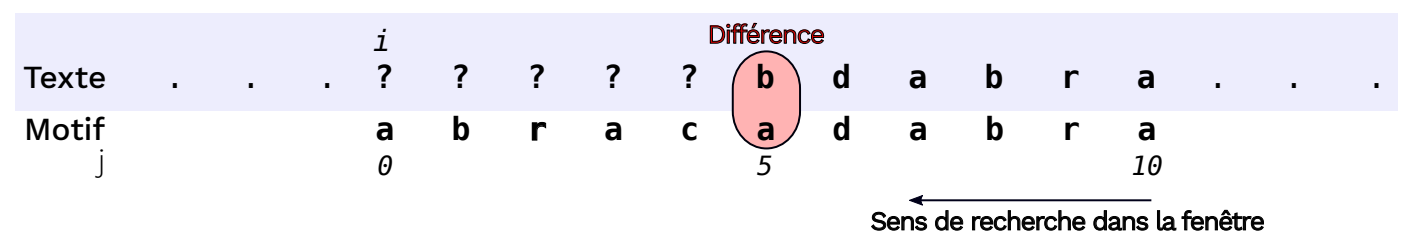


Figure 2: Exemple de recherche de coïncidence dans l'algorithme de Boyer Moore

Le décalage est alors calculé en recherchant dans la table de décalages le plus grand indice inférieur à  $j$ , de la lettre 'b' dans le motif. Cet indice  $k$  est 1 car le dernier 'b' avant l'indice  $j$  ( $= 5$ ) est placé en deuxième position dans le motif. Le décalage est alors à  $j - k$  soit  $5 - 1 = 4$ . Le motif est alors décalé vers la droite de 4 caractères afin d'aligner la lettre 'b' du texte avec le premier 'b' du motif.



La construction automatique de la table de décalage `tab_dec` peut être réalisée avec la fonction suivante :

```
def construire_table_decalage(motif) :
    global tab_dec
    tab_dec = [{} for _ in range(len(motif))]
    for j in range(len(motif)) :
        for k in range(j) :
            tab_dec[j][motif[k]] = k
```

Programme 3 : Construction de la table de décalage

**Q6. Déterminer** le contenu de la table de décalage `tab_dec` pour le motif 'banane' puis pour le motif 'chercher'.

Dans le cas d'une recherche de motif avec l'algorithme de Boyer-Moore, la fonction correspondance peut s'écrire :

```
def correspondance(texte, motif, i) :
    global tab_dec
    p, n = len(motif), len(texte)
    for j in range(p - 1, -1, -1) :
        x = texte[i + j]
        if x != motif[j] :
            k = tab_dec[j].get(x)
            if k == None :
                k = -1
            return (False, j - k)
    return (True, 1)
```

Programme 4 : Fonction de test de correspondance dans l'algorithme de Boyer Moore

**Q7.** En utilisant le résultat de la question précédente, dérouler à la main l'exécution de l'appel de la fonction `cherche('chercher, rechercher et chercher encore', 'chercher')` dans le cas où la fonction `correspondance` est définie par le programme 4. **Donner** notamment les valeurs successives de la variable `i`. **Indiquer** le nombre total de comparaisons de caractères. **Comparer** avec le résultat de la question **Q4**.

