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]`

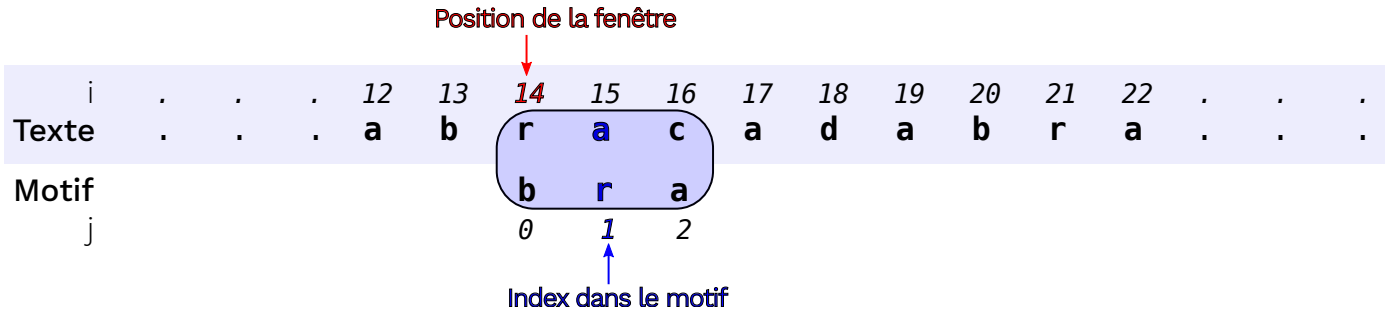

Figure 1 : Méthode de recherche à partir d'une fenêtre glissante de gauche à droite

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.

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

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

```
>>> cherche('abracadabra', 'ra')
[2, 9]
>>> 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.

Cette instruction est une assertion qui permet d'exécuter la fonction si la fenêtre glissante ne dépasse pas la longueur du texte. En cas de dépassement, l'assertion renvoie une erreur.

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')

cpt | i | texte | motif	cpt | i | texte | motif	
(1) 0 : c <-> c	(22) 10 : r <-> c	(43) 20 : <-> c
(2) 0 : h <-> h	(23) 11 : e <-> c	(44) 21 : e <-> c
(3) 0 : e <-> e	(24) 12 : c <-> c	(45) 22 : t <-> c
(4) 0 : r <-> r	(25) 12 : h <-> h	(46) 23 : <-> c
(5) 0 : c <-> c	(26) 12 : e <-> e	(47) 24 : c <-> c
(6) 0 : h <-> h	(27) 12 : r <-> r	(48) 24 : h <-> h
(7) 0 : e <-> e	(28) 12 : c <-> c	(49) 24 : e <-> e
(8) 0 : r <-> r	(29) 12 : h <-> h	(50) 24 : r <-> r
(9) 1 : h <-> c	(30) 12 : e <-> e	(51) 24 : c <-> c
(10) 2 : e <-> c	(31) 12 : r <-> r	(52) 24 : h <-> h
(11) 3 : r <-> c	(32) 13 : h <-> c	(53) 24 : e <-> e
(12) 4 : c <-> c	(33) 14 : e <-> c	(54) 24 : r <-> r
(13) 4 : h <-> h	(34) 15 : r <-> c	(55) 25 : h <-> c
(14) 4 : e <-> e	(35) 16 : c <-> c	(56) 26 : e <-> c
(15) 4 : r <-> r	(36) 16 : h <-> h	(57) 27 : r <-> c
(16) 4 : , <-> c	(37) 16 : e <-> e	(58) 28 : c <-> c
(17) 5 : h <-> c	(38) 16 : r <-> r	(59) 28 : h <-> h
(18) 6 : e <-> c	(39) 16 : <-> c	(60) 28 : e <-> e
(19) 7 : r <-> c	(40) 17 : h <-> c	(61) 28 : r <-> r
(20) 8 : , <-> c	(41) 18 : e <-> c	(62) 28 : <-> c
(21) 9 : <-> c	(42) 19 : r <-> c	(63) 29 : h <-> c
		(64) 30 : e <-> c
		(**65**) 31 : r <-> c

Coût d'exécution

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

Le texte est balayée des indices 0 à n-p-1. A chaque étape, le mot entier est évalué (p). On obtient donc un double balayage n-p fois p → Coût : p(n-p)

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 :

- ✗ 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.
- ✗ 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.
- ✗ Si tous les caractères coïncident, on a trouvé une occurrence. Sinon, on note `j` l'indice de la première différence.
- ✗ 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 `motif[k] == texte[i+j]` et 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.

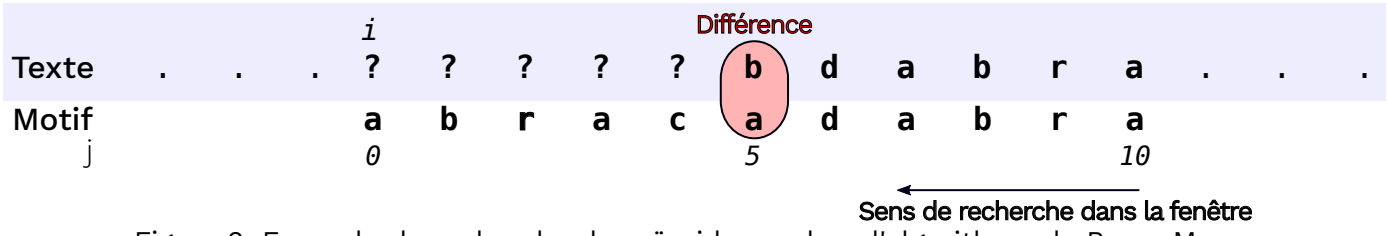

Figure 2: Exemple de recherche de coïncidence dans l'algorithme de Boyer Moore

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'`.

construire_table_decalage('banane') → tab_dec :

[{},
 {'b': 0},
 {'b': 0, 'a': 1},
 {'b': 0, 'a': 1, 'n': 2},
 {'b': 0, 'a': 3, 'n': 2},
 {'b': 0, 'a': 3, 'n': 4}]

construire_table_decalage('chercher') → tab_dec :

[{},
 {'c': 0},
 {'c': 0, 'h': 1},
 {'c': 0, 'h': 1, 'e': 2},
 {'c': 0, 'h': 1, 'e': 2, 'r': 3},
 {'c': 4, 'h': 1, 'e': 2, 'r': 3},
 {'c': 4, 'h': 5, 'e': 2, 'r': 3},
 {'c': 4, 'h': 5, 'e': 6, 'r': 3}]

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**

```
0 1 2 3 4 5 6 7 8 9 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3
                    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8
chercher, rechercher et chercher encore
chercher
0 1 2 3 4 5 6 7
```

```
Cpt   |  i  |texte |  decalage
( 8 )     0  :  c  ->  1
( 9 )     1  :  h  ->  8
( 10 )    9  :     ->  3
( 18 )    12 :  c  ->  1
( 19 )    13 :  h  ->  8
( 20 )    21 :  e  ->  3
( 28 )    24 :  c  ->  1
( 29 )    25 :  h  ->  8
```

Il y a 29 comparaisons dans cet algorithme comparé aux 65 comparaisons de la recherche naïve. Cet algorithme est très intéressant lorsque le motif (ou la fin du motif) n'apparaît pas souvent dans le texte.