ALG. Recherche textuelle

Le principe de ce chapitre est de savoir si un **motif** (mot, phrase) est **présent** dans un **texte**.

I/ Une première approche

Une première idée est de repérer les indices éventuels de la présence du motif recherché dans le texte, voici une animation :
https://glassus.github.io/terminale_nsi/T3_Algorithmique/3.3_Recherche_textuelle/cours/

On peut en déduire cet **algorithme** :

- On commence au début du texte.
- Tant que les lettres trouvées du texte correspondent au motif (on s'arrête à la longueur du motif).
    - Si le motif entier est présent, on ajoute l'indice de la première lettre.
- On passe à la lettre suivante du texte.
- On renvoie la liste d'indices.

Voici deux exemples de programmes :

Version avec les indices

```python
def recherche_naive(texte, motif):
    indices = [] # indices de présence du motif (1ere lettre)
    i = 0 # indice de chaque lettre du texte

    # Tant que l'on n'est pas au bout de texte en tenant
    # compte de la longueur du motif
    while i <= len(texte) - len(motif):
        j = 0
        while j < len(motif) and texte[i+j] == motif[j]:
            j += 1
        # Le motif entier est trouvé
        if j == len(motif):
            indices.append(i)
        # passage à la lettre suivante du texte
        i += 1

    return indices

# Jeu de tests
print(recherche_naive("coucou, c'est moi", "cou")) # Attendu : [0,3]
print(recherche_naive("coucou, c'est moi", "cout"))# Attendu : []
```

```
[0, 3]
[]
```

Version booléenne

```python
def recherche_naive(texte, motif):
    indices = [] # indices de présence du motif (1ere lettre)
    i = 0 # indice de chaque lettre du texte

    # Tant que l'on n'est pas au bout de texte en tenant
    # compte de la longueur du motif
    while i <= len(texte) - len(motif):
        j = 0
        while j < len(motif) and texte[i+j] == motif[j]:
            j += 1
        # Le motif entier est trouvé : on renvoie True
        if j == len(motif):
            return True
        # passage à la lettre suivante du texte
        i += 1

    return False # Pas de motif de trouvé

# Jeu de tests
print(recherche_naive("coucou, c'est moi", "cou")) # Attendu : True
print(recherche_naive("coucou, c'est moi", "cout"))# Attendu : False
```

```
True
False
```

On pourrait se demander quelle efficacité aurait ce type d'algorithme et si elle dépend de la longueur du motif.
Pour se faire, on va procéder à un jeu de test sur un chapitre l'œuvre des misérables de Victor Hugo, lien ici :
https://github.com/lmayer65/NSI_T/tree/main/Algorithmique/Algorithme%20de%20Boyer_Moore

Voici le programme :

```python
# Chargement du texte
with open("Les_Miserables.txt") as f:
    roman = f.read().replace('\n', ' ')
```

```python
import time

# Jeu de test (durée)
t0 = time.time()
motif = "maison"
print(recherche_naive(roman, motif))
print(time.time()-t0)

t0 = time.time()
motif = "La chandelle était sur la cheminée et ne donnait que peu de clarté."
print(recherche_naive(roman, motif))
print(time.time()-t0)

t0 = time.time()
motif = "parcoursup"
print(recherche_naive(roman, motif))
print(time.time()-t0)
```

Résultats

```
True
0.00760960578918457
False
0.5863466262817383
False
0.5610337257385254
```

On constate que la durée de l'exécution du programme **ne dépend pas** de la **longueur du motif** (on le voit ici avec les deux derniers tests) : ceci est logique d'un côté, on se contente de se déplacer d'une lettre à chaque fois.

II/ Algorithme de Boyer Moore

Une première piste est de partir de la **fin du motif** et non du début. Pourquoi ?
Eh bien, cela permet d'aller directement à un endroit du texte où le motif pourrait être présent.

En effet, si un caractère du texte n'est pas présent dans le motif, inutile de se déplacer d'un seul indice : on retomberait sur cette comparaison non valide dès le tour suivant (puisque l'on « remonte » le texte), autant se déplacer directement de la longueur du motif.

Programme modifié

```python
def par_la_fin(texte, motif):
    indices = []
    i = 0
    while i <= len(texte) - len(motif):
        # indice de la dernière lettre du motif
        j = len(motif)-1
        while j > 0 and texte[i+j] == motif[j]:
            j -= 1 # indice de la lettre précédente
        # Motif trouvé
        if j == 0:
            indices.append(i)
        i += 1

    return indices
```

Principe

Au lieu de ne se déplacer que d'un caractère si le motif n'est pas trouvé :

- Si un caractère du texte n'est pas présent dans le motif, on se déplace de la longueur du motif.
- Si ce caractère est présent, on cherche sa **dernière occurrence** dans le motif et on fait coïncider l'indice du texte avec celui dans le motif. On remarquera que s'il s'agit du dernier caractère du motif, inutile de se déplacer.

Voici une illustration de l'algorithme :
https://glassus.github.io/terminale_nsi/T3_Algorithmique/3.3_Recherche_textuelle/cours/

Au niveau du programme, on créé d'abord le **dictionnaire** des **indices** de chaque caractère du motif :

```
# Donne l'indice de la dernière occurrence de chaque
# lettre de 'mot'
def dico_lettres(mot):
    d = {}
    for i in range(len(mot)):
        d[mot[i]] = i
    return d
```

Et voici le programme (explications dedans) :

```
def BMH(texte, motif):
    dico = dico_lettres(motif)
    indices = []
    i = len(motif) - 1
    while i < len(texte):
        j = 0
        while j < len(motif) and motif[len(motif)-1-j] == texte[i-j]: #(1)
            j += 1
        if j == len(motif): #(2)
            indices.append(i-len(motif)+1)
            i += 1 #(3)
        else:
            if texte[i-j] in dico: #(4)
                i = max(i - j + len(motif) - dico[texte[i-j]] - 1, i+1) #(5)
            else:
                i = i - j + len(motif) #(6)

    return indices



#1. On remonte le motif à l'envers, tant qu'il y a correspondance et qu'on n'est pas arrivés au début du motif
#2. Si on est arrivés au début du motif, c'est qu'on a trouvé le mot.
#3. On a trouvé le motif, mais attention, il ne faut pas trop se décaler sinon on pourrait rater d'autres occurences
                # du motif (pensez à la recherche du motif «mama» dans le mot «mamamamama»). On se décale donc de 1.
#4. On s'est arrêté avant la fin, sur une lettre présente dans le mot : il va falloir faire un décalage intelligent.
#5. On décale juste de ce qu'il faut pour mettre en correspondance les lettres, en évitant le retour en arrière
                # (d'où le max pour se décaler au moins de 1 si j vaut 0 et dico[texte[i-j]] vaut len(motif) - 1  par ex)
#6. La lettre n'est pas dans le motif : on se positionne juste après elle.
```

Niveau **efficacité**, le pire des cas sera une comparaison systématique du motif entier (par exemple, chercher un motif « abbbbbbbb » dans un texte comme « bbb … bbb » soit une complexité de l'ordre de **longueur(texte)*longueur(motif)**.
A l'inverse, si un motif n'est pas présent, on se déplace à chaque fois de longueur(motif), la complexité devient intéressante, de l'ordre de **longueur(texte)/longueur(motif)** : on comprend alors que l'algorithme est plus efficace d'autant que le motif à rechercher est long !

Voici un test :

```
1  import time
2
3  # Jeu de test (durée)
4  t0 = time.time()
5  motif = "La chandelle était sur la cheminée et ne donnait que peu de clarté."
6  print(BMH(roman, motif))
7  print(time.time()-t0)
8
9  t0 = time.time()
10 motif = "parcoursup"
11 print(BMH(roman, motif))
12 print(time.time()-t0)
```

```
[]
0.05515098571777344
[]
0.1101067066192627
```

On note bien que le premier test est plus rapide que le second.

Les algorithmes de recherche textuelle sont notamment utilisés en bio-informatique.

Comme son nom l'indique, la bio-informatique est issue de la rencontre de l'informatique et de la biologie : la récolte des données en biologie a connu une très forte augmentation ces 30 dernières années. Pour analyser cette grande quantité de données de manière efficace, les scientifiques ont de plus en plus recourt au traitement automatique de l'information, c'est-à-dire à l'informatique. On peut citer l’analyse de l’ADN avec la recherche de séquences.