DIU-EIL – FOAD – Bloc 5

Recherche textuelle : Algorithme naïf - Algorithme de Boyer-Moore

Membres du groupe :

- MOUTON François
- QUIRIN Christophe

Prérequis élèves :

- Chaînes de caractères, tableaux, dictionnaires, module time, complexité

Point du programme étudié :

Recherche textuelle.	Étudier l'algorithme de Boyer-Moore pour la recherche d'un motif dans un texte.	L'intérêt du prétraitement du motif est mis en avant. L'étude du coût, difficile, ne peut être exigée.

Résumé de l'activité :

Cette activité se déroule en trois temps. La première partie d'une durée d'une heure est un cours/TD pouvant se faire en présentiel ou en distanciel avec des rappels de cours et la découverte de la recherche textuelle par la méthode find de Python puis la programmation d'un algorithme naïf. On compare d'ailleurs la rapidité d'exécution des deux fonctions afin de faire un rappel sur la complexité.

On étudie ensuite l'algorithme de Boyer-Moore version Horspool dont le but est d'en réussir l'implémentation de deux manières différentes en Python. Afin de bien appréhender la méthode, on commence par traiter à la main l'heuristique du Mauvais Caractère puis on construit les fonctions clés petit à petit.

La troisième partie correspond à la réalisation d'un exposé afin de préparer les élèves à l'épreuve du Grand Oral. Cet exposé de 10 minutes doit permettre à l'élève de réinvestir la bonne compréhension d'un algorithme puis d'arriver à le retranscrire oralement.

La dernière partie est adressé aux élèves à l'aise avec l'algorithme de Boyer-Moore et permet d'aller plus loin avec la deuxième heuristique du Bon Suffixe.

Le but de ce travail est de travailler sur la recherche textuelle et de faire prendre conscience aux élèves de l'intérêt d'accélérer une recherche pour trouver un motif dans un texte. De manière plus générale, l'élève doit sentir que la qualité de sa programmation permettra à ses solutions d'être plus efficaces. Il paraît également intéressant de prendre le temps de reformuler oralement la bonne compréhension de l'algorithme de Boyer-Moore afin de pouvoir l'expliquer simplement oralement. Enfin, la dernière partie de cette activité permet d'approfondir certains aspects pour les élèves les plus experts.

Lien pour accéder au Github dans lequel se trouve le notebook Jupyter (version élève et prof) (qui sera ensuite intégré à l'ENT via Capytale pour être consulté et modifié par les élèves puis possiblement évalué par le professeur).

https://mybinder.org/v2/gh/FcsM-lab/Foad_Bloc5/HEAD

A l'intérieur du Github se trouve normalement une branche correction où vous retrouvez certains éléments de correction, ils ont été intégrés à la fin de ce fichier.

Quelques copies d'écran du FOAD

Recherche Textuelle : Algorithme de Boyer-Moore

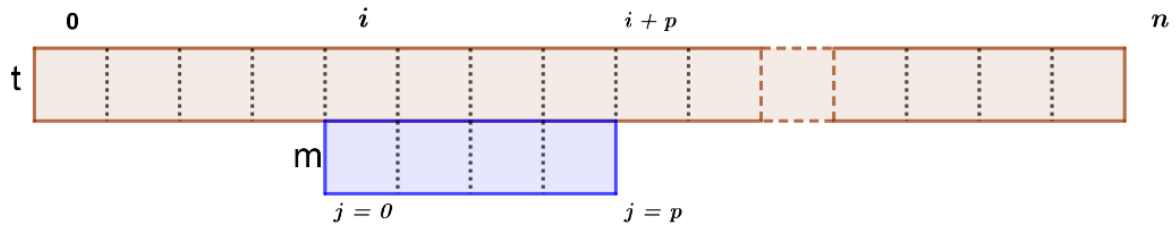
Application au chapitre

On note `m` de longueur p le motif que l'on cherche et `t` de longueur n le texte dans lequel on le recherche.

Une première remarque évidente est qu'il ne peut y avoir une occurence de `m` dans `t` que si $p \leq n$.

Plus précisément, une occurence de `m` dans `t` à la position i est contrainte par l'inégalité $0 \leq i \leq n - p$.

Il peut être utile de se représenter une occurence de `m` dans `t` à la position i comme ceci :



II. La méthode `find` de Python

Le site Gutenberg.org propose les grands classiques de la littérature qui sont tombés dans le domaine public. On peut par exemple y trouver le texte intégral du roman Le rouge et le noir de Stendhal dans l'encodage UTF-8 : Le Rouge et le Noir.
On pourra alors charger en mémoire ce roman par quelques lignes de code pour chercher ensuite si le motif `'Julien trembla'` apparaît quelque part dans le roman.

1. Prise en main

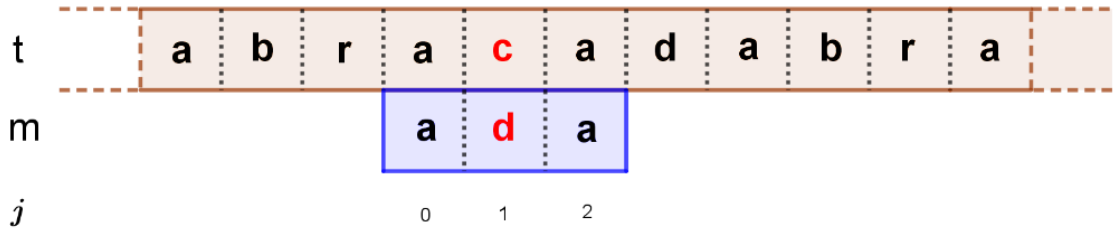
Exercice 1

a. Télécharger le livre Le rouge et le noir et enregistrer-le dans votre répertoire Recherche Textuelle de NSI.

b. Téléverser ce fichier txt dans votre Jupyter Notebook à l'aide de l'upload présent dans File>Open...

c. Expliquer brièvement ce que font les lignes de code ci-dessous.

```
1 fichier = open('LeRougeEtLeNoir.txt', 'r')
2 stendhal = fichier.read()
```



On est à nouveau en situation d'échec. Il faut donc décaler notre motif, et on effectue donc $i = i + 1$ et $j = 0$.

1. Implémentation de l'algorithme naïf

Exercice 3

2. Complexité temporelle

L'objectif premier d'un calcul de complexité algorithmique est de pouvoir comparer l'efficacité d'algorithmes résolvant le même problème. Dans une situation donnée, cela permet donc d'établir lequel des algorithmes disponibles est le plus optimal.

Ce type de question est primordial, car pour des données volumineuses la différence entre les durées d'exécution de deux algorithmes ayant la même finalité peut être de l'ordre de plusieurs jours.

Avant de se pencher de manière théorique sur la complexité temporelle, nous allons comparer brièvement la complexité des algorithmes de recherche vus depuis le début de ce cours. (Re-)Parlons maintenant du module time de Python, testez le programme suivant :

e [6]:
```
import time
debut = time.time()
```

e [7]:
```
# On attend quelques secondes avant de taper la commande suivante
fin = time.time()
print(debut < fin)
fin - debut # Combien de secondes entre debut et fin ?
```

```
True
```

ut[7]:
```
1.0350589752197266
```

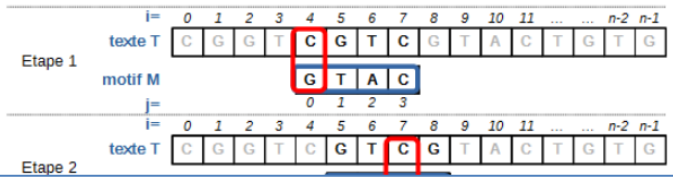
IV. Accélération de recherche

1. Hypothèse

Posons l'hypothèse que chaque lettre du motif n'apparait qu'une seule et unique fois dans ce motif.
Dans une boucle de vérification sur le motif, lorsque le test est négatif (T[*i*+*j*] != M[*j*]):
- La partie du motif qui précède ne peut contenir cette lettre qu'une seule fois dans une combinaison unique
- Puisque le motif qui précède concorde avec des lettres combinées de manière unique, un nouveau motif recommencera au mieux à l'emplacement du test.

Dans tous les cas le saut suivant serait au minimum de 1, donc rejoindrait la configuration de la recherche naïve. On peut donc décider de faire un saut égal à l'index du motif de l'échec j avec un saut de 1 au minimum.

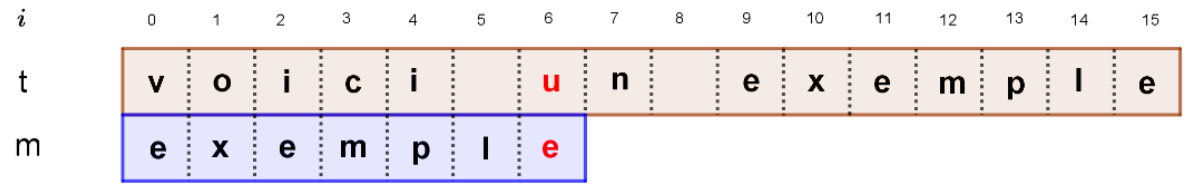
Illustration :



V) Boyer-Moore version simplifiée de Horspool

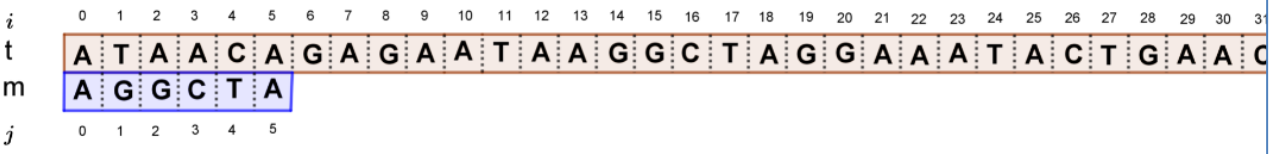
1. Une première idée pour accélérer la recherche

- L'idée est d'améliorer la recherche en utilsant certaines **heuristiques** dont nous détaillerons la première.
- L'algorithme est aussi à fenêtre glissante comme dans la recherche naïve mais la comparaison du motif avec le texte, `m[0:p-1]` avec `t[i:i+p-1]`, se fait **de droite à gauche**, en commençant par la fin !
- La première heuristique, souvent appelée celle du "mauvais caractère", va tirer parti du parcours inversé en testant d'abord s'il y a correspondance pour le dernier caractère du motif.
- Par exemple dans la recherche suivante de `m` dans `t` :



Exercice 4 : Un exemple à faire à la main

Appliquer l'algorithme vu précedemment à l'exemple suivant en déroulant les étapes à la main et en commençant à appréhender son codage futur.



1. Que faire si la lettre du texte n'apparaît pas dans le motif ?
2. Dans quel cas le décalage est-il le moins important ?
3. Chaque décalage fait est-il unique ?
4. Que pourrait-on faire avant de commencer les comparaisons ?

3. Prétraitement

5. Implémentation de l'algorithme de Boyer-Moore

Exercice 7

Commençons par implémenter une fonction `Dico_droite` qui calculera un dictionnaire dont les clés sont les caractères du motif et les valeurs la position la plus à droite du caractère.

[17]:
```
def Dico_Droite(motif):
# remplit (partiellement) un dictionnaire pour donner les positions les plus à droite de chaque caractère
    pass
```

C'est un bon début mais on voudrait calculer le décalage de la fenêtre même quand le caractère qui provoque l'échec n'apparaît pas dans le motif.

Coder maintenant la fonction `droite` qui renvoie −1 si le caractère n'est pas dans le dictionnaire `aDroite`.

[26]:
```
def droite(c,motif):
    # renvoie -1 si c n'est pas dans le motif ou sinon aDroite[c]
    pass
```

Coder maintenant une fonction `correspondance` où le parcours du motif se fait de droite à gauche et où le calcul du décalage se déduit du dictionnaire `aDroite`. Cette fonction prendra en paramètres le texte, le motif et l'indice i. Elle renverra sous la forme d'un tuple si les caractères correspondent à l'indice i et le décalage nécessaire s'il y a "mismatch". Attention à avoir toujours un décalage minimum quand les caractères ne correspondent pas !

VII. En route vers le grand oral

Il s'agit de préparer un petit exposé d'environ 10 minutes pour présenter l'un des différents algorithmes de recherche suivant ou un problème qui s'y rattache :

- recherche naïf
- Boyer-Moore
- Knuth-Morris-Pratt
- distance de Levenshtein

VIII. Pour aller plus loin : méthode du bon suffixe

L'algorithme de Boyer-Moore dispose d'une deuxième heuristique appelée du "bon suffixe", un tableau BS est utilisé dont chaque entrée BS[i] contient le décalage du motif en cas d'erreur de correspondance en position $i-1$, si le suffixe (la fin) du motif commençant position i correspond.

Exercice

En reprenant l'exemple précédent, appliquez à la main la règle du Bon Suffixe uniquement puis les règles Mauvais Caractère et Bon Suffixe. A t-on gagné en nombre de comparaison ?

Quelques éléments de correction :

```python
def occurence(m,t,i):
    '''indique s'il y a une occurence de la chaîne m dans la chaîne t à la position i'''
    if i<0 or i>len(t)-len(m):
        return False
    for j in range(len(m)):
        if t[i+j]!=m[j]:
            return False
    return True

def recherche_Naif(m,t):
    '''affiche toutes les occurences de m dans t'''
    for i in range(0,len(t)-len(m)+1):
        if occurence(m,t,i):
            print("occurence à la position",i)
```

```python
def recherche_Am(m,t):
    '''affiche toutes les occurences de m dans t
    version améliorée avec caracteres uniques dans le motif'''

    if len(t) < len(m):
        return -1
    i = 0 #compteur dans texte t
    while i <= len(t)-len(m)+1:
        j=0
        for j in range(len(m)):
            if m[j] != t[i+j]:
                i += max(1, j)
                break
            elif j == len(m)-1 and m[j] == t[i+j]:
                print("occurence à la position",i)
                i+= len(m)
```

```python
def Dico_Droite(motif):
# remplit (partiellement) un dictionnaire pour donner les positions les plus à droite
    p=len(motif)
    aDroite = {}
    for j in range(p):
        aDroite[motif[j]] = j
    return aDroite

def droite(c,motif):
    aDroite=Dico_Droite(motif)
    # renvoie -1 si c n'est pas dans le motif ou sinon aDroite[c]
    if c in aDroite.keys():
        return aDroite[c]
    else:
        return -1
```

```python
def correspondance(texte, motif,i):
    p=len(motif)
    for j in range(p - 1, -1, -1): # j varie de p-1 à 0 inclus en décroissant
        x = texte[i + j]
        if x != motif[j]:
            decalage = max(1, j - droite(x,motif))
            return (False, decalage)
    return(True, 0)

def Recherche_BM(texte, motif):
    n = len(texte)
    p = len(motif)
    Dico_Droite(motif)
    i = 0
    while i + p <= n:
        ok, decalage = correspondance(texte, motif, i)
        if ok:
            print('occurence en', i)
            i+=1
        else:
            i = i + decalage
```

```python
def table_MC(m):
    '''construit la table de décalages de Boyer-Moore :
    '''
    MC={}
    for i in range(len(m)):
        MC[m[i]] = len(m)-i-1
    return MC

def recherche_BM2(motif,texte):
    '''affiche toutes les occurences de m dans t avec l'algorithme de Boyer-Moore'''
    table_Decalage=table_MC(motif)
    i=0
    while i <= len(texte)-len(motif):
        for j in range(len(motif)-1, -1, -1):

            if motif[j] == texte[j+i]:
                if j == 0:
                    print("occurence à la position :", i)
                    i += len(motif)
            else:
                if texte[i+j] in table_Decalage:
                    if table_Decalage[texte[i+j]] == 0:
                        i += 1
                    else:
                        i += table_Decalage[texte[i+j]]
                else:
                    i += len(motif)
                break #sortie de la boucle for
```