TP 2 : Algorithmique du texte

Remarque pratique : si l'éditeur `geany` est utilisé, aller dans `Edit->Preferences->Editor->Indentation` et choisir `spaces` au lieu de `tabs`, pour éviter des problèmes avec l'indentation.

1 Quelques nouvelles notions de Python

On commencera les fichiers par la ligne suivante :

```
# -*- coding: utf-8 -*-
```

Afin de pouvoir écrire des caractères accentués dans les chaînes de caractères et commentaires.

1.1 Chaînes de caractères

Les chaînes de caractères en Python se manipulent comment les tableaux. Les fonctions suivantes seront utilisées :

```
str = 'Hello, world!' # Définir une chaîne de caractères
str[1] = 'a' # erreur => chaînes immutables en python
code_ascii = ord('a') # 97 (nombre entre 0 et 255)
longueur = len(str) # 13 (la longueur de la chaîne)
```

La fonction `ord` associe à chaque caractère un nombre entre 0 et 255 (cela correspond à un octet). *Remarque :* ce n'est pas tout à fait exact, car certains caractères sont en fait constitués de plusieurs octets, mais cela n'aura pas d'importance dans ce TP, car `python2` (la version utilisée), traite les chaînes de caractères comme des tableaux d'octets.

1.2 Quelques éléments utiles

On pourra utiliser la fonction `range` en décroissant, par exemple `range(5,-1,-1)` décrit les nombres de 5 à 0 en décroissant de 1 à chaque étape.

Le mot-clé `break` permet de sortir d'une boucle `for` ou `while` prématurément.

La syntaxe `[5] * 3` permet de générer le tableau `[5,5,5]`.

2 Recherche d'un mot dans un texte

L'objectif de ce TP est de programmer plusieurs algos pour rechercher un mot `mot` dans un texte `text`, i.e trouver une chaîne de caractère dans une autre. On notera `wl` la longueur du mot. On notera dans l'énoncé `text[i:j]` pour parler du sous-mot de `text` entre les indices `i` (inclus) et `j` (non inclus), comme en Python.

2.1 Algorithme naïf

Écrire une fonction `word_occurrences(word, text)`, qui renvoie le nombre d'occurrences du sous-mot `word` dans le texte `text`, en utilisant l'algorithme suivant :

— Pour `i` parcourant les indices du texte `text`, on teste si les mots `text[i:i+wl]` et `word` sont égaux.

— Si le test est positif, avancer `i` de `wl` (de sorte par exemple que `'aa'` apparaisse deux fois dans `'aaaa'` et non trois). Sinon, on décale `i` de 1.

Effectuer les tests suivants :

```
print(word_occurrences('aa','aaaa')) # résultat: 2
print(word_occurrences('sour','une chauve-souris sourde ne sourit pas'))
  # résultat: 3

# Test sur le premier tome des Misérables: le texte du tome est lu
# dans la variable "text". Fichier format texte disponible à l'adresse:
# http://perso.eleves.ens-rennes.fr/~yfern356/miserables-tome1.txt
with open('miserables-tome1.txt') as fh:
    text = fh.read()
print(word_occurrences('Valjean', text)) # résultat: 196

# Pour tester le temps d'exécution:
def test1():
    word_occurrences('Valjean', text)
from timeit import timeit
# Afficher en secondes le temps d'exécution de 5 appels à test1
print(timeit('test1()', 'from __main__ import test1', number = 5))
```

```
aabbabcaabc # texte
bca         # mot 'bca' en position initiale (i == 2)
  bca       # 'b' != 'a', on avance pour mettre le b du mot
            #    devant celui du texte
     bca    # 'a' == 'a', mais 'bba' != 'bca', et 'a' n'apparaît pas
            #    dans 'bc' donc on décale de la longueur du mot
     ===    # on a égalité ici
        bca # pas d'égalité: au total une seule occurrence
```

FIGURE 1 – Boyer-Moore sur un exemple

2.2 Algorithme de Boyer-Moore (simplifié, version d'Horspool)

Écrire une fonction `BMsimple` qui calcule la même chose que `word_occurrences`, mais en utilisant l'algorithme suivant :

— Comme dans l'algo précédent, on a un indice `i` qui parcourt le texte.

— On compare le dernier caractère du mot avec le caractère `text[i]` en `i`-ème position dans le texte.

- S'ils sont égaux, on regarde si l'on a une occurrence du mot (comme dans l'algo naïf), et si c'est le cas, on compte une occurrence, et on avance `i` de `wl`.
- Dans tous les autres cas, on avance `i` de sorte que la dernière lettre du mot `word[0:wl-1]` qui soit égale à `text[i]` se retrouve en face de `text[i]`, si elle existe, et sinon on avance `i` de `wl`. *Remarque :* dans le cas d'égalité de la dernière lettre (peu fréquent) mais non du mot, on peut se contenter en pratique d'incrémenter `i` de 1 comme dans le cas naïf.

Voir l'exemple de la Figure 1. Tester la fonction `BMsimple`, et comparer avec l'algorithme naïf.

Une technique courante en programmation pour améliorer certains algorithmes est de faire un pré-traitement sur l'entrée qui permet de réaliser le calcul de façon plus performante, avec parfois un compromis entre temps d'exécution et occupation mémoire.

Écrire une nouvelle fonction `BMsimple_with_table` implémentant le même algorithme que `BMsimple`, mais utilisant un tableau `delta` de 256 éléments précalculé pour accélérer le calcul de décalage, dont la définition est la suivante : `delta[ord(c)]` renvoie la distance de la dernière occurrence de `c` dans `word[0:wl-1]` (donc privé du dernier indice `wl-1`) à la fin du mot, ou `wl` si le caractère n'est pas présent dans le mot.

Tester la fonction `BMsimple_with_table`, et comparer les performances avec les précédentes sur le texte des Misérables. Chercher aussi des chaînes de caractères où Boyer-Moore donne des résultats similaires à l'algorithme naïf (des cas pathologiques).

Remarque culturelle : L'algorithme de Boyer-Moore est très utilisé, du fait de ses exellentes performances en pratique (sous-linéaire), malgré des performances médiocres dans des cas pathologiques.