Lycée Louis-Le-Grand, Paris
MPSI 4 – Informatique pour tous
A. Troesch, J.-P. Becirspahic

2013/2014

Corrigé du TP n° 6

1. Méthode directe

a. La fonction suivante détermine si un mot est en tête du suffixe de rang k d'un texte. Notez l'usage du mécanisme de rattrapage d'une exception, qui permet de ne pas se préoccuper de savoir si le suffixe en question est suffisamment long pour contenir le mot recherché.

```
def enTeteDeSuffixe(mot, texte, k):
    try:
        for i in range(len(mot)):
            if texte[k+i] != mot[i]:
                return False
        return True
    except IndexError:
        return False
```

b. On en déduit immédiatement la version naïve de recherche d'un mot dans un texte : il suffit de tester tous les suffixes à l'aide de la fonction précédente.

```
def rechercherMot(mot, texte):
    for k in range(len(texte)):
        if enTeteDeSuffixe(mot, texte, k):
            return True
    return False
```

c. De même, pour déterminer le nombre d'occurrences :

```
def compterOccurrences(mot, texte):
    s = 0
    for k in range(len(texte)):
        if enTeteDeSuffixe(mot, texte, k):
            s += 1
    return s
```

Fréquence d'apparition

d. Pour remplir le dictionnaire avec les lettres et leurs fréquences, il faut parcourir les lettres du texte en distinguant deux cas : lors de la première apparition d'une lettre il faut créer une nouvelle entrée ; lors des apparitions suivantes il faut itérer la valeur déjà associée à cette lettre.

```
def frequenceLettre(texte):
    d = {}
    for x in texte:
        if x in d:
            d[x] += 1
        else:
            d[x] = 1
    return d
```

Si on préfère rattraper une exception :

```
def frequenceLettre(texte):
    d = {}
    for x in texte:
        try:
            d[x] += 1
        except KeyError:
            d[x] = 1
    return d
```

On peut enfin utiliser la méthode `get` qui possède un second paramètre qui est renvoyé lorsque la clé n'est pas présente dans le dictionnaire :

```
def frequenceLettre(texte):
    d = {}
    for x in texte:
        d[x] = d.get(x, 0) + 1
    return d
```

e. Même chose avec les bigrammes :

```
def frequenceBigramme(texte):
    d = {}
    for k in range(0, len(texte) - 1):
        mot = texte[k: k + 2]
        d[mot] = d.get(mot, 0) + 1
    return d
```

2. Tableau des suffixes

On obtient le tableau des suffixes en triant ces derniers par ordre lexicographique : nous allons donc composer les éléments du tableau $[0, 1, 2, \ldots, n-1]$ par la fonction $k \mapsto \texttt{texte}[k:]$ avant de procéder au tri lexicographique.

```
def calculerSuffixes(texte):
    def f(k):
        return texte[k:]
    tabS = list(range(len(texte)))
    tabS.sort(key = f)
    return tabS
```

Plus élégamment, le mot-clé `lambda` permet de définir une fonction anonyme :

```
def calculerSuffixes(texte):
    tabS = list(range(len(texte)))
    tabS.sort(key = lambda k: texte[k:])
    return tabS
```

3. Exploitation du tableau des suffixes

a. On obtient la fonction de comparaison entre un mot et un suffixe du texte en adaptant la définition de la fonction `enTeteDeSuffixe` :

```
def comparerMotSuffixe(mot, texte, k):
    try:
        for i in range(len(mot)):
            if mot[i] < texte[k+i]:
                return -1
            elif mot[i] > texte[k+i]:
                return 1
        return 0
    except IndexError:
        return 1
```

b. On procède alors à la recherche par dichotomie dans le tableau des suffixes :

```
def rechercherMot2(mot, texte, tabS):
    i, j = 0, len(texte)
    while i < j:
        k = (i + j) // 2
        r = comparerMotSuffixe(mot, texte, tabS[k])
        if r == 0:
            return True
        elif r < 0:
            j = k
        else:
            i = k + 1
    return False
```

c. Si n désigne la longueur de la chaîne de caractères `texte`, le nombre de comparaisons entre mots de la fonction `rechercherMot` est égal au nombre d'appels à la fonction `enTeteDeSuffixe` ; il s'agit donc d'un $O(n)$.

De même, le nombre de comparaisons entre mots de la fonction `rechercherMot2` est égal au nombre d'appels à la fonction `comparerMotSuffixe` ; il s'agit cette fois d'un $O(\log n)$.

Évidemment, le coût total ne prend pas en compte le coût de la création du tableau des suffixes, mais dans le cas d'un moteur de recherche internet destiné à faire de très nombreuses recherches dans les pages qu'il aura indexées, il est intéressant de créer ce tableau des suffixes au moment de l'indexation de la page en question car cela permet ensuite de répondre beaucoup plus rapidement aux requêtes des utilisateurs.

4. Nombre d'occurrences d'un mot dans un texte

a. Pour obtenir le rang du premier suffixe dans `tabS`, on adapte la fonction `rechercherMot2` en poursuivant la recherche même si on a trouvé un suffixe qui convient :

```
def rechercherPremierSuffixe(mot, texte, tabS):
    i, j = 0, len(texte)
    m = -1
    while i < j:
        k = (i + j) // 2
        r = comparerMotSuffixe(mot, texte, tabS[k])
        if r == 0:
            m = j = k
        elif r < 0:
            j = k
        else:
            i = k + 1
    return m
```

b. On procède bien évidemment de même pour le plus grand des suffixes :

```
def rechercherDernierSuffixe(mot, texte, tabS):
    i, j = 0, len(texte)
    m = -1
    while i < j:
        k = (i + j) // 2
        r = comparerMotSuffixe(mot, texte, tabS[k])
        if r == 0:
            m, i = k, k + 1
        elif r < 0:
            j = k
        else:
            i = k + 1
    return m
```

c. Le nombre d'occurrences est alors immédiat :

```
def compterOccurrences2(mot, texte, tabS):
    p = rechercherPremierSuffixe(mot, texte, tabS)
    if p == -1:
        return 0
    d = rechercherDernierSuffixe(mot, texte, tabS)
    return d - p + 1
```

d. Nous pouvons observer que les suffixes qui débutent par le même mot de k lettres sont voisins dans le tableau des suffixes, il suffit donc de parcourir ce dernier pour remplir le dictionnaire des fréquences :

```
def frequenceKgramme2(texte, tabS, k):
    i = 0
    d = {}
    while i < len(texte):
        if tabS[i] + k <= len(texte):
            mot = texte[tabS[i]: tabS[i] + k]
            j = rechercherDernierSuffixe(mot, texte, tabS)
            d[mot] = j - i + 1
            i = j + 1
        else:
            i += 1
    return d
```

On peut observer que cette fonction ne remplit le dictionnaire qu'une fois pour chaque mot, ce qui ne serait pas le cas si on parcourait l'ensemble des sous-mots de longueur k.