Lycée Louis-Le-Grand, Paris
MPSI 4 – Informatique pour tous
D. Cazor, A. Troesch

2014/2015

TP n° 7 : Recherche d'un mot dans un texte

Correction du problème –
Le but de ce problème est d'étudier des algorithmes de recherche de mots (séquences) dans un texte donné. Dans tout l'énoncé, les variables `texte` et `mot` désigneront deux chaînes de caractères. Notre but est de trouver une ou plusieurs occurrences de `mot` dans `texte`. Nous dirons que `mot` a une occurrence dans `texte` à la place i si pour tout $k \in [\![0, \text{len}(\texttt{mot}) - 1]\!]$, on a l'égalité des caractères $\texttt{texte}[i+k] = \texttt{mot}[k]$.

Partie I – Méthode directe de recherche d'un mot dans un texte

1. On compare successivement les lettres de `mot`, et celles de `texte` à partir de la position i. Dès qu'on n'a plus concordance, on s'arrête. Si on a concordance jusqu'en fin de `mot`, on a une occurrence en position i.

```
def isprefixe(texte,mot,i):
    """Vérifie si mot a une occurrence dans texte en position i"""
    B = True
    j = 0
    while (j < len(mot)) and B:
        if texte[i+j] != mot[j]:
            B = False
        j +=1
    return B
```

2. On énumère tous les positionnements possibles du mot dans le texte, et on teste s'il a une occurrence en cette position grâce à la fonction précédente :

```
def cherche_occurrences(texte, mot):
    """Donne la liste de toutes les occurrences de mot dans texte"""
    occ = []  # liste des occurrences
    for i in range(len(texte)-len(mot)+1):
        if isprefixe(texte,mot,i):
            occ.append(i)
    return occ
```

Partie II – Utilisation et test de rapidité de la fonction `cherche_occurrence`

1. Plusieurs possibilités : on peut utiliser le code ascii, et rcuérer les caractères successifs grâce à leur numérotation (en utilisant `char`). On peut aussi créer une chaîne de caractères représentant les lettres de l'alphabet, qui permet de faire la correspondance entre lettre et numéro (via sa position) :

```
def texte_alea(n):
    """Création d'un mot aléatoire de longueur n"""
    lettres = 'abcdefghijklmnopqrstuvwxyz'
    texte = ''
    for i in range(n):
        texte = texte + lettres[random.randint(0,25)]
    return texte
```

Une variante est l'utilisation de la fonction `choice` du module `random`. Supposons ce module importé via `import random` :

```
def texte_alea2(n):
    """Création d'un mot aléatoire de longueur n"""
    texte= ''
    for i in range(n):
        texte += random.choice('abcdefghijklmnopqrstuvwxyz')
    return texte
```

Des tests semblent indiquer que cette dernière fonction serait un peu plus rapide que la précédente ; c'est donc celle-ci que nous utiliserons par la suite.

2. On compte le nombre d'occurrences de `'llg'` dans m mots de longueur n créés aléatoirement. On compte ces occurrences en additionnant les longueurs des listes d'occurrences obtenues par la fonction `cherche_occurrences`.

```
def nombre_moyen_apparition(n,mot,m):
    """ nombre moyen d'apparition du mot dans un texte aléatoire de
    longueur m; la moyenne est faite sur m expériences """
    compteur = 0
    for i in range(m):
        texte = texte_alea2(n)
        compteur += len(cherche_occurrences(texte,mot))
    return compteur / m
```

Évidemment, plus m est grand, plus l'estimation va être bonne (tendant, lorsque m tend vers $+\infty$, vers l'espérance mathématique). Par exemple, l'instruction

```
print('{}'.format(nombre_moyen_apparition(10000,'llg',1000)))
```

retourne la valeur 0.575.

Sans faire un calcul précis, on peut avoir une idée de l'ordre de grandeur de l'espérance mathématique. Même si les choses sont un peu plus compliquées que cela si on veut une expression exacte, on peut considérer que la probabilité que `llg` apparaisse en position i est $p = \frac{1}{26^3}$ (le choix des 3 lettres), et qu'il y a 9998 positions possibles. Le nombre moyen d'occurrences suit alors grossièrement une loi binomiale de paramètres $(9998, p)$. L'espérance en est $np = 9998 \times \frac{1}{26^3}$, ce qui donne 0.569, valeur assez proche de celle obtenue empiriquement

Ce calcul n'est pas tout-à-fait exact, car on n'a pas tenu compte des chevauchements des mots lors de l'énumération des positions possibles : ces chevauchements rendent les événements élémentaires « le mot apparaît en position i » dépendants les uns des autres. Sans l'hypothèse d'indépendance, la loi du nombre d'occurrences n'est donc pas tout-à-fait une loi binomiale (mais cela s'en approche tout de même beaucoup, pour un petit mot)

On aura l'occasion de voir plus tard en cours de probabilité comment faire un calcul exact.

On peut faire le graphe du nombre d'apparitions en fonction de la longueur du texte.

```
def graphe_nombre_apparitions(mot,m):
    """ graphe du nombre moyen d'apparition de mot dans un texte aléatoire,
    suivant la longueur de ce texte """
    X=[]
    Y=[]
    for n in range (200,30000,200):
        X.append(n)
        Y.append(nombre_moyen_apparition(n,mot,m))
    plt.plot(X,Y)
    plt.savefig('pb_py003-fig1.eps')
    plt.show()
```

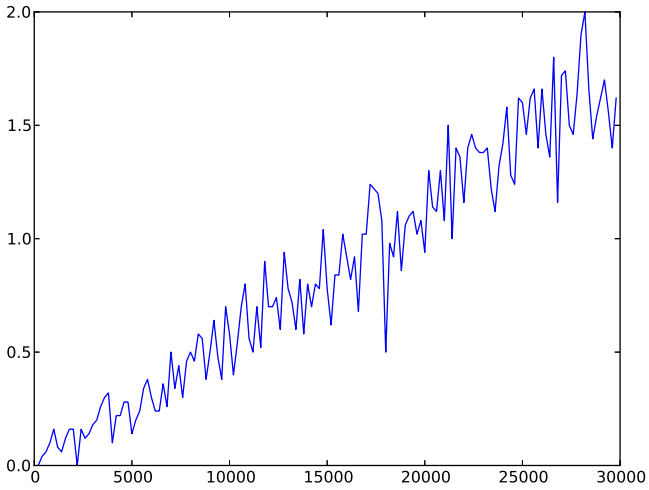

FIGURE 1 – Nombre d'apparition de `llg` dans un mot de longueur n

On obtient le graphe de la figure 1.

Ce graphe semble indiquer une linéarité (ce qui semble logique) et une certaine variabilité (les moyennes à chaque étape ont été faites sur 50 expériences, ce qui n'est pas suffisant pour avoir une estimation très précise, mais au-delà, le temps de calcul commence à être très long).

3. C'est à peu près la même chose, à part qu'on compte cette fois pour 1 toute liste contenant au moins une occurrence de `llg`.

```
def frequence_texte_avec_mot(n,mot,m):
    """ fréquence d'apparition d'au moins une occurrence de mot dans un
    texte aléatoire de longueur n. La moyenne est faite sur m expériences """
    compteur = 0
    for i in range(m):
        texte = texte_alea2(n)
        if  len(cherche_occurrences(texte,mot)) != 0:
            compteur += 1
    return compteur / m
```

Par exemple, l'instruction

```
print('{}'.format(frequence_texte_avec_mot(20000,'llg',100)))
```

retourne la valeur 0.38. Ici aussi, il faudrait augmenter la valeur de m pour obtenir une moyenne ayant une plus forte probabilité d'être bonne (plus précisément, on pourrait définir un intervalle de confiance pour notre mesure, avec un certain taux de confiance : augmenter m permet de réduire la longueur de l'intervalle de confiance, pour le même taux de confiance, ou alors d'augmenter le taux de confiance, pour le même intervalle).

Estimer la probabilité qu'un mot de longueur 10000 contienne au moins une occurrence du mot « llg »

4. Sans difficulté, en utilisant le module `time` (à importer) pour mesurer le temps d'exécution.

```
def teste_direct(N,n,p):
    texte = texte_alea(N)
    debut = time.time()
    for i in range(p):
        mot = texte_alea(n)
        cherche_occurrences(texte,mot)
    fin = time.time()
```

```
    return fin - debut

print(teste_direct(10000,5,100))
print(teste_direct(20000,5,100))
```

5. On obtient les valeurs suivantes :

```
0.7332489490509033
1.4677715301513672
```

ce qui va aussi dans le sens de la linéarité en N, que nous avons déjà suggérée plus haut.

Partie III – Utilisation d'une table des suffixes

On se propose, moyennant un traitement préalable du texte `texte`, d'accélérer la recherche d'un mot dans `texte`. Ce traitement préalable consiste en la création d'un tableau des suffixes. Les suffixes de `texte` sont toutes les chaînes terminales `texte[i:]`, pour $i \in [\![0, \texttt{len(texte)} - 1]\!]$. La table des suffixes consiste en la liste des suffixes de `texte` rangés par ordre alphabétique. Afin de ne pas utiliser trop de mémoire, ces suffixes sont représentés dans le tableau par leur indice initial dans `texte`. Par exemple, pour le texte 'abracadabra', les suffixes sont, rangés dans l'ordre croissant :

['a','abra','abracadabra','acadabra','adabra','bra','bracadabra','cadabra','dabra','ra','racadabra']

La table des suffixes sera alors la liste des indices initiaux de ces suffixes, à savoir :

$$[10, 7, 0, 3, 5, 8, 1, 4, 6, 9, 2].$$

1. Une clé de tri est une fonction f selon les valeurs de laquelle on tri les éléments d'un tableau : autrement dit, on trie les éléments a_i d'un tableau par ordre croissant (ou décroissant suivant le paramètre choisi) des $f(a_i)$. Ici, on veut trier les indices i (indiquant le début des suffixes) suivant l'ordre alphabétique des suffixes correspondants, donc des `texte[i:]`. Python sait comparer des chaînes de caractères (par ordre alphabétique).

   La méthode `sort` associée aux listes prend en argument optionnel une clé de tri (donc une fonction). Par défaut, il s'agit de la fonction identité. Pour la modifier, on précise en argument la nouvelle clé sous la forme `key = nouvelle_clé`, où `nouvelle_clé` est une fonction.

   Ainsi, on obtient la fonction suivante :

```
def table_suffixe(texte):
    suffixes = list(range(len(texte)))
    suffixes.sort(key = lambda i: texte[i:])
    return suffixes
```

   Le terme `lambda` permet de définir une fonction localement (sans la définir globalement par `def`). La syntaxe est :

```
  lambda variable: expression(variable)
```

   pour définir la fonction dont la variable (de type quelconque, par exemple réel ou tuple) est `variable` et définie par l'expression `expression(variable)` Par exemple

```
  lambda x: x**2
```

   définit la fonction $x \mapsto x^2$.

2. On fait une recherche dichotomique, le tableau étant trié suivant la clé `cl` : on coupe le tableau au milieu (à l'incide c), et on compare `cl(T[c])` à la valeur recherchée.

```
def recherche_dichotomique(table,cle,mot):
    a = 0
```

```
    b = len(table)-1
    while b-a > 1:
        if cle(table[a]) >= mot:
            return a
        c = (a+b) // 2
        if cle(table[c]) > mot:
            b = c
        else:
            a = c
    return a+1
```

3. Si `mot` a une occurrence dans le texte en position i, il est préfixe de `T[i:]`. Le mot `mot` se positionne alors par la fonction précédente `recherche_dichotomique` juste avant les suffixes commençant par `mot` (s'il y en a). Ces suffixes représentent toutes les occurrences de `mot` dans le texte, et sont rangées de façon contiguë dans le tableau des suffixes. On trouve donc par la fonction précédente l'indice i, puis à partir de cet indice, on teste si `mot` est en début des différents suffixes qui suivent, jusqu'à obtenir un suffixe ne commençant pas par mot. On aura alors l'ensemble des occurrences.

```
def recherche_par_suffixe(texte,suffixes,mot):
    occ = []
    i = recherche_dichotomique(suffixes,lambda j: texte[j:],mot)
    while (i < len(suffixes)) and isprefixe(mot,texte[suffixes[i]:]):
        occ.append(suffixes[i])
        i += 1
    return occ
def recherche_avec_suffixes(texte,suffixes,mot):
    i = recherche_dichotomique(suffixes,lambda j: texte[j:],mot)
    if (i < len(suffixes)) and isprefixe(mot,texte[suffixes[i]:]):
        return(suffixes[i])
```

4. On constate que les occurrences sont les mêmes (fort heureusement), mais pas dans le même ordre (ce qui est normal, puisque dans le deuxième cas, les occurrences sont données non pas par ordre d'apparition, mais par ordre alphabétique des suffixes correspondants).
5. On repète la fonction précédente, une fois un texte créé (afin de ne pas tenir compte du temps de fabrication du texte et de la table des suffixes, qui peut être assez important pour des grandes valeurs de N).

```
def teste_recherche_par_suffixes(N,n,p):
    texte = texte_alea(N)
    suffixes = table_suffixe(texte)
    debut = time.time()
    for i in range(p):
        mot = texte_alea(n)
        liste_occurrence_avec_suffixe(texte,suffixes,mot)
    fin = time.time()
    return fin - debut
```

6. On obtient les résultats suivants, comparés à ceux obtenus par la méthode directe :

```
Par la première méthode (recherche directe):
Pour N=10000, n=5, p=100: 0.7154421806335449
Pour N=20000, n=5, p=100: 1.4257688522338867
Avec la table des suffixes:
Pour N=10000, n=5, p=100: 0.004532575607299805
```

```
Pour N=20000, n=5, p=100: 0.00660395622253418
Pour N=50000, n=5, p=100: 0.0086517333984375
```

Le gain est évident. Par ailleurs, la comparaison des cas $N = 10000$ et $N = 50000$ suggère bien une complexité meilleure que linéaire.

7. La recherche dichotomique est en temps logarithmique. Ainsi, en négligeant le temps nécessaire à la recherche des occurrences successives dans le texte (ou en ne considérant que la première occurrence), on obtient une recherche en temps logarithmique (en considérant la taille du mot comme constante).

Ainsi, le traitement préalable de la table des suffixes ralentit globalement le temps de recherche, si le but est de faire une seule recherche (ce tri initial ne pouvant s'effectuer en moins de $\Theta(n\ln(n))$ en moyenne, alors que l'algorithme initial est linéaire). En revanche, une fois la table créée, la recherche est beaucoup plus rapide (logarithmique). Ainsi, la création de la table est intéressante si on a un grand nombre de recherches à effectuer dans un même texte. Cette idée est à la base des méthodes de recherche sur internet : les tables de suffixes des textes disponibles sur internet sont créées une fois pour toutes (indépendamment des demandes de recherche, et avec actualisations régulières), et prêtes à l'emploi au moment ou l'on souhaite effectuer une recherche. Ainsi, du point de vue de l'utilisateur, la recherche se fait en temps logarithmique.

Partie IV – Graphiques

La raison pour laquelle on utilise deux graphes séparés est que sinon, la première courbe écrase complètement la seconde. On passe en paramètre la valeur maximale de N, le pas et le nombre d'itération de l'expérience pour chaque valeur de N

On passe le nom de la fonction testée en paramètre, afin d'éviter d'avoir à écrire deux fois le même code :

```
def trace_graphe(max,pas,it,méthode,nom_méthode):
    X = []
    Y = []
    for N in range(pas,max + pas,pas):
        X.append(N)
        Y.append(méthode(N,5,it))
    plt.plot(X,Y)
    plt.savefig('pb_py003-fig3.eps')
    plt.title('Temps␣de␣réponse␣par␣la␣'+ nom_méthode)
    plt.show()
```

Évidemment, entre les deux méthodes, il faut changer le nom de sauvegarde du graphe, si on ne veut pas effacer le premier.

Vu les temps de calcul assez longs pour la première méthode, nous nous contentons d'une itération à chaque étape (ce qui est suffisant du fait du peu de variabilité du temps de complexité suivant les situations). Ainsi, l'instruction `trace_graphe(50000,200,1,teste_direct, 'méthode directe')` retourne le graphe de la figure 2. Ce graphe montre bien la linéarité de la méthode.

Pour la seconde méthode, la partie longue du traitement est la création de la table des suffixes. En revanche, les itérations à chaque étpa ne sont pas très longues. On effectue donc 100 itérations à chaque étape. L'instruction `trace_graphe(50000,200,100,teste_recherche_par_suffixes,'recherche avec suffixes')` retourne le graphe de la figure 3

On voit bien l'évolution logarithmique de la complexité, mais j'explique assez mal la grande variabilité des réponses à partir de N=12000.

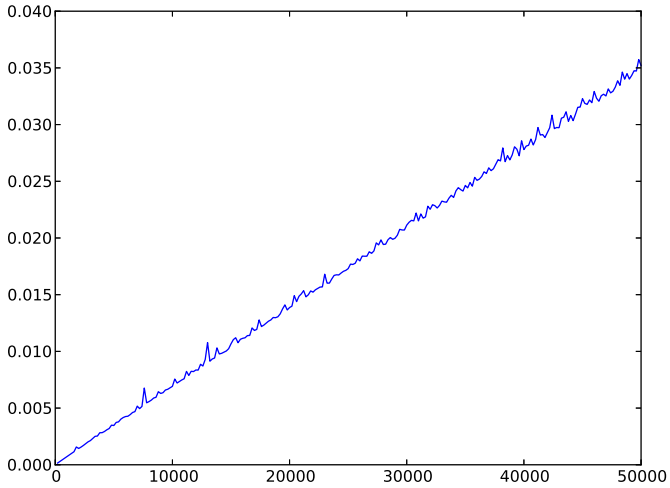

FIGURE 2 – Complexité de la méthode directe

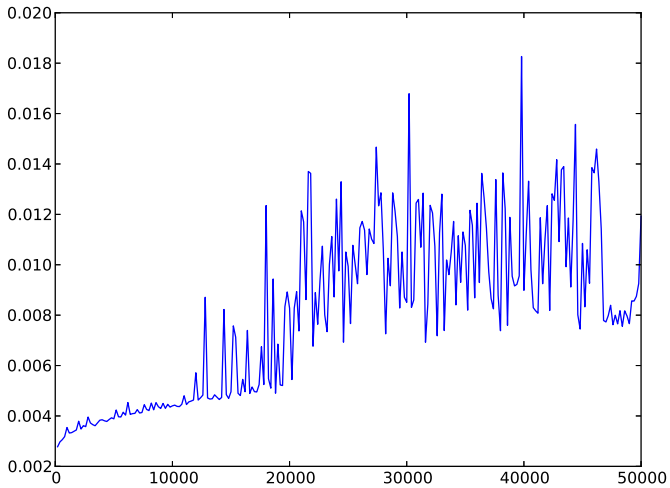

FIGURE 3 – Complexité de la méthode avec suffixes