Informatique PT. 2024 - 2025

Mathématiques & Informatique - F. Gaunard
http://frederic.gaunard.com
Lycée Voltaire, Paris 11e.

Devoir maison n°1

Solution

Préambule et objectifs

Le but de ce devoir est d'implémenter un algorithme efficace permettant de déterminer si un mot donné M apparaît dans un texte T et, le cas échéant, de préciser à quel endroit apparaît le mot.
Ce type d'algorithme possède de nombreuses applications (rechercher un mot dans une page web, rechercher une séquence codant une pathologie donnée dans un code génétique,...).

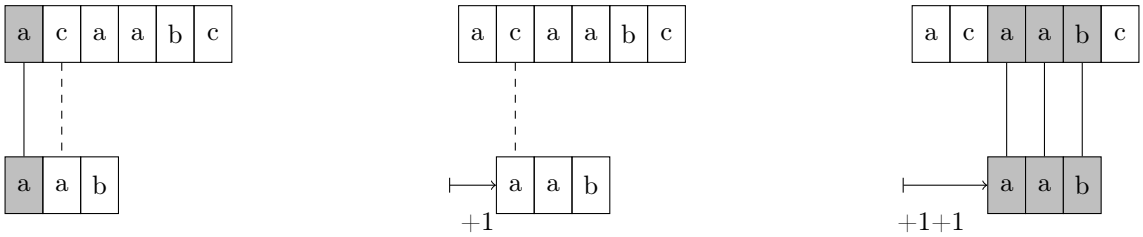
Le mot et le texte seront modélisés par des chaînes de caractères, que l'on notera M et T dans la suite: il s'agit ainsi de déterminer si la chaîne de caractères T possède une sous-chaîne égale à M. Par exemple `'jour'` est une sous-chaîne de `'bonjour'`.

On rappelle les commandes suivantes, qui s'appliquent à toute chaîne de caractères T:

- ✗ `len(T)` renvoie la longueur de T;
- ✗ si $i \leqslant n-1$ où n est la longueur de T, `T[i]` renvoie le i-ème caractère de la chaîne;
- ✗ la commande `+` permet de concaténer deux chaînes de caractères;
- ✗ si M est une seconde chaîne de caractères, alors `M==T` renvoie `True` si les deux chaînes sont égales, et `False` sinon.

1 Partie 1 : un algorithme naïf

Dans un premier temps, nous allons implémenter un algorithme naïf. Son principe est le suivant: on commence par comparer `M` aux `len(M)` premiers caractères de `T`, s'il y a égalité alors on peut renvoyer le résultat, et sinon on recommence en *décalant d'un cran* `M` pour le comparer à la sous-chaîne suivante de `T`.
Ce principe est expliqué sur la figure ci-dessous, où on a pris `T='acaabc'` et `M='aab'`.



1. Extraction d'une sous-chaîne: écrire une fonction `sous_chaine` qui prend en entrée une chaîne de caractères `T`, deux entiers $i \leqslant j \leqslant$ `len(T)`, puis renvoie la sous-chaîne de caractères de T allant du caractère d'indice i au caractère d'indice $j-1$. Par exemple, on doit avoir les résultats d'exécutions ci-dessous

```
>>>sous_chaine('bonjour',0,3)
'bon'
>>>sous_chaine('bonjour',0,1)
'b'
>>>sous_chaine('bonjour',0,0)
''
```

Solution. Les deux fonctions ci-dessous conviennent. Mieux vaut implémenter la seconde, si on connait la commande permettant d'extraire une sous-chaîne: si $i \leq j \leq n$ où n est la longueur de T, `T[i:j]` renvoie la sous-chaîne de T dont les indices vont de i à $j-1$.

```
def sous_chaine(T,i,j):
    souschaine=''
    for k in range(i,j):
        souschaine=souschaine+T[k]
    return souschaine

def sous_chaine(T,i,j):
    return T[i:j]
```

Dans la suite, on réutilisera la commande `T[i:j]` plutôt que `sous_chaine(T,i,j)`, afin de raccourcir les codes (cette commande doit de toute façon être connue). □

2. Écrire une fonction `sous_chaine_naif` qui prend en entrée deux chaînes de caractères `M` et `T`, puis renvoie `-1` si `M` n'est pas une sous-chaîne de `T`, et le plus petit entier i tel que `M` est la sous-chaîne de `T` commençant au terme d'indice `i` sinon. *On extraira une à une les sous-chaînes de T pour les comparer à M avec la commande* `==`.

Solution. Le code ci-dessous convient.

```
def sous_chaine_naif(M,T):
    m=len(M)
    n=len(T)
    for k in range(n-m+1):
        if M==T[k:k+m]:
            return k
    return -1
```

□

3. **Application**. Déduire de la question précédente une fonction `nb_occurrences` qui prend en entrée deux chaînes de caractères `M` et `T`, puis renvoie le nombre de fois où `M` apparaît dans `T`.

Solution. Voilà déjà un code réutilisant la fonction précédente.

```
def nb_occurrences(M,T):
    nb=0
    T2=T
    indice=sous_chaine_naif(M,T2)
    while indice!=-1:
        nb=nb+1
        T2=T2[indice+1:]
        indice=sous_chaine_naif(M,T2)
    return nb
```

Mais rien n'oblige à réutiliser `sous_chaine_naif`, et le code suivant est d'ailleurs plus simple.

```
def nb_occurrences2(M,T):
    m=len(M)
    n=len(T)
    i=0
    for k in range(n-m+1):
        if M==T[k:k+m]:
            i=i+1
    return i
```

□

4. **Test**. Télécharger le fichier DM1_PT2425.py et tester votre algorithme en précisant le nombre d'occurrences de votre groupe de colle dans le texte `T`.

5. Justifier que le nombre total de comparaisons entre caractères effectués lors de l'appel de la fonction `sous_chaine_naif` est inférieur à mn, où m, n sont les longueurs respectives de M et T.

Solution. Chacun des n termes de `T` est au plus comparé à chacun des m termes de `T`, il y a donc au plus mn comparaisons.
Mais ce nombre n'est qu'un majorant, car d'une part le premier terme de `T` n'est comparé qu'au premier terme de `M`, et d'autre part l'algorithme s'arrête dès que l'on a trouvé une sous-chaîne égale à `M` donc dans ce cas tous les termes de `T` ne sont pas utilisés. □

2 Partie 2 : préfixes, suffixes et motifs

Soit `M` une chaîne de caractères non vide. Un *préfixe* de `M` est une sous-chaîne de celle-ci, qui commence au terme d'indice 0. Par exemple, `'bon'` et `'bonj'` sont des préfixes de `'bonjour'`, mais `'bnj'` et `'onj'` n'en sont pas.

6. Quelle commande permet d'extraire de la chaîne de caractère `M` le préfixe de longueur i ?

Solution. La commande `M[0:i]` permet d'extraire le préfixe de longueur i de M. □

Un *suffixe* de `M` est une sous-chaîne de celle-ci, qui se termine au terme d'indice $n-1$, où n est la longueur de `M`. Par exemple, `'jour'` et `'our'` sont des suffixes de `'bonjour'`, mais `'jur'` et `'jou'` n'en sont pas.

Enfin, un *motif* de `M` est une sous-chaîne qui est à la fois un suffixe et un préfixe de `M`, mais qui est différente de `M`. Par exemple, `'a'` possède pour unique motif la chaîne vide `''`, et `'abcab'` possède deux motifs: `'ab'` et `''`.

7. Écrire une fonction `suffixe` qui prend en entrée deux chaînes de caractères `P` et `M`, puis renvoie `True` si `P` est un suffixe de `M`, et `False` sinon.

Solution. Il suffit de comparer M avec la sous-chaine de P de la bonne longueur.

```
def suffixe(P,M):
    return P==M[len(M)-len(P):len(M)]§
```

□

8. Écrire une fonction `motif_max` qui prend en entrée une chaîne de caractères `M`, puis renvoie le motif de `M` de longueur maximale.
On pourra extraire un à un les préfixes de `M` pour les étudier.

Solution. On initialise avec une chaine de caractères vide qu'on actualise si on rencontre un motif plus long, en utilisant la fonction suffixe précédente avec un préfixe extrait de M de plus en plus long.

```
def motif_max(M):
    m=len(M)
    motif_max=' '
    for k in range(m):
        if suffixe(M[0:k],M):
            motif_max=M[0:k]
    return motif_max
```

□

9. Écrire une fonction `motif_max_des_prefixes` qui prend en entrée une chaîne de caractères `M`, puis renvoie une liste `L` de même longueur que `M`, telle que `L[i]` est égal à la longueur du motif de longueur maximale du préfixe de longueur $i+1$ de M.
Par exemple, on doit avoir le résultat d'exécution ci-dessous

```
>>> motif_max_des_prefixes('abcab')
[0,0, 0, 1, 2]
```

Solution. La liste renvoyée contient les longueur des motifs de longueur maximale de chacuns des préfixes extraits de la chaine en argument.

```
def motif_max_des_prefixes(M):
    L=[ ]
    for i in range(len(M)):
        L.append(len(Motif_max(M[0:i+1])))
```

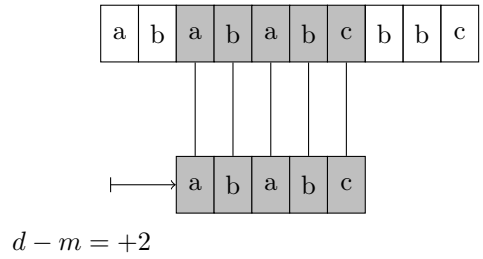
□

3 Partie 3 : l'algorithme KMP

Le principe de l'algorithme de Knuth-Morris-Pratt (KMP) consiste toujours à énumérer les sous-chaînes de `T`, mais on optimise cette fois le décalage à effectuer lorsque la sous-chaîne étudiée ne correspond pas à `M`: dans l'algorithme naïf on se décalait à chaque fois d'un cran, mais l'algorithme KMP propose d'effectuer un décalage potentiellement plus grand, en fonction de l'analyse des motifs des préfixes de `M`.

Sur le graphe ci-dessous, l'étude du cas `M='ababc'` et `T='abababcbbc'` montre qu'il n'est pas toujours possible après un échec de se décaler du nombre d de comparaisons effectuées avec succès à l'étape précédente, car on pourrait *sauter* une occurrence de `M` dans `T`:

Sur cet exemple, le problème résulte du fait que le préfixe `abab` de `M` possède un motif maximal de longueur $m = 2$ (en l'occurrence `ab`): la sous-chaîne qui échoue au test contient donc le début de la sous-chaîne qui doit réussir le test suivant! Ici, la seconde étape de l'algorithme KMP consiste à effectuer un décalage de $d - m$, ce qui permet de renvoyer le bon résultat:



Le principe de l'algorithme KMP est donc d'effectuer un décalage dépendant du plus long motif du préfixe de `M` que l'on vient de trouver dans la chaîne `T`, et pour cela on utilise la fonction `motif_max_des_prefixes`.

10. Écrire une fonction `Egalite` qui prend en entrée deux chaînes de caractères de même longueur `M1`, `M2`, puis renvoie -1 si celles-ci sont égales, et le plus petit entier k tel que `M1`$[k] \neq$ `M2`$[k]$ sinon.

Solution. On renvoie l'indice k dès que les termes sont différents. Sinon on renvoie -1.

```
def Egalite(M1,M2):
    n=len(M1)
    for k in range(n):
        if M1[k]!=M2[k]:
            return k
    return -1
```

□

11. Écrire une fonction `KMP` qui implémente l'algorithme de Knuth-Morris-Pratt: celle-ci prend en entrée deux chaînes de caractères `M` et `T`, puis renvoie `-1` si `M` n'est pas une sous-chaîne de `T`, et le plus petit entier i tel que `M` est la sous-chaîne de `T` commençant au terme d'indice `i` sinon.

Solution. Le code ci-dessous implémente la stratégie décrite dans l'énoncé. Attention toutefois il y a un cas particulier : lorsque les premiers termes comparés diffèrent, l'indice `k` doit être translaté d'une unité.

```
def KMP(M,T):
    m=len(M)
    n=len(T)
    k=0
    L=motif_max_des_prefixes(M)
    while k <= n-m:
```

```
        indice=Egalite(M,T[k:k+m])
        if indice==-1:
            return k
        elif indice==0:
            k=k+1
        else:
            k=k+indice-L[indice-1]
    return -1
```

□

12. Pouvez-vous modifier cet algorithme de sorte que chaque caractère de `T` soit lu au plus une seule fois? Que peut-on en déduire?

Solution. L'idée ici est de modifier la ligne `indice=Egalite(M,T[k:k+m])`, car à ce niveau on lit certains termes pour la seconde fois: ceux correspondant au motif maximal de la sous-chaîne trouvée à l'étape précédente dans T. Pour corriger cela on utilise une variable supplémentaire `dejavu` ayant pour valeur le nombre de termes déjà lus, que l'on peut donc "sauter".

```
def KMP2(M,T):
    m=len(M)
    n=len(T)
    k=0
    L=motif_max_des_prefixes(M)
    dejavu=0
    while k <= n-m:
        indice=Egalite(M[dejavu:],T[k+dejavu:k+m])
        if indice==-1:
            return k
        elif indice==0:
            k=k+1
            dejavu=0
        else:
            k=k+indice-L[indice-1]
            dejavu=L[indice-1]
    return -1
```

On peut en déduire que l'algorithme KMP est optimal du point de vue de la lecture des termes de T, puisqu'il est nécessaire de lire au moins une fois chaque terme pour déterminer si M est un sous-chaîne de T. Mais il faut noter qu'on a effectué un travail préliminaire sur M pour en déterminer les motifs, dans lequel les termes de M sont a priori lus plusieurs fois, ce qui ralentit tout de même l'exécution de l'algorithme. Mais en pratique la longueur de M est d'un ordre de grandeur inférieur à celle de T, donc le temps passé à effectuer ce travail peut être négligé.
On obtient donc un nombre de comparaisons de termes de l'ordre de n, alors que l'algorithme naïf présentait un nombre de comparaisons de l'ordre de mn (dans le pire des cas).

□

Bonus : mots conjugués

Cette dernière section, facultative, propose sans indication ni détail une recherche de réponse à un problème.

On dit que deux mots `M` et `M'` sont conjugués si `M=st` et `M'=ts` pour certains `s` et `t`.
Par exemple, `'lanver'` et `'verlan'` sont conjugués ou encore `'fou'` et `'ouf'`.

13. Écrire une fonction `test_mots_conjugués` qui prend en argument deux chaînes de caractères `M` et `M'` et renvoie `True` ou `False` selon que `M` et `M'` sont conjugués ou non.

Solution. La question n'est pas détaillée ce qui la rend un peu difficile. Après pas mal de réflexion, on se rend compte que

`M` et `M'` sont conjugués $\iff$ `len(M)=len(M')` et `M'` est une sous-chaîne de `MM`.

Par exemple on observe qu'on a bien `verlanverlan`. Il en découle le code suivant.

```
def test_mots_conjugues(M, M') :
    if len(M) != len(M') or KMP(M', M+M)==-1 :
        return False
    return True
```

□