DST Informatique pour tous
PC et MP
Durée : 2 heures

Samedi 21 décembre

Les trois parties peuvent êtres traitées indépendamment, la clarté de la présentation et la qualité de la rédaction font partie de l'évaluation. On veillera à utiliser des noms explicites pour les variables et les fonctions introduites. Le candidat pourra clarifier les programmes par l'ajout de commentaires judicieux si nécéssaire.

On se propose d'étudier trois algorithmes d'analyse et de recherche dans des chaines de caractère. Le premier consiste à mesurer la correspondance entre deux textes (aussi bien deux images). Les deux suivants sont des algorithmes de recherche de mots (de sous chaines) dans un texte donné.

1 Distance de Levenshtein

La distance de Levenshtein est une distance, au sens mathématique du terme, donnant une mesure de la différence entre deux chaînes de caractères. Elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne à l'autre. Elle a été proposée par Vladimir Levenshtein en 1965.

Cette distance est d'autant plus grande que le nombre de différences entre les deux chaînes est grand.

On appelle distance de Levenshtein entre deux chaînes `a` et `b` le coût minimal pour transformer `a` en `b` en effectuant les seules opérations élémentaires (au niveau d'un caractère) suivantes (on agit exclusivement sur M et successivement sur ses transformés) :

— substitution ;
— insertion (ou ajout)
— suppression (ou effacement).

L'algorithme est le suivant :

Soit a et b deux chaines de caractères de taille respective n et p, on note `lev(a,b)` la distance de Levenshtein entre deux chaines `a` et `b`.

Étape1 : Si une des chaines a et b est de longueur nulle on renvoie la longueur de la plus grandes des deux chaines.

Étape 2 : Si les premiers caractères de chacune des chaines a et b sont identique on recherche alors la distance de Levenshtein entre les deux sous chaines de a et b commençant au deuxième caractère.

Étape 3 : Sinon on renvoie `1+min(lev(a[1:],b),lev(a,b[1:]),lev(a[1:],b[1:]))`

On propose une programmation dynamique de la fonction `lev()`, qui consiste à compléter progressivement le tableau `D` de type `array` et de taille $(n+1)\times(p+1)$ (ou n et p sont les longueurs respectives des chaines a e b), afin que les $D[i,j]$ contiennent le cout, c-a-d la distance entre les sous chaines de taille respectives i et j, `a[n-i-1:]` et `b[p-j-1:]`.

1. Que vaut `lev('ab','ab')` puis `lev((abc','aac')` ?

2. Déterminer $D[0,j]$ et $D[i,0]$ pour $(i,j) \in [|0,n|] \times [l0,p|]$.
3. Programmer la fonction `Init_D(a,b)` prenant comme paramètres les chaines de caratères `a` et `b` et renvoyant la matrice de type `array`, initiale de D.
4. Compléter le programme suivant qui renvoie la valeur $lev(a,b)$:

```
import numpy as np
def lev(a,b):
    n=len(a)
    p=len(b)
    D=Init_D(a,b)
    cout=0
    for i in range(1,n+1):
        for j in range(1,p+1):
            if ..............:
                cout=0
            else:
                cout=.........
                ............................
    return D[n,p]
```

5. Donner un ordre de grandeur de la complexité de ce programme en fonction de n et p.

2 Algorithme de Rabin Karp

l'algorithme de Rabin-Karp ou algorithme de Karp-Rabin est un algorithme de recherche de sous-chaîne créé par Richard M. Karp et Michael O. Rabin (1987). Cette méthode recherche la présence d'un motif dans un texte, ou plus généralement la présence d'un motif parmi un ensemble de motifs donnés (c'est-à-dire des sous-chaînes) dans un texte.

1. Compléter la fonction de recherche naive, `naif(c,cle)`, prenant comme paramètres la chaine de caractère du texte `c` et le motif `cle`, paracourant tout le texte `c` et renvoyant l'entier i correspondant à la position dans `c` de la première apparition de la clé `cle` dans la chaine `c` et le booléen `False` si elle n'est pas dans `c` :

```
def naif(t,m):
    n=len(t)
    p=len(m)
    assert n>=p
    for i in range(n-p):
        ..............
        ..............
    ..................;
```

   Donner un ordre de grandeur de ce programme en fonction de n et p.

2. L'algorithme de Rabin-Karp se propose d'améliorer l'algorithme naïf en utilisant une fonction de hachage.

   Par exemple pour le texte `' abracadabra '` et la recherche d'un motif de 3 caractères `'cad'` dans le texte, on compare non plus les sous chaines `'abr'`, `'bra'` et ainsi de suite au motif `'cad'`, mais leurs valeurs haschées. On effectue ainsi deux comparaisons numériques au lieu de six comparaisons de caractères.

   Pour se faire, on se propose d'utiliser la commande `hash()` qui renvoie une valeur entière pour une chaine de caractère donnée.

Dans l'exemple précédant on obtient :

```
hash('abr')
Out[1]: 3245953727622197344

hash('bra')
Out[2]: 4184199061459475318

hash('cad')
Out[3]: -1646046172163184148
```

On constate alors que les valeurs numériques sont toutes différentes pour les deux premières étapes ; il n'ya donc pas de correspondance.

On se propose alors de modifier le programme `naif()` précédant, en se contentant non plus de comparer les caractères de chaque chaines mais en comparant une forme "hasher", donc numérique des sous chaines. On rappelle cependant que deux chaines numérique peuvent renvoyer une même valeur de haschage sans pourtant être identiques.

Adapter la fonction `naive()` précédente, en utilisant la fonction de hashage et compléter le script suivant :

```
def RK(t,m):
    n=len(t)
    p=len(m)
    assert n>=p

    mh=hash(m)

    for i in range(n-p):

        if ......................:
            if ..................:
                return .........
    return False
```

Evaluer la complexité de ce programme en fonction de n et $H(p)$ la complexité de la fonction de hashage, les longueurs respectives des chaines `cle` et `c`.

3. On souhaite maintenant programmer différentes fonctions de haschages adaptées à nos besoins. Pour se faire, on utilisera la commande `ord()`, qui renvoie le codage numerique ASCII d'un caractère. C'est une fonction qui associe à un caractère une valeur numérique, comme par exemple :

```
ord('a')
Out[1]: 97
```

   (a) La première fonction de hachage, que nous allons programmer, consiste à associer à une chaine de caractère la concaténation des valeurs numériques du codage ASCII des caractères composants la chaine. Par exemple pour `'ha'`, comme `ord('h')` est 104 et `ord('a')` est 97 on souhaite renvoyer 10497. Programmer cette fonction `num(c)`, qui renvoie l'entier correspondant à la chaine `c`.

   (b) Le codage numérique de la chaine, proposé par `num()`, n'est pas vraiment une fonction de hachage, pourquoi ? On souhaite modifier la fonction `num()` pour renvoyer une valeur congrue à l'entier q en programmant la fonction `hash1(c,q)`. Programmer une fonction Python `hash1(c,q)` correspondant.

   (c) Programmer la fonction `hash2(c)` renvoyant la somme des valeurs du codage ASCII des éléments de la chaine `c` modulo 10^{10}.

(d) Programmer la fonction `hash3(c,q)` renvoyant le codage numérique en base q sous la forme :

$$\texttt{hash3}(c,q) = \sum_{i=0}^{\texttt{len}(c)-1} c[len(c)-i] * q^i$$

Par exemple, l'empreinte de la première sous-chaîne, `'abr'`, en utilisant la base 101, est :

$$a = 97, b = 98, r = 114.$$
$$\texttt{hach3(' abr ')} = (97 \times 101^2) + (98 \times 101^1) + (114 \times 101^0) = 999509$$

4. Si le hashage de la clé s'effectue bien avec un unique traitement (on parle alors de prétraitement), il faudra le répéter pour chaque sous chaine de la chaine texte `c` que nous voulons comparer et donc appliquer successivement la fonction de hachage aux différentes sous chaines ; ce qui augmente sensiblement la complexité.

   Par exemple pour avec le texte `'abracadabra'` et la recherche d'un motif de 3 caractères, l'empreinte de la première sous-chaîne, `'abr'`,est 999509 mais il faut aussi déterminer celle de `'bra'` et ainsi de suite.

   L'interêt des fonctions de haschages proposées précédemment est qu'on peut facilement passer du haschage de la sous chaine `c[i:i+n]` à celui de `c[i+1:i+1+n]` en remarquant que :

$$\texttt{hash3}(t[i+1:n+i+1]) = (\texttt{hash3}(t[i:i+n]) - \texttt{ord}(t[i]))/q + \texttt{ord}(t[i+n]) * q^{n-1}$$

   Adapter la fonction `RK(c,cle)` à cette méthode, préciser la complexité dans le pire des cas.

3 Boyer Moore

L'algorithme de Boyer-Moore consiste aussi à recherche un mot, c-a-d une chaine de caractère `cle`, dans un texte `c`. Mais Il effectue la vérification à l'envers, en commençant par comparer la dernière lettre de la clé `cle` avec celle du texte `c`. Par exemple, s'il commence la recherche de la clé NOEL au début d'un texte, il vérifie d'abord la quatrième position dans le texte `c` en regardant si elle contient un L. Ensuite, s'il a trouvé un L, il vérifie la troisième position pour regarder si elle contient le dernier E de la sous-chaîne, et ainsi de suite jusqu'à ce qu'il ait vérifié la première position du texte pour y trouver un N.

1. Programmer la fonction `saut(cle,c,i,n)` prenant comme paramètres deux chaines de caractères, la clé `cle` et le texte `c`, deux entiers type `integer` i et n renvoyant le booléen `True` si le nième caractère de la clé est identique au $i+n$ ième du texte `c` et `False` sinon. Par exemple :

```
saut('abcabc','barbapapa',2,3) # compare la troisieme lettre de cle
                                                    #avec la cinquieme de c
Out[1]: False

saut('abcabc','barbapapa',2,4) # compare la quatrieme lettre de cle
                                                    #avec la sixieme de c
Out[2]: False

saut('abcabc','barbapapa',2,2) # compare la deuxieme lettre de cle
                                                     #avec la quatrieme de c
Out[3]: True
```

2. Programmer la fonction `Position_occurence(cle)` prenant comme paramètre la chaine de caractère `cle` et renvoyant le dictionnaire `S`, de clés les caractères distincts de la chaine `cle` auxquels on associe comme valeur, la plus grande position (en partant du début à gauche) de cette lettre dans la clé `cle`. Par exemple :

```
Position_occurence('babar')
Out[78]: {'b': 3, 'a': 4, 'r': 5}
```

Car `'r'` est en cinquième position alors que parmi les deux `'a'` celui en quatrième position est bien celui de plus grande position et le `'b'` en troisième positions est bien celui de plus grand indice .

3. Si on se donne une chaine de caractère `c` et une clé `cle` de taille n. L'algorithme compare à chaque étape, le terme $i+n$ de la chaine c avec le dernier terme de la cle, où initialment $i=1$ (car on commence par la première lettre de `c`).
   — Si ce terme n'est pas une lettre de la clé on peut décaler la recherche automatiquement de la longueur de la clé et se positionner en $i+n+1$ ième position car on est assuré que pour tout $j \leq n$ les sous chaines `c[i+j-1:i+n+j-1]` ne peuvent pas correspondre à la clé (elle contienne une lettre qui n'est pas dans la clé).
   — Par contre si le terme `c[i+n-2]` ne correspond pas à `cle[n-1]` mais qu'il est dans la clé, on ne peut pas se décaler de n mais de la distance qui sépare, dans la clé, la lettre `c[i+n-2]` de la dernière lettre.

   Prenons un exemple :

   Dans le texte `c='pasperdu'` avec la `cle='per'`.
   — La cle est de longueur $n=3$ et initialement $i=1$.
   — On regarde directement la troisième lettre de `c`, c-a-d `c[2]` donc ici `'s'`.
   — On la compare à la dernière lettre de la clé soit `'r'`. Elles ne correspondent pas, et `'s'` n'est pas dans la clé.
   — On décale la recherche de 3.
   — On se place alors en quatrième position dans le texte `c`, pour finalement continuer la recherche sur `'perdu'`, en comparant le dernier terme `'r'` de la cle avec le troisième terme de `'perdu'` (en fait le $3+3$ éme de `c`), qui ici est `'r` et qui correspond.

   Prenons maintenant un autre exemple :

   `c='tperduouquoi'` avec la `cle='per'`.
   — On compare donc la troisième lettre de `c`, donc ici `'e'` avec le `'r'` de la clé `'per'` qui ne correspondent pas.
   — Comme `'e'` est une lettre de la clé (c'est même la deuxième lettre de clé)et qu'elle est positionnée dans `cle` à une distance $3-2=1$ de la dernière lettre `'r'` : On se décalera alors de 1.
   — En se plaçant maintenant en $i=1+1$. Pour continuer la recherche dans `perduouquoi'`. On compare alors le dernier terme `'r'` de la clé avec le troisième terme de `perduouquoi'` (en fait le quatrième de `c`), qui ici est `'r` et qui correspond.

   On souhaite, dans cette question, programmer la fonction `Tableau_saut(cle)` prenant comme paramètre la clé, `cle`, et renvoyant un dictionnaire de saut `S` qui à chaque caractère de `cle` associe la distance de saut, c-a-d la distance entre la position de la lettre dans la clé et la dernière lettre de cette même clé.

   Les clés du dictionnaires `S` seront donc les lettres de la clé `cle` et leur valeur : la distance de saut. Comme plusieurs mêmes lettres peuvent être présentes dans la clé `cle`, on ne conservera alors que la plus petite distance de saut comme valeur.

   Ainsi pour la clé `'noel'` on obtiendra :

```
Tableau_saut('noel')
Out[1]: {'n': 3, 'o': 2, 'e': 1, 'l': 0}

Tableau_saut('babar')
Out[2]: {'b': 2, 'a': 1, 'r': 0}
```

   Compléter le programme suivant :

```
def Tableau_saut(cle):
    S=Position_occurence(cle)
    ................
    ............
    ............
    return S
```

4. On souhaite maintenant comparer les lettres de la sous chaine `c[i-1:i-1+n]` de `c` avec celle de la clé `cle`. On pose n la longueur de la clé.
   On cherche le plus petit indice j, tel que `c[i+j-1:i+n-1]` corresponde à `cle[j-1:n]`, c-a-d le plus grand indice sans correspondance avec la clé à partir de la ième position.
   Par exemple, pour `c='commetoujours'`, `cle='bonjour'`, et $i = 6$, on compare `'toujour'` et `'bonjour'`. Le plus petit indice j rechercher est donc 3.
   Ou encore :

```
corespondance('per','tperdu',1)
Out[1]: 3

corespondance('per','tperdu',2)
Out[2]: 0
```

   Programmer la fonction `corespondance(cle,c,i)` renvoyant le plus petit indice j, tel que `c[i+j-1:i+n-1]` corresponde à `cle[j-1:n]`.

5. On supposera dans cette question que la clé est constituée de lettres toutes distinctes, comme par exemple `cle='perdu'`.
   (a) L'algorithme consiste à effectuer une recherche successive en effectuant les sauts, les plus grands possible (pour avancer au plus vite).
   Le choix du saut de décalage s'obtient ici, en s'attachant à analyser la lettre `c[j+i-1]`, où j le plus petit indice tel que `c[i+j-1:i+n-1]` correspondent à `cle[j-1:n]`, c-a-d la première lettre en partant de la droite, sans correspondance avec celle de la clé `cle`. Dans le cas où les lettres de la clé sont distinctes, la fonction `Tableau_saut(cle)` renvoie le dictionnaire de saut `S`.
   On souhaite programmer la fonction `Test(cle,c,S,i)` renvoyant le couple `(T,k)` où `T` est le booléen `True` si il ya correspondance totale entre la chaine `c[i-1,i+n-1]` et la clé `cle`, ainsi que l'indice de position `k`, et renvoie `False` sinon, ainsi que la valeur du décalage maximal, soit la longueur n de la clé.
   Par exemple

```
Test('per','tperdu',S,1)
Out[1]: (False, 1)

Test('per','tperdu',S,2)
Out[2]: (True, 2)
```

   Compléter le programme de la fonction `Test(cle,c,S,i)` suivant :

```
def Test(cle,c,S,i):
    n=len(cle)
    m=corespondance(cle,c,i)
    if m==0:
        return ..........
    u=c[..........]
```

```
        if u in S:
              return ............

        return .........
```

(b) Programmer la fonction `RBM1(cle,c)` prenant comme paramètre la clé `cle` et la chaine de caractère `c` et renvoyant :
- Le uplet `(True)` si la clé est dans le texte `c`, avec i est la position dans la chaine `c` de la première lettre de la clé dans le texte `c`
- `False` si la clé n'est pas dans le texte `c`.

6. On suppose maintenant que le mot clé peut contenir plussieurs fois la même lettre. Par exemple `cle='toujours'`. Si au cours de notre test de comparaison on rencontre un `'o'` mal placé; de combien puis je me décaler. Par exemple avec `c='ctojours'`. Il y a cinq correspondance jusqu'à obtenir le caractère `'o'` qui est aussi dans la clé. Doit on se décaler de trois comme prévu initialement ? Ici on voit bien qu'il n'y a pas de sous chaine de taille six `'ojours'` dans `'toujours'`. On peut se décaler du maximum possible soit de 7 (la taille de la clé). Par contre si la clé est `'atoutout'` et le texte `c='babatouttout'`, la comparaison avec `'babatout'` donne une première difference avec le `'a'` qui est dans la clé et en cinquième position en partant de la droite. Or il existe bien une sous chaine `'atout'` de la clé qui corresponde et soit de la bonne taille : elle commence avec la première lettre de la clé et se termine en cinquième position (évidement) on doit donc se décaler de trois, la longueur de la clé moins cinq.

   On propose la fonction suivante :

```
def Prepa(c,cle):
      if len(cle)==0:
         return [Tableau_saut(c)]
      n=len(cle)
      m=len(c)
      S2={}
      for i in range(len(c)-n):
           if c[i+1:i+1+n]==cle:
              S2[c[i]]=m-n-i-1
      cle=cle[1:]
      L=Prepa(c,cle)
      if len(S2)!=0:
          L.append(S2)
      return L
```

(a) Que renvoie `Prepa('per','per')`

(b) Si on execute la commande suivante :

```
L=Prepa('bbaba','bbaba')

L[2]['b']
Out[1]: 2
```

A quoi correspond le nombre 2 ? Que renvoie `L[0]`

(c)

7. Adapter le programme `Test()` qu'on notera `Test2(cle,c,L,i)`, ou L est la liste des sauts obtenue en appliquant la fonction `Prepa()`, vérifiant :

```
Test2('bbaba','barbabbababababar',L,1)
Out[1]: (False, 5)

Test2('bbaba','barbabbababababar',L,2)
Out[2]: (False, 1)

Test2('bbaba','barbabbababababar',L,6)
Out[3]: (True, 6)
```

8. Programmer une fonction récurssive `Recherche(cle,c,L,i)` renvoyant `(False,np.inf)` si la clé n'est pas dans la chaine et `(True,j)` sinon, où j est la position du mot clé dans la chaine `c`. Ainsi les commandes suivantes renvoient :

```
L=Prepa('bbaba','bbaba')

Recherche('bbaba','barbabbababababar',S,1)
Out[132]: (True, 6)
```

```
L=Prepa(cle,cle)
def Recherche(cle,c,L,i):
    m=len(c)
    if i>=m:
        return (False,np.inf)
    u=Test2(cle,c,S,i)

    ............
    ............
    ..............
```