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.

1. `lev('ab','ab')`$= 0$ puis `lev((abc','aac')`$= 1$.
2. $D[0,j] = j$ et $D[i,0] = i$ 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.

```
def init_D(a,b):
    n=len(a)
    p=len(b)
    D=np.zeros((n+1,p+1))
    for i in range(n+1):
        D[i,0]=i
    for j in range(p+1):
        D[0,j]=j
    return 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 a[n-i-1]==b[p-j-1]:
                cout=0
            else:
                cout=1
            D[i,j]=min(D[i-1,j]+1,D[i,j-1]+1,D[i-1,j-1]+cout)
    return D[n,p]
```

5. Un ordre de grandeur de la complexité de ce programme est $O(np)$.

2 Algorithme de Rabin Karp

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+1):
        if t[i:p+i]==m:
            return i+1
    return False
```

2. La complexité de ce programme est d'ordre $O(nH(p))$ avec $H(p)$ la complexité du haschage de p caractères.

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

    mh=hash(m)

    for i in range(n-p):

        if mh==hash(t[i:i+p]):
            if t[i:i+p]==m:
                return i+1
    return False
```

3.

```
def num(c):
    n=len(c)
    nb=0
    for i in range(1,n+1):
```

```
            nb+=ord(c[n-i])*10**(len(str(nb)))
        return nb//10
    def hash1(t,q):
        return num(t)%q

def hash2(t):
    h=0
    for i in t:
        h+=ord(i)
    return h%10**10

def hash3(t,q):
    n=len(t)
    nb=ord(t[-1])
    for i in range(1,n-1):

        nb+=ord(t[n-i])*q**(i)
    return nb
    def RKbis(t,m):
        n=len(t)
        p=len(m)
        assert n>=p

        mh=hash3(m)
        h=(hash3(t[i:i+n])
        for i in range(n-p):

            if mh==h:
                if t[i:i+p]==m:
                    return i+1
            h=h-ord(t[i])*q*n)*q+ord(t[i+n])
        return False

# h=(hash3(t[i:i+n])-ord(t[i])*q*n)*q+ord(t[i+n])
```

3 Boyer Moore

1.
```
def saut(cle,c,i,n):
    return c[n-2+i]==cle[n-1]
```

2.
```
def Position_occurence(cle):
    S={}
    n=len(cle)
    for i in range(n):
        S[cle[i]]=i+1
    return S
```

```
def Tableau_saut(cle):
```

```
    S=Position_occurence(cle)
    n=len(cle)
    for u in S:
        S[u]=n-S[u]
    return S
```

```
def corespondance(cle,c,i):
    n=len(cle)
.
    while saut(cle,c,j,n) and n>0:
        n=n-1

    return n
def Test(cle,c,S,i):
.   n=len(cle)
    m=corespondance(cle,c,i)
    if m==0:
        return (True,i)
    u=c[i+m-2]

    if u in S:
        return (False,S[u])

    return (False,n)
```

```
def RBM(cle,c):
    S=Tableau_saut(cle)
    i=1
    m=len(c)-len(c)
    while Test(cle,c,S,i)[0]==False and i<m:
        i+=Test(cle,c,S,i)[1]
    if i>=m:
        return False
    return (True,i)
```

```
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
```

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

```
[{'p': 2, 'e': 1, 'r': 0}, {'e': 0}, {'p': 0}]
```

(b) Si on execute la commande suivante :

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

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

A quoi correspond le nombre 2 ?

signifie que 'bba' estr décaler de deux par rapport à 'aba'.

Que renvoie `L[0]`

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

```
L
Out[4]: [{'b': 1, 'a': 0}, {'b': 0}, {'b': 2, 'a': 0}, {'b': 0}, {'b': 0}]
L[0]
Out[5]: {'b': 1, 'a': 0}
```

```
def Test2(cle,c,S,i):
    n=len(cle)
    m=corespondance(cle,c,i)

    if m==0:
        return (True,i)
    u=c[i+m-2]

    if u in S[n-m]:
        return (False,S[n-m][u])

    return (False,n)
```

```
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)

    if u[0]:
        return (True,i)
    return Recherche(cle,c,S,i+u[1])
```