ISM MP, Informatique
Année 2025/2026 01/12/2025

Corrigé du devoir surveillé n°1

I Préliminaires

1 . On a

`tab[0]=[0, 1,..., 11, 13]`, `tab[1]=[0, 4,..., 6, 1]`, `tab[0][0]=0`, `tab[1][-1]=1`

2 . On propose :

```
def ordMin(tab,n):
    ymin=tab[1][0]
    for y in tab[1]:
        if y<ymin:
            ymin=y
    return ymin
```

3 . On propose :

```
def occOrd(tab,n,y):
    res=[]
    for k in range(n):
        yk=tab[1][k]
        if yk==y:
            res.append(k)
    return res
```

4 . On propose :

```
def plusBas(tab,n):
    ymin=ordMin(tab,n)
    tab_occ=occOrd(tab,n,ymin)
    ind=tab_occ[0]
    xmin=tab[0][ind]
    for i in tab_occ:
        x=tab[0][i]
        if x<xmin:
            xmin,ind=x,i
    return ind
```

5 . On propose :

```
def plusBas2(tab,n):
    ymin,ind=tab[1][0],0
    for i in range(n):
        x,y=tab[0][i],tab[1][i]
        if y<ymin:
            ymin,ind=y,i
        elif y==ymin and x<tab[0][ind]:
            ind=j
    return ind
```

Cette deuxième version est préférable. Il y a le coût temporel et spatial de la construction des indices d'occurrences de `ymin` pour la première version. Pour la deuxième version, le coût spatial est en O(1). On réalise éventuellement plus d'affectations `ind=j` que d'occurences de `ymin` et donc de `res.append(k)` puisque la détermination de `ymin` est en cours de réalisation mais c'est une opération à coût constant.

6 . On propose :

```
def orient(tab,i,j,k): # O(1)
    xi,yi=tab[0][i],tab[1][i]
    xj,yj=tab[0][j],tab[1][j]
    xk,yk=tab[0][k],tab[1][k]
    d=(xj-xi)*(yk-yi)-(yj-yi)*(xk-xi)
    if d==0:
        return 0
    return d//abs(d)
```

II Algorithme du paquet cadeau

7 . On propose :

```
def prochainPoint(tab,n,i):
    if i!=0:
        ind=0
    else:
        ind=1
    for k in range(n):
        if k!=i:
            if orient(tab,i,ind,k)<=0:
                ind=k
    return ind
```

8 . On propose :

```
def convJarvis(tab,n):
    res=[]
    deb=plusBas(tab,n)
    fini=False
    pt=deb
    while not fini:
        res.append(pt)
        pt=prochainPoint(tab,n,pt)
        fini=pt==deb
    return res
```

9 . On propose :

```
def conv_rec(tab,n,res):
    nxt=prochainPoint(tab,n,res[-1])
    if nxt!=res[0]:
        res.append(nxt)
        conv_rec(tab,n,res)
```

10 . On propose :

```
def convJarvisRec(tab,n):
    deb=plusBas(tab,n)
    res=[deb]
    conv_rec(tab,n,res)
    return res
```

III Algorithme de tri

11 . Le tri rapide et tri fusion s'appuient sur le paradigme « diviser pour régner ». On a le tableau des complexités :

	Complexité temporelle
tri par insertion	$\mathrm{O}(n) - \mathrm{O}(n^2)$
tri rapide	$\mathrm{O}(n\log(n)) - \mathrm{O}(n^2)$
tri fusion	$\mathrm{O}(n\log(n))$

12 . On propose le tri rapide pas en place :

```
def tri(L):
    if L==[]:
        return []
    else:
        pivot=L[0]
        L1,L2=[],[]
        for x in L[1:]:
            if x<pivot:
                L1.append(x)
            else:
                L2.append(x)
        return tri(L1)+[pivot]+tri(L2)
```

13 . On propose :

```
def tri_ind(L,I):
    if L==[]:
        return [],[]
    else:
        n=len(I)
        pivot,ind_pivot=L[0],I[0]
        L1,I1=[],[]
        L2,I2=[],[]
        for k in range(1,n):
            x,ind_x=L[k],I[k]
            if x<pivot:
                L1.append(x)
                I1.append(ind_x)
            else:
                L2.append(x)
                I2.append(ind_x)
        L1,I1=tri_ind(L1,I1)
        L2,I2=tri_ind(L2,I2)
        return L1+[pivot]+L2,I1+[ind_pivot]+I2
```

14 . On propose :

```
def tri_tab(tab):
    return tri_ind(tab[0],tab[1])
```

IV Modèle de piles

15 . La programmation récursive utilise la structure de piles pour gérer les appels récursifs.

16 . On propose :

```
def Pile():
    return []

def estVide(P):
    return P==[]

def empiler(P,x):
    P.append(x)

def sommet(P):
    return P[-1]

def depiler(P):
    return P.pop()
```

V Algorithme de balayage

17 . On propose :

```
def majES(tab,es,i): # taille de es<=i
    j=depiler(es)
    # pour la condition d'arrêt :
    # on tire profit de l'évaluation paresseuse
    while not estVide(es) and orient(tab,i,j,sommet(es))<0:
        j=depiler(es)
    empiler(es,j)
    empiler(es,i)
```

18 . On propose :

```
def majEI(tab,ei,i):
    j=depiler(ei)
    while not estVide(ei) and orient(tab,i,j,sommet(ei))>0:
        j=depiler(ei)
    empiler(ei,j)
    empiler(ei,i)
```

19 . On propose :

```
def convGraham(tab,n):
    es=Pile()
    ei=Pile()
    empiler(es,0)
    empiler(ei,0)
    for i in range(1,n):
        majES(tab,es,i)
        majEI(tab,ei,i)
    # on élimine les doublons
    # et on rassemble enveloppe supérieure et inférieure
    depiler(es)
    while not estVide(es):
        empiler(ei,depiler(es))
    depiler(ei)
    return ei
```

20 . Chaque point entre une seule fois dans chaque pile et en sort zéro ou une fois avec des opérations sur les piles et des calculs d'orientation en $O(1)$ d'où un coût en $O(n)$. Si les abscisses de `tab` ne sont pas préalablement triées, on commence par faire ce tri :

```
def convGraham(tab,n):
    tab=tri_tab(tab)
    ...
```

Le coût du tri est en $O(n\log(n))$ d'où une complexité temporelle en $O(n) + O(n\log(n)) = O(n\log(n))$.