NOM :..... Prénom :.....

CPGE PSI* 2025/2026 **Informatique**
Lycée La Fayette **Nathalie Planche**

Devoir surveillé n°3 - Document Réponse

Exercice 1

1. Suite d'instructions pour construire les listes `LX` et `LY`.

```
f=open('integrale.csv','r')
total_lignes=f.readlines()
f.close()
LX=[]
LY=[]
for ligne in total_lignes:
    coord=ligne.split(';')
    LX.append(float(coord[0]))
    LY.append(float(coord[1]))
```

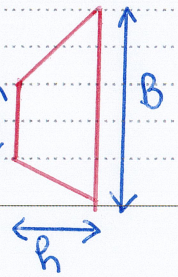
2. Suite d'instructions pour représenter graphiquement les points

```
import matplotlib.pyplot as plt
plt.plot(LX,LY)
```

3. Fonction **trapeze**

```
def trapeze(liste_abs, liste_ord):
    n=len(liste_abs)
    aire=0
    for i in range(n+1):
        aire+=(liste_abs[i+1]-liste_abs[i])
              *(liste_ord[i+1]+liste_ord[i])/2
    return aire
```

Rappel aire d'un trapèze



$$Aire = \frac{(B+b)*h}{2}$$

4. Instruction pour calculer I

```
trapeze(LX,LY)
```

Exercice 2

Q1.

$$M = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Mode de construction de la matrice d'adjacence

Le coefficient à la $i^{ème}$ ligne et $j^{ème}$ colonne vaut :
- 1 s'il existe une arête entre les sommets i et j
- 0 sinon

Si n est le nombre de sommets du graphe, cette matrice a donc n lignes et n colonnes

Q2. Liste d'adjacence du graphe G_{ex}

```
LA = [[1,3,4,6,7], [0,2,3], [1,3], [0,1,2,4], [0,3,5,6,7],
      [4,6,7], [0,4,5,7], [0,4,5,6]]
```

Q3. Un avantage et un inconvénient d'une matrice d'adjacence, d'une liste d'adjacence

Matrice :
- avantage : accès direct à l'existence d'une arête entre i et j
- inconvénient : beaucoup de place en mémoire, "inutilement" si le graphe est peu dense (beaucoup de 0)

Liste :
- avantage : moins de place en mémoire si le graphe est peu dense
- inconvénient : complexité linéaire pour la recherche d'une arête entre 2 sommets

Q4.

Sommet	0	1	2	3	4	5	6	7
Degré	5	3	2	4	5	3	4	4

Q5. Fonction `voisins`

```
def voisins(i, j, LA):
    return j in LA[i]
```

Q6. Fonction `coloration_valide`

```
def coloration_valide(LA,C):
    n=len(C)  #le nombre de sommets
    for i in range(n):
        for j in LA[i]:  #on parcourt les voisins de i
            if C[j]==C[i]:
                return False
    return True
```

Q7. Complexité temporelle dans le pire des cas de `coloration_valide`

On a deux boucles imbriquées, la boucle extérieure tourne n fois, et dans le pire des cas la boucle intérieure tourne (n-1) fois (nombre de voisins maximum de i). L'instruction "if ..." est de complexité constante.
Il s'agit donc d'une complexité quadratique, en $O(n^2)$

Q8. Liste composée de `n` éléments valant `-1`

```
C=[-1]*n
```

Q9. Compléter la fonction selon les instructions de l'énoncé

```
def colore_sommet(C,s,LA) :
    # on détermine la liste des couleurs des voisins de s déjà colorés
    coul_vois = []
    for j in LA[s]: #les voisins de s
        if C[j]!=-1 : #voisin déjà coloré
            coul_vois.append(C[j])

    # coul_vois est maintenant déterminée et on recherche la
    # plus petite couleur, notée num_coul, absente de coul_vois:
    num_couleur = 0
    while num_couleur in coul_vois:
        num_couleur += 1
    #la boucle s'arrêtera car au pire on utilisera
    #n couleurs

    # la valeur num_coul trouvée devient la couleur du sommet s:
    C[s]=num_couleur
```

Q10. Fonction `colorer1`

```
def colorer1(LA)
    n=len(LA)
    C=[-1]*n
    for s in range(n):
        colore_sommet(C,s,LA)
    return C
```

Q11. Fonction `colorer2`

```
def colorer2(ordre, LA)
    n=len(LA)
    C=[-1]*n
    for s in ordre:
        colore_sommet(C,s,LA)
    return C
```

Q12. Liste des couleurs renvoyée par `colorer2` pour G_{ex}. Nombre de couleurs utilisées.

sommet 7 → couleur 0
" 6 " 1
" 5 " 2
" 4 " 3
" 3 " 0
" 2 " 1
" 1 " 2
" 0 " 4

On a utilisé 5 couleurs
La liste C renvoyée est
[4,2,1,0,3,2,1,0]

Q13. Fonction `degre`

```
def degre(LA):
    liste_deg=[]
    n=len(LA)
    for s in range(n):
        liste_deg.append(len(LA[s]))
    return liste_deg
```

Q14. Fonction `init`

```
def init(n):
    return [[]]*n
```

Q15. Fonction `ranger`

```
def ranger(LA):
    n=len(LA)
    liste=init(n)
    liste_deg=degre(LA)
    for i in range(len(liste_deg)):
        liste[liste_deg[i]].append(i)
    return liste
```

Q16. Fonction `renverse`

```
def renverse(L):
    n=len(L)
    Lnew=[]
    for i in range(n)
        Lnew.append(L[n-i-1])
    return Lnew
```

Q17. Fonction `trier_sommets`

```
def trier_sommets(LA):
    R=ranger(LA)
    liste=[]
    for x in R: #x est la liste des sommets avec un certain degré
        liste=liste+x    #concaténation des listes
    return renverse(liste)
```

Q18. Complexité temporelle dans le pire des cas de la fonction `trier_sommets`

fonction init : $O(n)$

fonction degre : $O(n)$

fonction ranger : $O(n)+O(n)+O(n)=O(n)$

fonction renverse : $O(n)$

fonction trier_sommets : $O(n)+O(n^2)+O(n)=O(n^2)$ (boucle)

complexité linéaire

Q19. Fonction `colorer3`

```
def colorer3(LA):
    ordre = trier_sommets(LA)
    return colorer2(ordre, LA)
```

Complexité de la fonction `colorer3`

$O(n) + O(n \times n) = O(n^2)$

trier_sommets ; boucle dans colorer2 ; colore_sommet ; complexité de colorer2

Complexité quadratique

Q20. Liste `C` des couleurs renvoyée par `colorer3` pour le graphe G_{ex}

l'ordre renvoyé par trier_sommets est [0,4,3,6,7,1,5,2]

C=[0,1,0,2,1,0,2,3] On a utilisé 4 couleurs.