Proposition de corrigé du DS1

Exercice 1 - Graphe océan

1. On a :

```
M=array([[0, 0, 0, 1],
         [1, 0, 1, 1],
         [0, 0, 0, 1],
         [0, 0, 0, 0]])
```

et `L=[[3], [0, 2, 3], [3], []]`.

Remarques : les valeurs sur la diagonale de `M` importent peu (`0` ou `1`) même si le fait que le graphe soit supposé sans boucle incite à mettre `0`. Même remarque pour savoir si `i` doit appartenir ou non à `L[i]`.

2. Si `u` est une source de `G` alors pour tout autre sommet `v` de `G`, `(u,v)` est un arc de `G` donc `v` n'est pas une source de `G`. Il y a donc au maximum une seule source d'un même graphe.

3. On parcourt les lignes de la matrice tant qu'un sommet satisfaisant n'a pas été trouvé. Sachant que la matrice ne contient que des `0` et des `1` et que la diagonale est nulle, on se sert de la somme :

```
import numpy as np
def source_mat(M):
    n=np.shape(M)[0]
    source=-1
    for i in range(n):
        s=0
        for k in range(n):
            s+=M[i,k]
        if s==n-1:
            source=i
            break
    if source==-1:
        return False
    for k in range(n):
        if M[k,source]==1:
            return False
    return True
```

4. Dans le pire des cas, il y a deux boucles imbriquées parcourues n fois donc la complexité est quadratique c'est-à-dire en $\mathcal{O}(n^2)$.

5. On parcourt la liste d'adjacence jusqu'à ce qu'on trouve une liste de longueur $n-1$:

```
def source_list(L):
    n=len(L)
    source=-1
    for i in range(n):
        if len(L[i])==n-1:
            source=i
            break
    if source==-1:
        return False
    for l in L:
        if source in l:
            return False
    return True
```


6. L'accès à la longueur d'une liste étant à coût constant, on trouve une complexité linéaire c'est-à-dire en $\mathcal{O}(n)$. C'est plus efficace que pour le programme précédent.

7. On choisit une représentation sous forme de liste d'adjacence car il plus aisé de supprimer un élément d'une liste (c'est ici une liste) que se supprimer une ligne et une colonne d'une matrice. On utilise le « slicing » pour supprimer un élément de la liste.

```
def supprime(L,v):
    n=len(L)
    L=L[:v]+L[v+1:]
    for i in range(n-1):
        for k in range(len(L[i])):
            if L[i][k]==v:
                L[i]=L[i][:k]+L[i][k+1:]
                break
    return L
```

8. On commence par modifier `source_list` pour qu'il renvoie le sommet source universelle dans le cas où il existe et `False` sinon.

```
def source_L(L):
    n=len(L)
    source=-1
    for i in range(n):
        if len(L[i])==n-1:
            source=i
            break
    if source==-1:
        return False
    for l in L:
        if source in l:
            return False
    return source
```

On construit maintenant le programme en supprimant les sommets sources universelles au fur et à mesure s'ils existent :

```
def ocean(L):
    while L!=[[]]:
        s=source_L(L)
        if s==False:
            return False
        L=supprime(L,s)
    return True
```

9. Quitte à réordonner les sommets, un graphe océan à n sommets aura toujours une liste d'adjacence respectant la structure suivante : `L=[[],[0],[0,1], ⋯, [0,1,2, ⋯,n-1]]`. Il y a donc autant de graphes océan à n sommets qu'il y a de possibilités de réordonner les sommets soit $n!$.

On peut vérifier que les degrés d'incidence des sommets du graphe sont échelonnés de 0 à $n-1$ et que le sommet de degré k est incident à chaque sommet de degré j pour $j \in [\![0, k-1]\!]$.

Exercice 2 - Algorithmes de contrôle

1. Il y a deux problèmes dans la première fonction : l'oubli des deux points à la cinquième ligne qui produit une erreur de syntaxe et la variable de la fonction `range` à la quatrième ligne qui entraine un dépassement de capacité dans le cas d'une liste déjà triée. On propose la version corrigée suivante :

```
def fonction1(X):
        n = len(X)
        res = True
        for i in range(n-1):
                if X[i] >= X[i+1]:
                        res = False
        return(res)
```

   Dans le second programme, il n'y a pas de problème de syntaxe mais la fonction s'arrêtera en renvoyant `True` si les deux premiers éléments de la liste sont dans le bon ordre, ce qui n'est pas suffisant. On propose donc la version corrigée suivante :

```
def fonction2(X):
    n = len(X)
    for i in range(n-1):
        if X[i] < X[i+1]:
            None
        else:
            return(False)
    return(True)
```

2. Dans le pire des cas, les fonctions effectueront $n-1$ comparaisons.
3. Dans un premier temps, notons p l'entier supérieur ou égal à 1 tel que $N = 2p$ si N est pair ou $N = 2p+1$ si N est impair. La commande `N//2` renvoie alors p.

   La commande `X[0:N//2]` renvoie alors une liste de longueur p contenant les éléments de `X` d'indices 0 à $p-1$ et la commande `X[N//2:]` renvoie les éléments de `X` d'indices p à $N-1$.
4. Il ne faut pas oublier de comparer le dernier élément de `X[0:N//2]` avec le premier de `X[N//2:]` :

```
def test(X):
    N=len(X)
    if N<=1:
        return True
    else :
        if X[N//2-1]>= X[N//2]:
            return False
        else:
            return test(X[0:N//2]) and test(X[N//2:])
```

5. Pour tout $p \geq 0$, notons $b_p = B_{2^p}$. Soit $p \geq 0$. On a alors, $b_p = 1 + 2b_{p-1}$. La suite $(b_p)_{p\geq 0}$ est donc arithmético-géométrique de raison 2. Après calculs, on trouve : $b_p = 2^p(b_0+1) - 1 = 2^p + 1$ car $b_0 = B_1 = 0$. Soit $n \in \mathbb{N}$ tel qu'il existe $p \in N^*$ tel que $n = 2^p$. Alors $p = \log_2(n)$ et donc $B_n = b_p = n+1$.
6. Si n n'est pas une puissance de deux, on peut raisonnablement supposer que la fonction $n \mapsto B_n$ est croissante. Soit donc p tel que $2^p \leq n \leq 2^{p+1}$. On aura alors, $2^p + 1 \leq B_n \leq 2^{p+1} + 1$. En particulier, $\frac{n}{2} \leq B_n \leq 2n+1$ donc la complexité est linéaire. La fonction récursive ne présente pas d'intérêt en terme de complexité par rapport aux deux fonctions précédemment introduites.