Correction du DS1

Exercice I : graphe eulérien

1. a) Le degré est tout simplement la longueur de la liste d'adjacence du sommet `s` :

```
def degre(G,s) :
    return len(G[s])
```

b) La fonction `len` possède une complexité constante donc $\boxed{O(1)}$

2. Pour le parcours en profondeur, on peut se contenter de travailler avec une liste pour simuler la pile, à condition de se limiter aux méthodes `.pop()` et `.append()` qui agissent à coût constant. Inutile de mémoriser les sommets rencontrés dans une liste, il suffit de les compter (avec la variable `nb_visited`) ; on vérifie à la fin si on est passé par tous les sommets de `G` en comparant ce compteur avec `len(G)` :

```
def connexe(G) :
    p = []
    vus = {x:False for x in G}
    nb_visited = 0
    p.append(0)
    while len(p)>0 :
        som = p.pop()
        if not vus[som] :
            vus[som] = True
            nb_visited += 1
        for s in G[som] :
            if not vus[s] :
                p.append(s)
    return nb_visited == len(G)
```

3. On commence par tester si le graphe est connexe, puis les degrés des sommets, en interrompant la fonction dès qu'une « anomalie » apparaît :

```
def eulerien(G) :
    if not connexe(G) :
        return False
    for s in G :
        if degre(G,s)%2 == 1 :
            return False
    return True
```

4. a) Même principe que pour la fonction `degre` :

```
def isole(G,s) :
    return len(G[s])==0
```

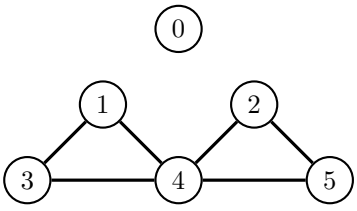
b) On compte toutes les arêtes, en faisant attention qu'une arête de `s` vers `t` sera comptée deux fois : une fois comme partant de `s` et une fois en partant de `t`. Attention à la division finale par 2 qui doit renvoyer un entier :

```
def aretes(G) :
    r = 0
    for s in G :
        r += len(G[s])
    return r//2
```

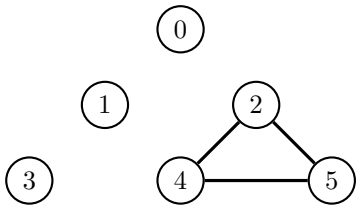
c) Il faut penser à supprimer l'arête dans les deux listes d'adjacence (sinon le graphe restant sera orienté), inutile de placer un `return` puisque la modification se fait sur place :

```
def supprimer(G,s,t) :
    G[t].remove(s)
    G[s].remove(t)
```

5. a) Une boucle issue de 0 est `[0,1,2,0]` : à chacun des sommets 1 et 2, le choix du sommet suivant est imposé par le sujet du DS. Le graphe après suppression des arêtes empruntées est



On parcourt les sommets de `[0,1,2,0]` : 0 est isolé, 1 ne l'est pas donc on détermine une boucle issue de 1, qui est `[1,3,4,1]`. On insère cette liste dans la précédente à la place du sommet 1 initial pour obtenir la liste `0,1,3,4,1,2,0]`. Le graphe devient



Il reste des arêtes donc on réitère le processus à partir de `[0,1,3,4,1,2,0]` : les sommets 0,1 et 3 sont isolés ; une boucle issue de 4 est `[4,2,5,4]` à la suite de laquelle toutes les arêtes sont parcourues. On insère cette boucle dans la liste précédente à la place du sommet 4 pour obtenir le cycle `[0,1,3,4,2,5,4,1,2,0]`

b) Une boucle peut passer plusieurs fois par son sommet initial (les sommets intermédiaires peuvent aussi apparaître plusieurs fois dans la boucle) : une boucle depuis 1 dans le graphe initial précédent donnerait (toujours en choisissant le sommet de plus petit numéro en cas de choix) `[1,0,2,1,3,4,1]`

Le graphe étant eulérien, tous les sommets sont de degré pair. Lorsqu'on part de `s`, on enlève une arête donc il reste un nombre impair d'arêtes en `s` alors qu'à chaque fois qu'on passe par un autre sommet du graphe, on enlève deux arêtes (une pour y arriver et une autre pour en repartir). Pour qu'un sommet devienne isolé, il ne doit posséder qu'une seule arête (donc un nombre impair d'arêtes) avant qu'on y accède : la seule possibilité est donc de terminer sur `s`, le sommet de départ, qui sera donc isolé après le parcours de cette boucle.

c) Une boucle se termine lorsqu'on arrive sur un sommet qui ne possède plus d'arête, donc un sommet isolé :

```
def boucle(G,s) :
    L = [s]
    while not isole(G,s) :
        t = G[s][0]
        L.append(t)
        supprimer(G,s,t)
        s = t
    return L
```

d) La recherche se termine lorsque la liste `L` contient un élément de plus que le nombre d'arêtes du graphe (car on doit répéter le sommet initial à la fin pour décrire la dernière arête). On insère les boucles trouvées par slicing, en faisant attention de supprimer le sommet que l'on remplace dans la liste :

```
def cycle(G) :
    p = aretes(G)
    L = boucle(G,0)
    while len(L) <= p :
        i = 0
        while isole(G,L[i]) :
            i += 1
        s = L[i]
        M = boucle(G,s)
        L = L[:i]+M+L[i+1:]
    return L
```

e) Le coût de la fonction `aretes` est $O(n)$ car la fonction `len` a un coût constant.

Pour la fonction `boucle` : le coût de la fonction `isole` est $O(1)$, le `while` est exécuté au plus p fois, le coût de la méthode `.append(t)` est $O(1)$ alors que celui de la fonction `supprimer` est $O(n)$ (utilisation de `.remove()` sur deux listes de longueurs au plus n). La fonction `boucle` a donc une complexité en $O(np)$.

La liste `L` est de longueur maximale $p+1$ donc la recherche du premier sommet non isolé (fonction de coût $O(1)$) est en $O(p)$, l'utilisation qui suit de la fonction `boucle` est en $O(np)$, le coût du slicing en $O(p)$. Ces opérations, dans la boucle `while` sont exécutées au plus p fois donc la complexité finale est $\boxed{O\left(np^2\right)}$

Exercice II : algorithme de Kaprekar

1. a) On obtient

k	n	r	L
0	12	3	[3]
1	1	2	[3,2]
2	0	1	[3,2,1]

`f(123,3)` renvoie `[3,2,1]`

k	n	r	L
0	2	2	[2]
1	0	2	[2,2]
2	0	0	[2,2,0]

`f(22,3)` renvoie `[2,2,0]`

b) La fonction `f` renvoie la liste des `p` chiffres de `n`

2. Un seul balayage de `L` suffit

```
def occ(L) :
    F = [0]*10
    for k in L :
        F[k] += 1
    return F
```

3. a) Les chiffres de `n` sont donc (dans un certain ordre) 0,0,2,4,4,5,7,7,7,8 donc `M=8777544200` et `m=24457778`

b) On peut le faire directement (la variable `s` permet de savoir à quel chiffre on en est pour ajuster la puissance de 10 à considérer)

```
def K(n,p) :
    L = occ(f(n,p))
    M,m = 0,0
    s = 0
    for k in range(10) :
        for i in range(L[k]) : # boucle vide si L[k]=0
            M += k*10**s
            m += k*10**(p-s-1)
            s += 1
    return M,m
```

On peut aussi le faire avec des chaines de caractères qui rendent les concaténation plus faciles puis les convertir en entiers à la fin

```
def Kbis(n,p) :
    L = occ(f(n,p))
    M,m = '','' 
    for k in range(10) :
        for i in range(L[k]):
            M = str(k)*L[k]+M # sans effet si L[k]=0
            m = m+str(k)*L[k]
    return int(M),int(m)
```

4. Il faut penser à rajouter la première répétition dans la liste avant de la renvoyer

```
def Kaprekar(n,p) :
    L = [n]
    M,m = K(n,p)
    while M-m not in L :
        n = M-m
        L.append(n)
        M,m = K(n,p)
    L.append(M-m)
    return L
```