Toute calculatrice autorisée.
Le sujet comporte un total de 3 exercices
que vous pouvez traiter dans l'ordre de votre choix.

Les codes demandés devront être **clairement** commentés !

EXERCICE 1. Le tri de SHELL. 10 points

En 1959, Donald SHELL a proposé une amélioration du tri par insertion. Le principe général est exposé ci-dessous.

Avant de trier par insertion une liste L donnée, on va en trier des sous-listes définies par des sauts (gaps). Pour un gap g entier donné, on va trier par insertion les g sous-listes correspondant aux éléments de L dont les indices suivent des progressions arithmétiques de raison g :

- $0, g, 2g, 3g, \ldots$
- $1, g+1, 2g+1, 3g+1, \ldots$
- $2, g+2, 2g+2, 3g+2, \ldots$
- …
- $g-1, 2g-1, 3g-1, 4g-1, \ldots$

On obtient ainsi une nouvelle liste sur laquelle on recommence l'opération précédente avec un gap plus petit jusqu'à terminer avec un gap égal à 1.

Les tris des sous-listes à l'aide de gaps strictement supérieurs à 1 peuvent donc être considérés comme des tris préparatoires à un tri par insertion classique (gap égal à 1). Cependant, le choix des valeurs des gaps n'est pas anodin et on a pu constater que certaines valeurs de gap pouvaient rendre cette méthode de tri instable. Expérimentalement, les premières valeurs optimales à retenir pour les gaps sont : 1, 4, 10, 23, 57, 132, 301, 701.

Ainsi, pour une liste de longueur 200, on utilisera successivement les gaps 132, 57, 23, 10, 4 et 1.

A titre d'exemple, considérons la liste initiale suivante :

```
[45,7,98,2,12,59,20,46,25,32,11,10,25]
```

La longueur de la liste est égale à 13.

Nous commençons donc par travailler avec un gap de 10.
On s'intéresse à la sous-liste des éléments d'indices 0 et 10 :

[45,7,98,2,12,59,20,46,25,32,11,10,25]

Un tri par insertion appliqué à cette sous-liste donne :

[11,7,98,2,12,59,20,46,25,32,45,10,25]

On continue avec la sous-liste des éléments d'indices 1 et 11 :

[11,7,98,2,12,59,20,46,25,32,45,10,25]

Cette sous-liste est triée et n'engendre donc aucune modification.
On termine avec la sous-liste des éléments d'indices 2 et 12 :

[11,7,98,2,12,59,20,46,25,32,45,10,25]

Un tri par insertion appliqué à cette sous-liste donne :

[11,7,25,2,12,59,20,46,25,32,45,10,98]

Il n'y a plus de sous-liste correspondant à un gap de 10, on considère alors un gap égal à 4.
On s'intéresse à la sous-liste des éléments d'indices 0, 4, 8 et 12 :

[11,7,25,2,12,59,20,46,25,32,45,10,98]

Cette sous-liste est triée et n'engendre aucune modification.
On continue avec la sous-liste des éléments d'indices 1, 5 et 9 :

[11,7,25,2,12,59,20,46,25,32,45,10,98]

Un tri par insertion appliqué à cette sous-liste donne :

[11,7,25,2,12,32,20,46,25,59,45,10,98]

On continue avec la sous-liste des éléments d'indices 2, 6 et 10 :

[11,7,25,2,12,32,20,46,25,59,45,10,98]

Un tri par insertion appliqué à cette sous-liste donne :

[11,7,20,2,12,32,25,46,25,59,45,10,98]

On termine avec la sous-liste des éléments d'indices 3, 7 et 11 :

[11,7,20,2,12,32,25,46,25,59,45,10,98]

Un tri par insertion appliqué à cette sous-liste donne :

[11,7,20,2,12,32,25,10,25,59,45,46,98]

Il n'y a plus de sous-liste correspondant à un gap de 4, on considère alors un gap égal à 1 et on applique donc un tri par insertion classique à la liste ci-dessus.

A titre de rappel, nous fournissons un code Python correspondant à une fonction réalisant un tri par insertion en place d'une liste `L` passée en argument :

```
def tri_insert(L):
    for i in range(1,len(L)):
        x = L[i]
        k = i
        while k > 0 and x < L[k-1]:
            L[k] = L[k-1]
            k -= 1
        L[k] = x
```

[Q.1 / 4 points] Modifier la fonction `tri_insert` ci-dessus pour construire une fonction `tri_insert_gap` qui recevra en argument : une liste `L` à trier et un gap `g`. Cette fonction, comme dans l'exemple ci-dessus, triera par insertion les `g` sous-listes de `L` ayant pour premiers éléments les éléments de `L` d'indices 0, 1, …, `g-1`.

La fonction `tri_insert_gap` effectuera des tris en place sans créer la moindre sous-liste intermédiaire.

Nous avons indiqué plus haut les premiers gaps optimaux obtenus expérimentalement. Au-delà de 701, les autres gaps sont obtenus en considérant que la suite des gaps est une suite géométrique de raison 2,3. Un gap étant entier, on retiendra bien sûr la partie entière des valeurs ainsi obtenues. Par exemple, après 701 on a aura : $\mathrm{E}(701\times 2,3)=1612$ puis $\mathrm{E}(1612\times 2,3)=3707$.

[Q.2 / 3 points] Ecrire une fonction Python `gap_gen` qui recevra en argument un entier `N` (cet entier correspond à la longueur de la liste à trier et est supposé supérieur ou égal à 2) et renverra la liste des gaps à utiliser pour le tri de la liste initiale.

[Q.3 / 3 points] Ecrire enfin une fonction Python `tri_SHELL` qui recevra en argument une liste L à trier et en effectuera un tri de SHELL en place. On utilisera bien sûr les fonctions `gap_gen` et `tri_insert_gap` des questions 1 et 2.

EXERCICE 2. La fonction d'Ackermann — 15 points

La fonction d'Ackermann a été initialement introduite en 1928 comme une fonction de trois variables entières naturelles. En fixant la première variable à la valeur 2, Peter a obtenu une fonction de deux variables entières naturelles classiquement appelée « fonction d'Ackermann » et définie récursivement comme suit :

$$A:(m,n)\mapsto\begin{cases}n+1 & \text{si } m=0\\ A(m-1,1) & \text{si } m>0 \text{ et } n=0\\ A(m-1,A(m,n-1)) & \text{si } m>0 \text{ et } n>0\end{cases}$$

[Q.1 / 3 points] Ecrire une fonction Python `AP` qui reçoit en argument deux variables entières `m` et `n` et renvoie $A(m,n)$ en implémentant la définition récursive ci-dessus.

Votre fonction devra d'abord vérifier que `m` et `n` sont bien entiers et positifs. Dans le cas contraire, elle devra renvoyer un message d'erreur approprié.

Pour tout entier naturel *n*, on a évidemment : $A(0,n)=n+1$.

[Q.2 / 1 point] Vérifier que l'on a : $A(1,0)=2$.

[Q.3 / 3 points] Démontrer par récurrence que l'on a : $\forall n\in\mathbb{N},\ A(1,n)=2+(n+3)-3$.

L'expression de $A(1,n)$ ci-dessus est volontairement non simplifiée car on a, pour tout *n* entier naturel :

$$A(2,n)=2\times(n+3)-3$$
$$A(3,n)=2^{n+3}-3$$
$$A(4,n)=\underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{n+3\text{ "2" empilés}}-3$$

Ainsi :

$$A(3,4)=2^{4+3}-3=2^{7}-3=128-3=125$$
$$A(4,1)=2^{2^{2^{2}}}-3=2^{2^{4}}-3=2^{16}-3=65536-3=65533$$

Rappelons que les exponentiations successives se calculent de la droite vers la gauche.

Ainsi, malgré la simplicité calculatoire apparente de la fonction d'Ackermann, les valeurs prises augmentent (vraiment) très vite…

On peut cependant continuer de donner des expressions « simples » pour $A(5,n)$, $A(6,n)$, … en utilisant la notation des puissances itérées de Knuth.

On note par exemple : $2^{n+3} = 2 \uparrow (n+3)$ et $\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n+3 \text{ "2" empilés}} = 2 \uparrow\uparrow (n+3)$.

On montre alors que l'on a : $A(5,n) = 2 \uparrow\uparrow\uparrow (n+3) - 3 = \underbrace{2 \uparrow\uparrow \left(2 \uparrow\uparrow \left(2 \uparrow\uparrow \left(\ldots \uparrow\uparrow 2\right)\right)\right)}_{\text{"2" apparaît } n+3 \text{ fois}} - 3$

Par exemple : $A(5,0) = 2 \uparrow\uparrow\uparrow 3 - 3 = 2 \uparrow\uparrow 2 \uparrow\uparrow 2 - 3 = 2 \uparrow\uparrow \left(2 \uparrow\uparrow 2\right) - 3$.

Or : $2 \uparrow\uparrow 2 = 2^2 = 4$. Donc $A(5,0) = 2 \uparrow\uparrow \left(2 \uparrow\uparrow 2\right) - 3 = 2 \uparrow\uparrow 4 - 3 = 2^{2^{2^2}} - 3 = 65533$.

On vérifie en passant que l'on a retrouvé $A(4,1)$.

La notation des puissances itérées de Knuth est ainsi fondamentalement récursive :

$$a \underbrace{\uparrow\uparrow\uparrow \ldots \uparrow\uparrow}_{\substack{\text{la flèche } \uparrow \\ \text{apparaît } n \text{ fois}}} b = \underbrace{a \underbrace{\uparrow\uparrow\uparrow \ldots \uparrow\uparrow}_{\substack{\text{la flèche } \uparrow \\ \text{apparaît } n-1 \text{ fois}}} a \underbrace{\uparrow\uparrow\uparrow \ldots \uparrow\uparrow}_{\substack{\text{la flèche } \uparrow \\ \text{apparaît } n-1 \text{ fois}}} a \underbrace{\uparrow\uparrow\uparrow \ldots \uparrow\uparrow}_{\substack{\text{la flèche } \uparrow \\ \text{apparaît } n-1 \text{ fois}}} a \ldots \underbrace{\uparrow\uparrow\uparrow \ldots \uparrow\uparrow}_{\substack{\text{la flèche } \uparrow \\ \text{apparaît } n-1 \text{ fois}}} a}_{\text{"}a\text{" apparaît } b \text{ fois}}$$

En utilisant enfin la notation condensée : $\underbrace{\uparrow\uparrow\uparrow \ldots \uparrow\uparrow}_{\substack{\text{la flèche } \uparrow \\ \text{apparaît } k \text{ fois}}} = \uparrow^k$, on peut définir la puissance itérée de Knuth de *a* et *b* au rang *n* par :

$$a \uparrow^n b = \begin{cases} a^b & \text{si } n = 1 \\ 1 & \text{si } b = 0 \\ a \uparrow^{n-1} \left(a \uparrow^n (b-1)\right) & \text{sinon} \end{cases}$$

[Q.4 / 4 points] Ecrire une fonction Python `KnuthPI` qui reçoit en argument trois variables entières `a`, `b` et `n` et renvoie $a \uparrow^n b$.
Votre fonction devra d'abord vérifier que `a`, `b` et `n` sont bien entiers avec `b` positif et `n` positif non nul. Dans le cas contraire, elle devra renvoyer un message d'erreur approprié.

A l'aide des notations précédentes, on montre que l'on a finalement, pour tout entier naturel *m* supérieur ou égal à 3 et tout entier naturel *n* :

$$A(m,n) = 2 \uparrow^{m-2} (n+3) - 3$$

[Q.5 / 4 points] Ecrire une fonction Python `AP2` qui reçoit en argument deux variables entières `m` et `n` et renvoie $A(m,n)$ en implémentant le résultat ci-dessus et en utilisant la fonction `KnuthPI` lorsque $m \geq 3$ (pour les autres cas, on utilisera les formules appropriées).
Votre fonction devra d'abord vérifier que `m` et `n` sont bien entiers et positifs. Dans le cas contraire, elle devra renvoyer un message d'erreur approprié.

EXERCICE 3. Des chaînes eulériennes — 15 points

I. Eléments historiques et définitions

Historiquement parlant, la théorie des graphes, est née avec le géant des mathématiques, Leonhard EULER (1707-1783), même s'il ne l'avait pas vraiment baptisée de la sorte à l'époque…

En des termes modernes, le problème qu'Euler résolut consistait à chercher une… « chaîne eulérienne » dans un graphe simple non orienté.

Définitions

On dit qu'une chaîne d'un graphe simple non orienté est « eulérienne » si elle passe une fois et une seule par chaque arête du graphe.

On dit d'un graphe qu'il s'agit d'un « graphe eulérien » s'il admet un cycle eulérien, c'est-à-dire une chaîne eulérienne qui est un cycle.

Euler a établi les deux théorèmes suivants :

Théorème d'existence d'une chaîne eulérienne.

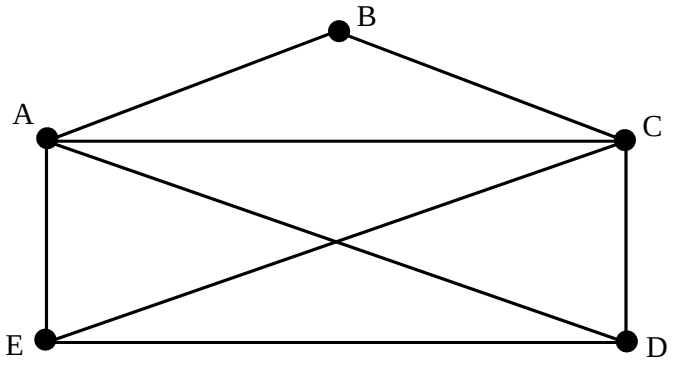
Un graphe simple non orienté admet une chaîne eulérienne si, et seulement si, le nombre de sommets de degré impair est égal 0 ou 2. Dans le deuxième cas, les sommets de degré impair sont les extrémités de toute chaîne eulérienne.

Théorème d'existence d'un cycle eulérien.

Un graphe simple non orienté admet un cycle eulérien si, et seulement si, le nombre de sommets de degré impair est égal 0.

II. L'enveloppe

On considère le graphe suivant :



[Q.1 / 2 points] Le graphe ci-dessus :

a) admet-il une chaîne eulérienne ? Si oui, en donner une.

b) admet-il un cycle eulérien ? Si oui, en donner un.

III. Existence

[Q.2 / 4 points] Ecrire une fonction Python `OddVertices` qui reçoit en argument un tableau numpy `A` correspondant à la matrice d'adjacence d'un graphe simple non orienté (on ne cherchera pas à valider qu'il s'agit bien d'une telle matrice) et renvoie la liste (éventuellement vide) des indices des sommets de degré impair.

[Q.3 / 3 points] Ecrire une fonction Python `CCEuler` qui reçoit en argument un tableau numpy `A` correspondant à la matrice d'adjacence d'un graphe simple non orienté (on ne cherchera pas à valider qu'il s'agit bien d'une telle matrice) et renvoie :

a) le booléen `False` si le graphe n'admet pas de chaîne eulérienne.

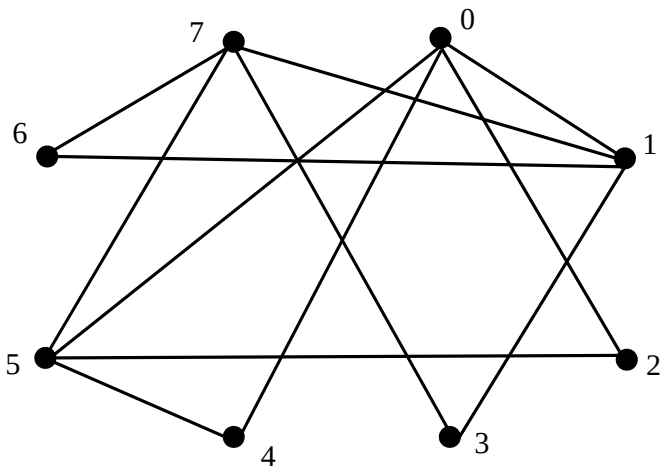
b) Le booléen `True` si le graphe admet un cycle eulérien.

c) Un tuple constitué du booléen `True` et des indices des deux sommets de degré impair si le graphe admet exactement deux sommets de degré impair.

La fonction `CCEuler` utilisera la fonction `OddVertices` de la question précédente.

IV. Recherche

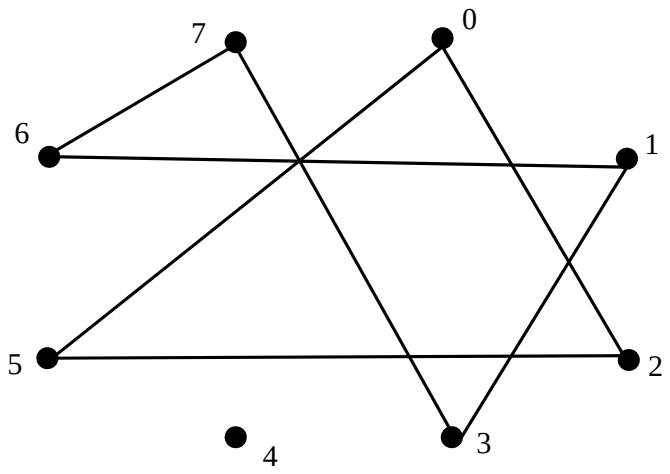
Pour décrire un algorithme permettant d'obtenir une chaîne ou un cycle eulérien, nous allons partir du graphe G d'ordre 8 suivant :



Tous les sommets étant de degré pair, le graphe G est eulérien.

1ère étape : on détermine une chaîne (un cycle ici) sans répétition d'arêtes

Par exemple : (4,5,7,1,0,4). On retire alors les arêtes correspondantes du graphe G et on obtient le graphe G' suivant :



Remarque importante : si nous avions eu affaire à un graphe admettant « seulement » une chaîne eulérienne mais pas de cycle eulérien, on aurait choisi pour extrémités de cette première chaîne, les deux sommets de degré impair.

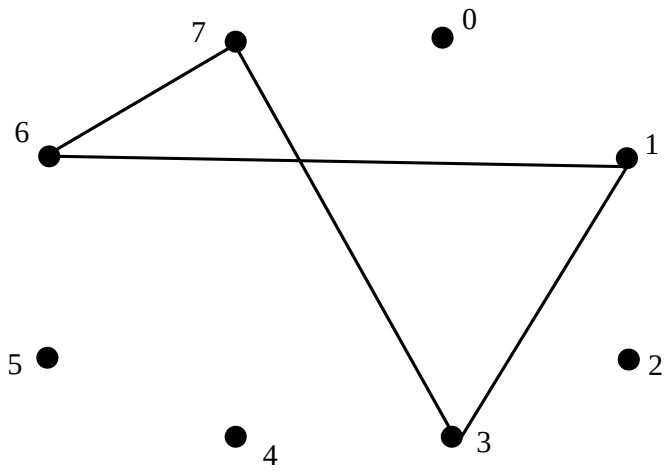
2ème étape : on choisit un sommet non isolé S de G' et on construit un cycle passant par S

On peut choisir ici le sommet 5 et on obtient le cycle (5,0,2,5).

3ème étape : on insère le cycle obtenu dans la chaîne initiale.

(4,5,7,1,0,4) devient (4,**5,0,2,5**,7,1,0,4).

4ème étape : on retire de G' les arêtes du cycle précédemment obtenu



On recommence alors les étapes 2, 3 et 4 tant que le graphe obtenu à l'étape 4 comporte encore des arêtes.

[Q.4 / 3 points] Ecrire une fonction Python `Insert` qui reçoit en argument deux listes d'entiers (représentant des listes d'indices de sommets d'un même graphe) `L` et `M`, `L` contenant le premier élément de `M`, et renvoyant une liste `N` obtenue en remplaçant dans `L` une occurrence de `M[0]` par les éléments de `M`.

Par exemple avec `L=[4,5,7,1,0,4]` et `M=[5,0,2,5]`, la fonction `Insert` devra renvoyer la liste :

`[4,5,0,2,5,7,1,0,4]`

[Q.5 / 3 points] Ecrire une fonction Python `Delete` qui reçoit en argument un tableau `numpy A` correspondant à la matrice d'adjacence d'un graphe simple non orienté et une liste `L` d'entiers représentant un cycle de ce graphe. La fonction renverra le tableau `A` mis à jour en mettant à zéro les coefficients correspondant aux arêtes du cycle.

La fonction devra vérifier que le premier élément et le dernier élément de `L` sont identiques.

[BONUS] Proposer une fonction Python `SingleEdgesChain` qui reçoit en argument un tableau `numpy A` correspondant à la matrice d'adjacence d'un graphe simple non orienté et l'indice `i` d'un sommet de ce graphe. La fonction renverra un cycle passant par le sommet d'indice i, chaque arête du cycle étant empruntée une fois et une seule.

FIN DU SUJET