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. La Formule de Bailey-Borwein-Plouffe.

Partie A : quelques manipulations hexadécimales

Aller : de l'écriture hexadécimale à l'écriture décimale

Soit N un réel positif.

L'écriture en base 16 d'un tel nombre (écriture dite « hexadécimale ») utilise 16 « chiffres » classiquement notés : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E et F.
Ainsi, si on a : $N = 1C3F32,E3_{16}$ (l'indice « 16 » indique qu'il s'agit de l'écriture en base 16 du réel N), l'écriture en base 10 de N sera :

$$\begin{aligned} N &= 1C3F32,E3_{16} \\ &= 1\times16^5 + 12\times16^4 + 3\times16^3 + 15\times16^2 + 3\times16^1 + 2\times16^0 + 14\times16^{-1} + 3\times16^{-2} \\ &= 1\,048\,576 + 786\,432 + 12\,288 + 3\,840 + 48 + 2 + 0,875 + 0,011\,718\,75 \\ &= 1\,851\,186,886\,718\,75 \end{aligned}$$

1. Ecrire une fonction Python `Hexa2Dec` qui reçoit en argument une chaîne de caractères `s` (on ne cherchera pas à vérifier la correction de cette chaîne) correspondant à l'écriture d'un réel en base 16 et renvoyant l'écriture en base 10 de ce nombre.

   On rappelle que la méthode `split` d'une chaîne de caractères permet de « découper » celle-ci en sous-chaînes séparées par le séparateur spécifié en argument. Par exemple, si on a `s='1C3F32,E3'`, l'instruction `s.split(',')` renverra la liste `['1C3F32','E3']`.

Et retour… : de l'écriture décimale à l'écriture hexadécimale

Dans cette partie, on se limite au cas où N est un entier naturel.

Un tel entier est encadré, de façon unique, par deux puissances successives de 16 :

$$16^n \leq N < 16^{n+1}$$

On en tire immédiatement : $n\ln 16 \le \ln N < (n+1)\ln 16$ puis $n \le \dfrac{\ln N}{4\ln 2} < n+1$. D'où, finalement :

$$n = \mathrm{E}\left(\frac{\ln N}{4\ln 2}\right) = \mathrm{E}\left(\frac{1}{4}\log_2(N)\right)$$

Le premier chiffre de l'écriture hexadécimale de *N* sera alors le quotient *q* de la division euclidienne de *N* par 16^n. Pour obtenir le second chiffre (éventuel) de cette écriture, on applique la démarche précédente à $N - q\times 16^n$ (si cette différence est non nulle).

2. On propose ci-dessous le code (incomplet !) d'une fonction Python `Dec2Hexa` qui reçoit en argument un entier naturel `N` et renvoie, sous forme d'une chaîne de caractères, l'écriture en base 16 de ce nombre.

```
def Dec2Hexa(N):
      Coeff=('0','1','2','3','4','5','6','7','8','9','A','B','C','D','E'
      ,'F')
      if N <= 15:
         return(Coeff[N])
      else:
         # Conctruction de la liste de tuples (coefficient,exposant)
         L = []
         while N > 0:
               n = int(0.25 * log(N,2))
               p = 16**n
               (q,r) = divmod(N,p)
               L.append((q,n))
               N -= q * p
         # Construction de la chaîne de caractères correspondant à
         l'écriture hexadécimale de N.
         s = ''
```

PARTIE A COMPLETER

```
         return s
```

Compléter le code proposé.

Partie B : utilisation de la formule de Bailey-Borwein-Plouffe (1995)

Considérations générales

La formule, en abrégé « formule BBP », est la suivante :

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k}\left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\right)$$

En la récrivant comme suit, elle permet de calculer la $(N+1)$-ième décimale de π en base 16 (multiplier par 16^N en base 16 équivaut à multiplier par 10^N en base 10 : la virgule est décalée de *N* positions vers la droite) : cette décimale sera la première décimale de $16^N \times \pi$.

$$16^N \times \pi = \sum_{k=0}^{\infty} 16^{N-k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right)$$

En posant $S_N(a) = \sum_{k=0}^{\infty} \frac{16^{N-k}}{8k+a}$, on a : $16^N \times \pi = 4S_N(1) - 2S_N(4) - S_N(5) - S_N(6)$.

Enfin, on a : $S_N(a) = \sum_{k=0}^{\infty} \frac{16^{N-k}}{8k+a} = \sum_{k=0}^{N-1} \frac{16^{N-k}}{8k+a} + \sum_{k=N}^{\infty} \frac{16^{N-k}}{8k+a} = A_N(a) + B_N(a)$.

Calcul de $A_N(a)$

On a : $A_N(a) = \sum_{k=0}^{N-1} \frac{16^{N-k}}{8k+a}$.

On ne s'intéresse pas à la partie entière de $A_N(a)$.

On effectue donc la division euclidienne de 16^{N-k} par $8k+a$. Elle s'écrit :

$$16^{N-k} = q(k) \times (8k+a) + r(k) \text{ avec } 0 \le r(k) < 8k+a$$

Il vient alors : $A_N(a) = \sum_{k=0}^{N-1} \frac{16^{N-k}}{8k+a} = \sum_{k=0}^{N-1} \frac{q(k) \times (8k+a) + r(k)}{8k+a} = \sum_{k=0}^{N-1} q(k) + \sum_{k=0}^{N-1} \frac{r(k)}{8k+a}$.

Il nous faut donc calculer $A'_N(a) = \sum_{k=0}^{N-1} \frac{r(k)}{8k+a}$.

3. Ecrire une fonction Python `CalcAP` qui reçoit comme argument deux entiers correspondant à *N* et *a* et renvoie $A'_N(a)$.

4. On cherche la partie fractionnaire de $4A_N(1) - 2A_N(4) - A_N(5) - A_N(6)$. Il n'y a aucune raison qu'elle soit égale à $4A'_N(1) - 2A'_N(4) - A'_N(5) - A'_N(6)$ qui une somme algébrique de parties fractionnaires. Par exemple, avec $N = 10$, on obtient : $4A'_{10}(1) - 2A'_{10}(4) - A'_{10}(5) - A'_{10}(6) \simeq -2,365$ (valeur arrondie au millième). La partie fractionnaire de $4A_{10}(1) - 2A_{10}(4) - A_{10}(5) - A_{10}(6)$ vaut donc $3 - 2,365 = 0,635$ (valeur arrondie au millième).

   Ecrire une fonction Python `SPA` qui reçoit comme argument un entier correspondant à *N* et renvoie la partie fractionnaire de $4A_N(1) - 2A_N(4) - A_N(5) - A_N(6)$.

   Rappel : la fonction Python `floor` correspond à la fonction mathématique partie entière et se trouve dans le module `math`.

Calcul de $B_N(a)$

On a : $B_N(a) = \sum_{k=N}^{\infty} \frac{16^{N-k}}{8k+a}$.

On a affaire à une série convergente dont le comportement asymptotique est proche de celui d'une série géométrique de raison 1/16. Plus précisément, en notant $b_k = \frac{16^{N-k}}{8k+a}$, il vient :

$$\frac{b_{k+1}}{b_k} = \frac{\frac{16^{N-(k+1)}}{8(k+1)+a}}{\frac{16^{N-k}}{8k+a}} = \frac{1}{16} \times \frac{8k+a}{8k+8+a} = \frac{1}{16} \times \left(1 - \frac{8}{8k+8+a}\right)$$

Comme *N* est grand (c'est pour cela que l'on utilise la formule BBP !), le premier terme de la somme est petit devant 1 et le rapport ci-dessus est très proche de 1/16 dans la formule de $B_N(a)$ et chaque nouveau terme calculé comporte au moins un zéro de plus après la virgule que dans le précédent. Pour obtenir $B_N(a)$ avec une précision de *P* chiffres après la virgule (si l'on souhaite obtenir *P* décimales de π et pas seulement une), on va en fait calculer (on tient compte des effets de retenue en sommant jusqu'à $N+P+10$) :

$$B'_N(a) = \sum_{k=N}^{N+P+10} \frac{16^{N-k}}{8k+a}$$

5. Ecrire une fonction Python `CalcBP` qui reçoit comme argument trois entiers correspondant à *N*, *P* et *a* et renvoie $B'_N(a)$.

6. Ecrire une fonction Python `SPB` qui reçoit comme arguments deux entiers correspondant à *N* et *P* et renvoie $4B'_N(1) - 2B'_N(4) - B'_N(5) - B'_N(6)$.

Pour obtenir les *P* décimales $N+1$ à $N+P$ de π en base 16, on procède alors comme suit :

- On demande *N* et *P* à l'utilisateur.
- On appelle les fonctions `SPA` et `SPB` avec les arguments appropriés et on somme les résultats renvoyés.
- On multiplie cette somme par 16^P et on considère la partie entière du résultat obtenu.
- On détermine l'écriture hexadécimale de cette partie entière (Attention ! La chaîne de caractères obtenue doit comporter *P* caractères !).

7. Ecrire un script Python codant la démarche précédente.

EXERCICE 2. Le problème du drapeau hollandais (ou : Dijkstra, le retour).

Comme vous le savez, Edsger DIJKSTRA était hollandais. Vous avez pu faire sa connaissance à travers son célèbre algorithme de détermination d'un plus court chemin entre deux sommets d'un graphe pondéré. Le revoici, mais cette fois à l'origine d'un bel algorithme de tri…

C'est le drapeau de son pays (le nôtre aurait tout autant fait l'affaire ☺) qui lui a inspiré le problème suivant :

« *Soit une liste d'objets alignés dans le désordre et uniquement colorés en bleu, blanc ou rouge. Comment trier ces objets de sorte que les objets bleus soient situés à gauche, les objets blancs au centre et les objets rouges à droite ?* »

Par exemple si la situation initiale est la suivante (On utilise les lettres B, W et R pour désigner les couleurs bleu, blanc et rouge respectivement car l'impression noir et blanc constitue un réel obstacle à l'illustration de la situation… ☺) :

R R B W R B W W R B R

alors l'algorithme devra fournir la situation finale suivante :

B B B W W W R R R R R

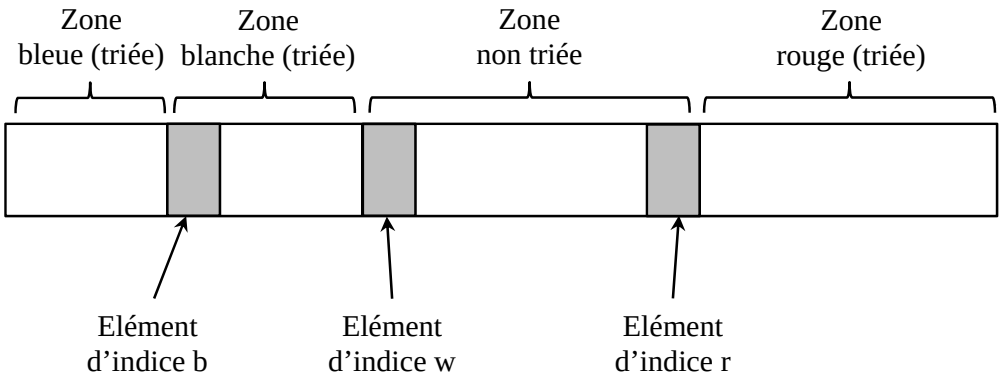
Une implémentation de l'algorithme recevra en fait en entrée une liste de *n* entiers ne prenant que les valeurs 0 (pour le bleu), 1 (pour le blanc) ou 2 (pour le rouge).
Remarque : un ou deux couleurs peuvent parfaitement être absentes de la liste !

Par exemple, pour la situation initiale ci-dessus, la liste passée en argument sera :

```
L = [2,2,0,1,2,0,1,1,2,0,2]
```

Le tri doit se faire <u>en place</u> et ne pas utiliser de tableau(x) auxiliaire(s). Pendant l'exécution de l'algorithme, la liste comporte 4 zones (de la gauche vers la droite) : une zone dont tous les éléments sont bleus (des 0), une zone dont tous les éléments sont blancs (des 1), une zone dont les éléments sont de couleur quelconque (ce sont les éléments non triés) et, enfin, une zone dont tous les éléments sont rouges (des 2).
On va gérer en permanence trois variables :

- `b`, correspondant à l'indice du premier élément de la liste qui n'est pas bleu.
- `w`, correspondant à l'indice du premier élément de la liste qui n'est pas blanc.
- `r`, correspondant à l'indice du dernier élément de la liste qui n'est pas rouge.



Initialement, les zones triées sont vides (tous les éléments sont considérés comme non triés) et, en tenant compte de l'indexation des listes Python, on aura :

`b=0`, `w=0` et `r=n-1`

On a alors l'algorithme :

```
Tant que w <= r faire :
     Si L[w] = blanc alors :
          w = w + 1
     Sinon :
          si L[w] = bleu alors :
               Echanger L[w] et L[b]
               b = b + 1
               w = w + 1
          sinon :
               Echanger L[w] et L[r]
               r = r – 1
Fin Tant que
```

1. Ecrire une fonction Python `DutchFlag` qui code cet algorithme.
2. Pourquoi l'algorithme se termine-t-il ?

Dans cet algorithme, les principaux éléments de complexité sont les tests et les échanges.

On note *n* la longueur de la liste à trier et $n(b)$, $n(w)$ et $n(c)$ les nombres d'éléments respectivement bleus, blancs et rouges dans la liste. On a bien sûr : $n = n(b) + n(w) + n(r)$.

On note également $t(n)$ et $e(n)$ le nombre de tests et d'échanges effectués pour trier une telle liste.

3. Evaluer $t(n)$ et $e(n)$ dans le meilleur et le pire des cas (on donnera à chaque fois un exemple correspondant à la situation).

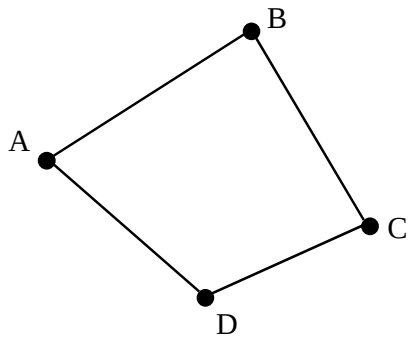
4. Evaluer la complexité moyenne en termes de tests puis en termes d'échanges.

   Dans un premier temps, on pourra considérer que l'on a : $n(b)=n(w)=n(r)=\frac{n}{3}$.

   Dans un second temps, on considèrera que l'on a : $n=N(b)+N(w)+N(r)$ où $N(b)$, $N(w)$ et $N(r)$ sont des variables aléatoires d'espérances égales à $\frac{n}{3}$.

EXERCICE 3. Des quadrilatères dans des graphes

Dans cet exercice, on cherche à déterminer le nombre de « quadrilatères vrais » présents dans un graphe simple non orienté, c'est-à-dire le nombre de configurations du type :



où A, B, C et D sont quatre sommets du graphe, deux à deux distincts.

Notons, classiquement :

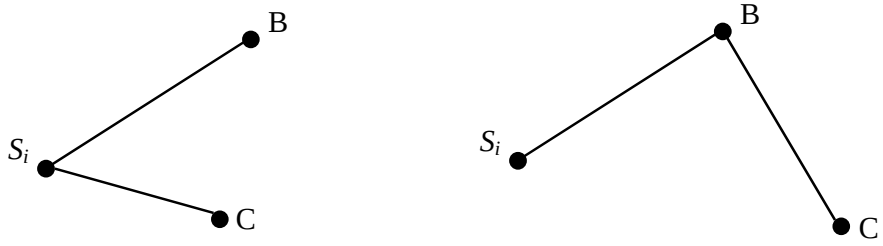
- n l'ordre du graphe considéré.
- Pour tout entier naturel i dans $[\![0\,;n-1]\!]$, d_i le degré du sommet S_i.
- D la somme des degrés des sommets : $D=\sum_{i=0}^{n-1} d_i$.
- $A=\left(a_{ij}^{(0)}\right)$ sa matrice d'adjacence.
- Pour tout entier naturel p, $A^p=\left(a_{ij}^{(p)}\right)$ la puissance p-ième de la matrice A.

On note alors $Q(n)$ le nombre de « quadrilatères vrais » d'un tel graphe.

Pour évaluer $Q(n)$, nous allons nous intéresser, légitimement (☺), à $\mathrm{Tr}\left(A^4\right)=\sum_{i=0}^{n-1} a_{ii}^{(4)}$, la trace de la matrice A^4.

1. Expliquer pourquoi la contribution de $Q(n)$ à $\mathrm{Tr}\left(A^4\right)$ est égale à $8\times Q(n)$.

2. Les autres configurations contribuant à $\mathrm{Tr}\left(A^4\right)$ sont :



Avec les chaînes :

- Configuration de gauche :
  S_i-B-S_i-C-S_i et S_i-C-S_i-B-S_i, sachant que dans ce type de configuration, on peut avoir B = C.
- Configuration de droite :
  S_i-B-C-B-S_i, sachant que dans ce type de configurations, on a $\mathrm{B}\neq\mathrm{C}$.

a. Montrer que la contribution des configurations de gauche à $\mathrm{Tr}\left(A^4\right)$ vaut :

$$\sum_{i=1}^{n} d_i^2$$

b. Montrer que la contribution des configurations de droite à $\mathrm{Tr}\left(A^4\right)$ vaut :

$$\sum_{\substack{i,j\\ i\neq j}} a_{ij}^{(2)}$$

3. Déduire des questions précédentes :

$$\boxed{Q(n)=\frac{1}{8}\times\left(\mathrm{Tr}\left(A^4\right)-\sum_{i=1}^{n} d_i^2-\sum_{\substack{i,j\\ i\neq j}} a_{ij}^{(2)}\right)=\frac{1}{8}\times\left(\mathrm{Tr}\left(A^4\right)-\sum_{i=1}^{n} d_i^2-\sum_{i,j} a_{ij}^{(2)}+D\right)}$$

4. Ecrire une fonction Python `NumOfQ` qui reçoit en argument la matrice d'adjacence A d'un graphe simple non orienté et en renvoie le nombre de « quadrilatères vrais ».

   On rappelle que :

   - La fonction `dot` de la bibliothèque `numpy` permet de calculer le produit de deux matrices (tableaux `numpy`).
   - La fonction `shape` (équivalente à la méthode `shape`) de la bibliothèque `numpy` permet d'obtenir les dimensions d'une matrice (tableaux `numpy`). Cette fonction renvoie un tuple.