Le sujet comporte un total de 3 exercices indépendants
qui peuvent être traités dans l'ordre de votre choix.

Une importance particulière devra être apportée à la qualité de la rédaction
Et on s'efforcera, dans la limite du raisonnable, de commenter les codes fournis.

Exercice N°1 – Une simulation

Soit L une liste Python initialement vide.

On considère la simulation suivante : on tire au hasard uniformément et successivement *n* entiers naturels dans l'intervalle $[\![1;2p]\!]$ où *p* est un entier naturel non nul :

- Si l'entier obtenu est pair, on l'ajoute à la fin de la liste L.
- Si l'entier obtenu est impair, on vide la liste L.

On effectue *N* fois la simulation précédente.

Ecrire une fonction Python qui recevra en argument les trois entiers *p*, *n* et *N* effectuera *N* fois la simulation décrite ci-dessus et renverra :

- La longueur moyenne de la liste L.
- La valeur moyenne de la somme des éléments de L.

On utilisera la fonction Python `randint` du module `random` : l'appel `randint(a,b)`, où `a` et `b` sont deux entiers, renvoie un entier uniformément choisi dans l'intervalle $[\![a;b]\!]$.

Exercice N°2 – Exponentiation : une autre approche

On cherche à calculer efficacement a^n où *a* est un réel et *n* un entier naturel.

Comme la division d'un entier par une puissance de 2 est rapide et que la méthode de l'exponentiation rapide est très efficace lorsque l'exposant est une puissance de 2, on adopte ici la démarche suivante : on écrit *n* comme somme de *p* puissances de 2 :

$$n = \sum_{i=1}^{p} 2^{n_i}$$

où $i < j \Leftrightarrow n_i > n_j$

Cette décomposition est unique.
Par exemple : $93 = 64+16+8+4+1 = 2^6+2^4+2^3+2^2+2^0$ ($n_1 = 6$, $n_2 = 4$, etc.).

Ainsi, on a : $a^n = a^{\sum_{i=1}^{p} 2^{n_i}} = \prod_{i=1}^{p} a^{2^{n_i}}$.

Pour le calcul de $a^{2^{n_i}}$, on suppose que l'on dispose d'une fonction Python `QuickExpo` qui reçoit en argument un flottant `x` et un entier naturel `n` et renvoie le flottant x^n. Ainsi, pour le calcul de a^{93}, on calculera en fait $a^{64+16+8+4+1} = a^{2^6+2^4+2^3+2^2+2^0} = a^{2^6} \times a^{2^4} \times a^{2^3} \times a^{2^2} \times a^{2^0}$, chaque facteur du dernier produit étant calculé grâce à la méthode de l'exponentiation rapide (fonction `QuickExpo`).

1. Ecrire une fonction <u>non récursive</u> `NewExpo_nr` qui reçoit en argument un flottant `a` et un entier naturel `n` et renvoie un flottant correspondant au réel a^n.
2. Ecrire une fonction <u>récursive</u> `NewExpo_r` qui reçoit en argument un flottant `a` et un entier naturel `n` et renvoie un flottant correspondant au réel a^n.

On rappelle que la fonction log du module `math` permet de calculer le logarithme d'un réel quelconque dans une base $a > 0$. Ainsi, le logarithme en base 2 de *n* sera obtenu via l'appel : `log(n,2)`.

Exercice N°3 – Matrices triangulaires

Un élève de CPGE s'intéresse aux matrices triangulaires réelles (franchement, il a bien raison ! ☺).

Programmant en langage Python mais ne disposant pas de la bibliothèque `numpy` sur son ordinateur (que n'utilise-t-il Pyzo ??? Version 2015 bien sûr… ☺), il décide de représenter une telle matrice sous forme d'une liste (celles des lignes de la matrice considérée) mais en ne stockant pas les « 0 » situés au-dessus de la diagonale (pour les triangulaires inférieures) ou sous la diagonale (pour les triangulaires supérieures).
Ainsi, il représente les matrices :

$$A = \begin{pmatrix} 2 & 0 & 0 \\ -1 & 1 & 0 \\ 3 & -5 & 13 \end{pmatrix} \text{ et } B = \begin{pmatrix} 4 & -7 & 20 & 0 \\ 0 & 0 & -3 & 5 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 10 \end{pmatrix}$$

par les listes :

```
[[2],[-1,1],[3,-5,13]]
```

```
[[4,-7,20,0],[0,-3,5],[2,1],[10]]
```

Dans la suite, toutes les matrices triangulaires seront représentées de la sorte et on suppose que l'on dispose d'une classe Python `TriMat` où l'unique attribut d'un objet est noté `content` et correspond à une liste de l'un des deux types ci-dessus. Par exemple, la première matrice ci-dessus pourra correspondre à l'objet `LA` créé comme suit :

```
LA=TriMat([[2],[-1,1],[3,-5,13]])
```

et on aura évidemment :

```
LA.content=[[2],[-1,1],[3,-5,13]]
```

1. Ecrire une méthode Python `TriMat_test` qui reçoit en argument un objet de type `TriMat` et renvoie la chaîne de caractères `'Tinf'` ou `'Tsup'` selon que l'objet correspond à une matrice triangulaire inférieure ou supérieure.

2. Ecrire une méthode Python `TriMat_det` qui reçoit en argument un objet de type `TriMat` et renvoie le déterminant de la matrice triangulaire correspondante.

3. Ecrire une méthode Python `TriMat_transpose` qui reçoit en argument un objet de type `TriMat` et renvoie un nouvel objet de type `TriMat` correspondant à la transposée de la matrice de l'objet passé en argument.

4. Ecrire une fonction Python `TriMat_inverse` qui reçoit en argument un objet de type `TriMat` et :
   - teste l'inversibilité de la matrice T correspondant à cet objet.
   - renvoie le booléen `True` et un nouvel objet de type TriMat correspondant à la matrice T^{-1} ou seulement le booléen `False` suivant que T est ou non inversible.

   Lorsque T est inversible, évaluer la complexité (nombre d'opérations) du calcul de T^{-1}.

5. En vous appuyant sur quelques arguments mathématiques simples, proposer un code pour la surcharge de l'opérateur + (méthode `__add__` uniquement…).

6. Même question que précédemment avec l'opérateur * (méthodes `__mul__` et `__rmul__`).