Toutes calculatrices autorisées.
Le sujet comporte un total de 3 exercices.

EXERCICE 1. Décollage d'un engin spatial.

On s'intéresse dans cet exercice au décollage d'un engin spatial depuis le sol terrestre.

On modélise la situation comme suit :

- La trajectoire de l'engin est supposée verticale et il est repéré (variable *z*) le long d'un axe vertical passant par le centre de la terre, pris comme origine ($z = 0$), et le centre du pas de tir ($z = R_T > 0$ où R_T désigne le rayon de la Terre assimilée à une sphère).
- L'origine des temps est l'instant du décollage. On a donc : $z(0) = R_T$ et $\left(\frac{dz}{dt}\right)(0) = 0$.
- La masse de l'engin est notée *m* et est une fonction du temps.
- La vitesse d'éjection des gaz par rapport à l'engin est supposée constante et notée *u* ($u > 0$).
- On néglige les frottements.

Avec les hypothèses précédentes en considérant le système fermé {fusée, gaz} entre les instants *t* et $t + dt$, on effectue un bilan de la variation de la quantité de mouvement globale et le principe fondamental de la dynamique nous donne :

$$m\frac{d^2z}{dt^2} + \frac{dm}{dt}u = -G\frac{mM_T}{z^2} \qquad (\text{E})$$

où *G* désigne la constante universelle de la gravitation et M_T la masse de la Terre.

1. On s'intéresse en fait à la variable *h* correspondant à l'altitude de l'engin (i.e. sa distance au sol). Vérifier que la variable *h* satisfait l'équation différentielle (E') suivante et préciser les conditions initiales (à $t = 0$) :

$$\frac{d^2h}{dt^2} = -\frac{GM_T}{(R_T + h)^2} - \frac{1}{m}\frac{dm}{dt}u \qquad (\text{E}')$$

2. On cherche à résoudre l'équation différentielle (E') avec les conditions initiales précisées à la question précédente à l'aide d'un schéma d'Euler explicite sur un intervalle de temps $[0\,;t_f]$. On utilise un pas de temps constant Δt. Pour la masse m et la dérivée $\frac{dm}{dt}$, on dispose d'une fonction Python `FM` qui reçoit comme argument un flottant correspondant au temps t et renvoie un tuple de deux éléments correspondant respectivement à $m(t)$ et $\left(\frac{dm}{dt}\right)(t)$. Les variables `G`, `MT`, `RT` et `u` correspondant respectivement à la constante G, à la masse M_T, au rayon R_T et à la vitesse d'éjection u seront déclarées globales.

3. Ecrire une fonction Python `D2H` qui reçoit comme arguments une variable `t` correspondant à un instant t, une variable `h` correspondant à l'altitude de l'engin à l'instant t et une variable `v` correspondant à sa vitesse (dérivée de h par rapport au temps) à ce même instant. La fonction `D2H` renverra la dérivée seconde de h par rapport au temps à l'instant t.

```
def D2H(t,h,v):
    ...
```

4. Ecrire une fonction `EULER_decollage` qui permet de résoudre numériquement l'équation différentielle (E') sur un intervalle de temps $[t_i\,;t_f]$ ($t_i < t_f$). Elle reçoit comme arguments :

- Une variable `ti` correspondant au temps initial t_i.
- Une variable `tf` correspondant au temps final t_f.
- Une variable `pas` correspondant au pas de temps Δt.
- Une variable `h0` correspondant à l'altitude initiale.
- Une variable `v0` correspondant à la vitesse initiale.

La fonction construira trois listes `Lt`, `Lh` et `Lv` contenant les valeurs des instants, de l'altitude et de la vitesse de l'engin aux instants 0, *pas*, $2\times pas$, $3\times pas$, …
La fonction affichera un graphique donnant l'altitude de l'engin en fonction du temps (le script contient l'instruction `import matplotlib.pyplot as plt`).

```
def EULER_décollage(ti,tf,pas,h0,v0):
    ...
```

Par exemple, si l'on prend la seconde comme unité de temps et si l'on souhaite obtenir l'altitude atteinte par l'engin au bout de 2 minutes avec un pas de temps égal à une seconde, on appellera la fonction `EULER_decollage` dans le script comme suit :

```
EULER_décollage(0,120,1,0,0)
```

5. Le décollage se divise en fait en quatre phases principales correspondant aux intervalles de temps $I_1 = [0\,;t_1]$, $I_2 = [t_1\,;t_2]$, $I_3 = [t_2\,;t_3]$ et $I_4 = [t_3\,;t_4]$. On dispose ainsi de quatre fonctions Python `FM1`, `FM2`, `FM3` et `FM4` donnant à un instant *t* quelconque, dans l'un de ces intervalles, la masse $m(t)$ et la valeur prise par sa dérivée $\left(\frac{dm}{dt}\right)(t)$. On suppose par ailleurs que les temps t_1, t_2, t_3 et t_4 sont fournis dans une liste `LT`.

   Comment modifier le script et la fonction `EULER_decollage` pour tenir compte de ces quatre phases et résoudre l'équation différentielle (E') sur l'intervalle de temps $[0\,;t_4]$?

EXERCICE 2. Les courbes de BEZIER

Une courbe de Bézier à $N+1$ ($N \geq 1$) points de contrôle $P_0, P_1, P_2, ..., P_N$ est une courbe paramétrée d'extrémités les points P_0 et P_N définie par :

$$\left\{ P(t) = \sum_{i=0}^{N} B_i^N(t).P_i,\ t \in [0\,;1] \right\}$$

où les B_i^N sont les polynômes de Bernstein définis par :

$$\forall t \in \mathbb{R},\ B_i^N(t) = \binom{N}{i} \times t^i \times (1-t)^{N-i}$$

<u>Remarque importante</u> : l'écriture B_i^N ne désigne en rien une puissance !

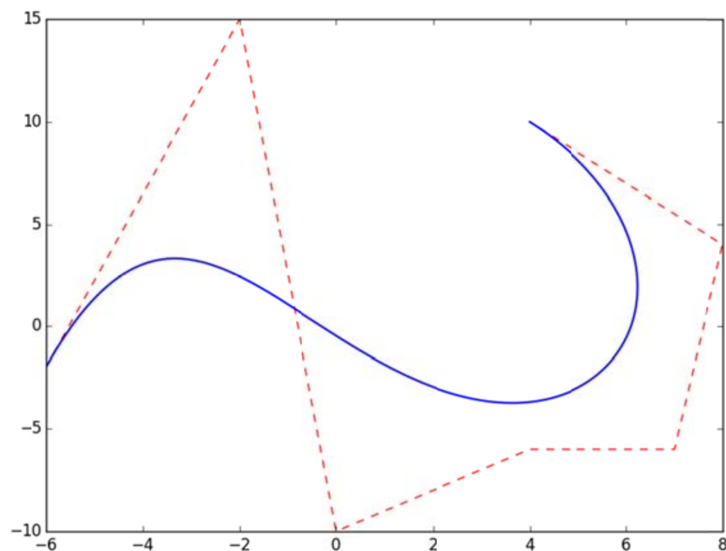
Par exemple, pour $N = 6$ et en considérant les points de contrôle suivants :

$$P_0\begin{pmatrix}-6\\-2\end{pmatrix}, P_1\begin{pmatrix}-2\\15\end{pmatrix}, P_2\begin{pmatrix}0\\-10\end{pmatrix}, P_3\begin{pmatrix}4\\-6\end{pmatrix}, P_4\begin{pmatrix}7\\-6\end{pmatrix}, P_5\begin{pmatrix}8\\4\end{pmatrix} \text{ et } P_6\begin{pmatrix}4\\10\end{pmatrix}$$

on obtient la courbe de Bézier correspondant aux points $P(t)\begin{pmatrix}x(t)\\y(t)\end{pmatrix}$ ($t \in [0\,;1]$) avec :

$$\begin{cases} x(t) = \sum_{i=0}^{6} B_i^N(t) \times x_i = -6B_0^N(t) - 2B_1^N(t) + 4B_3^N(t) + 7B_4^N(t) + 8B_5^N(t) + 4B_6^N(t) \\ y(t) = \sum_{i=0}^{6} B_i^N(t) \times y_i = -2B_0^N(t) + 15B_1^N(t) - 10B_2^N(t) - 6B_3^N(t) - 6B_4^N(t) + 4B_5^N(t) + 10B_6^N(t) \end{cases}$$

Graphiquement (la ligne en pointillés rouges est la ligne polygonale des points de contrôle) :



Partie A : évaluation directe des polynômes de Bernstein

Dans cette partie, on se donne un réel t dans l'intervalle $[0;1]$ et un entier naturel i dans $[\![0;N]\!]$. On cherche à évaluer le nombre $C(N,i)$ de multiplications et de divisions requises par le calcul de $B_i^N(t)=\binom{N}{i}\times t^i\times(1-t)^{N-i}$.

1. On a : $B_i^N(t)=\dfrac{N!}{i!\times(N-i)!}\times t^i\times(1-t)^{N-i}$.

   Dans cette question, on suppose que l'on calcule $B_i^N(t)$ en calculant chacune des trois factorielles apparaissant dans le coefficient binomial.

   Calculer $C(N,i)$.

2. On a, pour $i\neq 0$:

$$B_i^N(t)=\frac{N\times(N-1)\times(N-2)\times\ldots\times(N-i+1)}{i\times(i-1)\times(i-2)\times\ldots\times2\times1}\times t^i\times(1-t)^{N-i}=\frac{\prod_{k=0}^{i-1}(N-k)}{\prod_{k=0}^{i-1}(k+1)}\times t^i\times(1-t)^{N-i}$$

   On a donc calculé $\binom{N}{i}$ en ayant simplifié le rapport $\dfrac{N!}{(N-i)!}$.

   a. Calculer $C(N,i)$.

On a aussi $\binom{N}{i} = \binom{N}{N-i}$.

b. Calculer $C(N, N-i)$.

c. Ecrire une fonction Python `Bernstein_base` recevant en argument deux entiers N et i et un réel t dans $[0\,;1]$ et renvoyant $B_i^N(t)$ en effectuant un minimum de multiplications.

3. On a, pour i non nul : $\binom{N}{i} = \frac{N}{i} \times \binom{N-1}{i-1}$.

   On propose le code récursif suivant pour le calcul de $\binom{N}{i}$:

```
def binomial(n,i):
     if i == 0 or i == n:
          return 1
     else:
          return((n / i) * binomial(n - 1,i - 1))
```

   Ce code peut conduire à des résultats erronés. Pouvez dire pourquoi ?
   En ne modifiant que le deuxième `return`, proposer un code correct.

Partie B : évaluation récursive des polynômes de Bernstein

Les polynômes de Bernstein peuvent être calculés selon la définition récursive suivante :
$B_0^1(t) = 1 - t$, $B_1^1(t) = t$ et pour $N > 1$:

$$B_i^N(t) = \begin{cases} (1-t) \times B_i^{N-1}(t) & \text{si } i = 0 \\ (1-t) \times B_i^{N-1}(t) + t \times B_{i-1}^{N-1}(t) & \text{si } i \in [\![1\,; N-1]\!] \\ t \times B_{i-1}^{N-1}(t) & \text{si } i = m \end{cases}$$

4. Ecrire une fonction récursive `Bernstein_rec` qui calculera $B_i^N(t)$ et recevra en argument deux entiers N et i (dans $[\![0\,; N]\!]$) et un réel t (dans $[0\,;1]$). Les appartenances mentionnées ne seront pas testées par la fonction.

Partie C : l'algorithme de Casteljau

L'algorithme de Casteljau est un algorithme visant à déterminer, pour *t* fixé dans $[0\,;1]$, les coordonnées $\begin{pmatrix} x(t) \\ y(t) \end{pmatrix}$ d'un point $P(t)$ en tirant parti de la définition récursive précédente. Le principe en est le suivant :

- On note $P_0^0, P_1^0, P_2^0, ..., P_N^0$ la liste initiale des points de contrôle. Cette liste comporte $N+1$ points.
- A partir de la liste précédente, on construit la liste des *N* barycentres $P_i^1 = \left\{\left(P_i^0, t\right), \left(P_{i+1}^0, 1-t\right)\right\}$ avec $i \in [\![0\,; N-1]\!]$. On a ainsi : $P_i^1 = t.P_i^0 + (1-t).P_{i+1}^0$, c'est-à-dire, en notant $P_i^1 \begin{pmatrix} x_i^1(t) \\ y_i^1(t) \end{pmatrix}$: $\begin{cases} x_i^1(t) = t \times x_i^0(t) + (1-t) \times x_{i+1}^0(t) \\ y_i^1(t) = t \times y_i^0(t) + (1-t) \times y_{i+1}^0(t) \end{cases}$.
- On recommence l'étape précédente jusqu'à obtenir le point P_0^N qui correspond au point $P(t)$ de la courbe de Bézier ayant les points $P_0^0, P_1^0, P_2^0, ..., P_N^0$ pour points de contrôle.

5. Pourquoi l'algorithme précédent se termine-t-il ?
6. Ecrire une fonction récursive `PointBezier_rec` qui renverra le point $P(t) = P_0^N$ et recevra en argument une liste de points et un réel *t* (dans $[0\,;1]$). Un point sera représenté par une liste de deux éléments correspondant à ses coordonnées. De fait, une liste de points sera une liste de listes. Avec l'exemple fourni en début d'exercice, on appellera initialement la fonction avec la liste *PC* :

```
[[-6,-2],[-2,15],[0,-10],[4,-6],[7,-6],[8,4],[4,10]]
```

7. En supposant disponible la fonction `PointBezier_rec`, écrire une fonction `Bezier_Casteljau` qui recevra comme arguments une liste *PC* de points et un entier *NP* et tracera *NP* points (correspondant à $NP+1$ valeurs équiréparties de *t* dans $[0\,;1]$) de la courbe de Bézier ayant pour points de contrôle les points de *PC*.

EXERCICE 3. Le tri de SHELL.

En 1959, Donald SHELL a proposé une amélioration du tri par insertion. Le principe général en est le suivant.

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, $2g$, $3g$, …
- 1, $g+1$, $2g+1$, $3g+1$, …

- $2,\ g+2,\ 2g+2,\ 3g+2, \dots$
- …
- $g-1,\ 2g-1,\ 3g-1,\ 4g-1, \dots$

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
```

1. 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 : $E(701\times 2,3)=1612$ puis $E(1612\times 2,3)=3707$.

2. 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.

3. 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.