TD n°1 et 2 : EXERCICES DE RÉVISION

Les exercices ci-dessous n'ont aucune prétention ; il s'agit seulement de révisions du programme de 1re année sur les instructions de base du langage Python.

I. Exercice 1

Écrire une fonction `binom(n,k)` permettant de calculer le coefficient binomial $\binom{n}{k}$ avec $n \in \mathbb{N}$ et $k \in \mathbb{Z}$ en utilisant exclusivement les propriétés suivantes :

1. Si $k < 0$ ou $k > n$, alors $\binom{n}{k} = 0$.
2. $\forall n \in \mathbb{N}$, $\binom{n}{0} = 1$.
3. $\forall k, n \in (\mathbb{N}^*)^2$, $\binom{n}{k} = \dfrac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1}$.

II. Exercice 2

1. On souhaite construire le triangle de Pascal qui peut s'écrire de la façon suivante

$$\begin{array}{cccc} \binom{0}{0} & & & \\ \binom{1}{0} & \binom{1}{1} & & \\ \binom{2}{0} & \binom{2}{1} & \binom{2}{2} & \\ \binom{3}{0} & \binom{3}{1} & \binom{3}{2} & \binom{3}{3} \end{array} \quad \text{ou encore} \quad \begin{array}{cccc} 1 & & & \\ 1 & 1 & & \\ 1 & 2 & 1 & \\ 1 & 3 & 3 & 1 \end{array}$$

   Écrire une procédure `pascal(n)` qui construit ce triangle (sous forme d'une liste de listes de longueurs variables) en utilisant la fonction de l'exercice 1.
   L'exemple précédent est donc le résultat de `pascal(3)`, qui devrait renvoyer la liste `[[1],[1,1],[1,2,1],[1,3,3,1]]`.

2. Il y a néanmoins une manière plus facile de calculer les termes successifs du triangle de Pascal en utilisant la formule éponyme :

$$\binom{n}{p} = \binom{n}{p-1} + \binom{n-1}{p-1}.$$

   En utilisant la formule précédente, implémentez une fonction `pascal2(n)` qui renvoie le triangle de Pascal sans utiliser d'appel à la fonction `binom(n,p)`. Comparez la vitesse d'exécution des deux fontions `pascal(n)` et `pascal2(n)` pour $n = 50, 100, 200, 300, 400$. Représentez le résultat obtenu sous forme d'un graphique.

3. À présent que l'on sait construire les éléments du triangle de Pascal facilement, on peut résoudre le problème 203 du projet Euler. Les 8 premières lignes ($n = 7$) du triangle de Pascal s'écrivent

$$\begin{array}{ccccccccc} 1 & & & & & & & \\ 1 & 1 & & & & & & \\ 1 & 2 & 1 & & & & & \\ 1 & 3 & 3 & 1 & & & & \\ 1 & 4 & 6 & 4 & 1 & & & \\ 1 & 5 & 10 & 10 & 5 & 1 & & \\ 1 & 6 & 15 & 20 & 15 & 6 & 1 & \\ 1 & 7 & 21 & 35 & 35 & 21 & 7 & 1 \end{array}$$

   It can be seen that the first eight rows of Pascal's triangle contain twelve distinct numbers : 1, 2, 3, 4, 5, 6, 7, 10, 15, 20, 21 and 35.
   A positive integer n is called squarefree if no square of a prime divides n. Of the twelve distinct numbers in the first eight rows of Pascal's triangle, all except 4 and 20 are squarefree. The sum of the distinct squarefree numbers in the first eight rows is 105.
   Find the sum of the distinct squarefree numbers in the first 51 rows of Pascal's triangle.

   Plus précisément, écrivez un programme `squarefree_pascal(n)` qui calcule la somme des nombres « squarefree » distincts présents dans les n premières lignes du triangle de Pascal.

III. Exercice 3

Considérons le triangle suivant :

3
7 4
2 **4** 6
8 5 **9** 3

En partant du sommet et en descendant uniquement selon les chiffres adjacents sur la ligne du dessous, la somme maximale que l'on peut obtenir est $3+7+4+9=23$. Écrivez une fonction `somme_maximale(triangle)` qui, étant donné un triangle proposé comme une liste de liste comme ceux de la section précédente, calcule la somme maximale sur un des chemins qui mène du sommet à la base.

Une petite mise en garde néanmoins : pour un triangle de N lignes, il existe 2^N-1 chemins différents du sommet à la base. S'il est plausible de tous les explorer par une méthode « force brute » pour de faibles valeurs de N (disons jusqu'à 20 environ), cela n'est plus possible pour un triangle de 100 lignes. La méthode se doit d'être un peu plus réfléchie.

IV. Exercice 4

Soit f continue strictement monotone sur un intervalle $[a,b]$, telle que $f(a)f(b)<0$. On sait alors que f s'annule une fois et une seule sur $]a,b[$.

Écrire une fonction `dicho(f,a,b,eps)` permettant d'encadrer le zéro de f à `eps` près.

Pour tester, on pourra chercher le point fixe de la fonction cos sur $\left[0\,;\frac{\pi}{2}\right]$, en considérant la fonction $f : x \mapsto x-\cos x$.

Comparer le résultat obtenu avec celui que l'on obtient en considérant la suite définie par récurrence par $u_{n+1}=f(u_n)$.

V. Exercice 5

1. Écrire une fonction qui renvoie, sous forme de liste, les chiffres d'un entier naturel n (on n'utilisera pas les fonctions sur les chaînes de caractères).
2. Écrire une fonction qui renvoie la somme des carrés des chiffres d'un entier n.
3. Un entier est un *nombre heureux* si, lorsque l'on calcule la somme des carrés de ses chiffres puis la somme des carrés des chiffres du nombre obtenu et ainsi de suite, on aboutit au nombre 1.
   Écrire une fonction qui teste si un nombre est heureux. On utilisera pour cela le fait que, si l'on applique un tel processus à partir d'un entier quelconque, on finit toujours par obtenir soit $\{1\}$, soit un nombre de l'ensemble $\{4,16,37,58,89,145,42,20\}$ (qui est alors malheureux).
4. Afficher la liste de tous les nombres heureux inférieurs à 100.
5. Afficher la liste de tous les couples heureux $(n,n+1)$ et de tous les triplets heureux $(n,n+,n+2)$ avec $n\leqslant 10000$. Généraliser.

Pour tester vos résultats, on donne :

- la liste des nombres heureux entre 1 et 100 :

$$\{1,7,10,13,19,23,28,31,32,44,49,68,70,79,82,86,91,94,97\}$$

- les premiers couples heureux :

$$(31,32),\ (129,130),\ (192,193),\ (262,263),\ldots$$

- les premiers triplets heureux :

$$(1880,1881,1882),\ (4780,4781,4782),\ (4870,4871,4872),\ldots$$

- les premiers quadruplets heureux :

$$(7839,7840,7841,7842),\ (8769,8740,8741,8742),\ (11248,11249,11250,11251),\ldots$$

VI. Exercice 6

Un nombre entier est dit *tricolore* si son carré s'écrit uniquement avec les chiffres 1, 4 et 9.

1. Écrire une fonction `tricolore(n)` permettant de vérifier si un entier n est tricolore.
2. Écrire une fonction `liste_tricolores(N)` qui donne la liste de tous les entiers tricolores $\leqslant N$.

VII. Exercice 7

Écrire un programme qui permet d'afficher la liste des nombres premiers inférieurs à un entier N donné, en utilisant la méthode du crible d'Ératosthène.

Cette méthode consiste à considérer la table formée des entiers de 2 à N ; dans cette table, on raye déjà les multiples de 2, puis à chaque fois on raye les multiples du plus petit entier restant.

On peut s'arrêter lorsque le carré du plus petit entier restant est supérieur au plus grand entier restant puisque tout nombre non premier admet forcément un diviseur inférieur ou égal à sa racine.

VIII. Exercice 8

Soit f la fonction définie par :

$$f : n \longmapsto \begin{cases} n/2 & \text{si } n \text{ est pair} \\ 3n+1 & \text{si } n \text{ est impair} \end{cases}$$

On s'intéresse à la suite définie par récurrence par :

$$u_0 \in \mathbb{N} \quad \text{et} \quad u_{n+1} = f(u_n).$$

Par exemple, avec $u_0 = 13$, on obtient

$$13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, 4, 2, 1, \ldots$$

C'est ce qu'on appelle la suite de Collatz (ou suite de Syracuse) du nombre 13. Après avoir atteint le nombre 1, la suite de valeurs (1,4,2,1,4,2,1,4,2...) se répète indéfiniment en un cycle de longueur 3 appelé cycle trivial.

La conjecture de Collatz est l'hypothèse selon laquelle la suite de Syracuse de n'importe quel entier strictement positif atteint 1. Bien qu'elle ait été vérifiée pour les 5,7 premiers milliards de milliards d'entiers et en dépit de la simplicité de son énoncé, cette conjecture défie depuis de nombreuses années les mathématiciens. Paul Erdös a dit à son propos que « les mathématiques ne sont pas encore prêtes pour de tels problèmes... »

1. Implémenter la fonction `f(n)` ci-dessus.

2. Soit $u_0 = a \in \mathbb{N}^*$. On appelle orbite de a la liste des termes de la suite $u_{n+1} = f(u_n)$ jusqu'à ce que l'on tombe sur 1 (compris). Écrire le programme `orbite(a)` qui prend comme entrée l'entier `a` et qui renvoie son orbite sous forme d'une liste.

Plusieurs caractéristiques d'une orbite peuvent être explorées :

- son « temps de vol » correspond au nombre total d'entiers visités sur l'orbite.
- son « altitude » est donnée par le plus grand entier visité sur l'orbite.
- son « temps de vol en altitude » correspond au nombre d'étapes avant de passer strictement en dessous du nombre de départ.
- et enfin son « temps de vol avant la chute » correspond au nombre d'étapes minimum après lequel on ne repasse plus au-dessus de la valeur de départ.

Par exemple pour l'orbite du nombre 13, l'altitude vaut 40 (maximum de la suite), le temps de vol vaut 10, le temps de vol en altitude vaut 3 (on passe à $10 < 13$ lors de la 3^e étape) et le temps de vol avant la chute vaut 6 (on passe à $8 < 13$ lors de la 6^e itération de la fonction `f`).

3. Écrire une fonction `temps_de_vol(a)` qui renvoie le temps de vol correspondant à l'orbite de `a`.

4. Écrire une fonction `altitude(a)` qui renvoie l'altitude de l'orbite de `a`. Pour s'entraîner, on implémentera la recherche du maximum « à la main ».

5. Écrire une fonction `temps_en_altitude(a)` qui renvoie le temps de vol en altitude correspondant à l'orbite de `a`.

6. Écrire une fonction `temps_avant_chute(a)` qui renvoie le temps de vol avant la chute pour l'orbite de `a`.

Nous allons utiliser ces procédures pour résoudre les questions suivantes.

7. Pour des entiers de départ de valeur strictement inférieures à un million, déterminer à chaque fois

a) celui qui monte le plus haut en altitude et la valeur de cette altitude maximale (à stocker dans la liste `le_plus_haut` qui aura donc deux éléments `[a,altitude(a)]`).

b) celui dont le temps de vol est le plus long et la valeur de ce temps de vol (à stocker dans la liste `le_plus_long`).

c) celui dont le temps de vol en altitude est le plus long et la valeur de ce temps de vol (à stocker dans la liste `le_plus_long_en_altitude`).

d) celui dont le temps de vol avant la chute est le plus long et la valeur de ce temps de vol (à stocker dans la liste `le_plus_long_avant_la_chute`).

8. Ne trouvez-vous pas que la procédure suggérée fasse un peu « gâchis » ? Implémentez une procédure unifiée qui puisse effectuer tous les calculs précédents en moins de temps. Estimez le gain de temps que l'on peut espérer et mesurez-le (via un `import time` et à l'aide de `time.clock()`, plus d'info avec `help(time.clock)`). N'oubliez pas de tester votre procédure en retrouvant les résultats précédent

IX. Exercice 9

Cet exercice concerne le procédé d'orthonormalisation de Schmidt, dont je rappelle brièvement le principe.

Dans un espace préhilbertien réel, il s'agit, à partir d'une famille libre de vecteurs $(v_1,\dots,v_n)$, de construire une famille orthogonale $(e_1,\dots,e_n)$ qui engendre les mêmes sous-espaces vectoriels successifs, c'est-à-dire telle que la matrice de passage d'une base à l'autre soit triangulaire supérieure.

L'étape générale de l'algorithme consiste à soustraire au vecteur v_{k+1} sa projection orthogonale sur le sous-espace vectoriel $\text{Vect}(v_1,\dots,v_k)=\text{Vect}(e_1,\dots,e_k)$, puis à le normer. Il peut donc s'écrire :

$$\left|\begin{array}{ll}\textbf{Initialisation :} & \text{On pose } u_1=v_1 \text{ puis } e_1=\dfrac{u_1}{\|u_1\|}\\[2ex] \textbf{k-ième étape :} & \text{On pose } u_k=v_k-\displaystyle\sum_{j=1}^{k-1}\langle e_j\,|\,v_k\rangle\,e_j \text{ puis } e_k=\dfrac{u_k}{\|u_k\|}\cdot\end{array}\right.$$

A. Orthonormalisation de vecteurs dans $\mathbb{R}^m$.

1. Implémenter ce procéder en créant une fonction `gramschmidt` qui prend en entrée une liste de n vecteurs de $\mathbb{R}^m$, chaque vecteur étant lui-même une liste de longueur m, puis qui renvoie la famille orthonormale obtenue. On veillera à afficher un message d'erreur si la famille n'est pas libre ; cela correspond, dans l'algorithme ci-dessus, au cas où l'un des u_k est le vecteur nul, c'est-à-dire, en Python, si $\|u_k\|<10^{-10}$ (par exemple).

2. La méthode précédente est instable numériquement à cause des erreurs d'arrondi qui font que, dans certains cas, les vecteurs u_j ne sont pas vraiment orthogonaux. Pour l'illustrer, considérer la famille de vecteurs de $\mathbb{R}^4$:

$$v_1=(1,\varepsilon,0,0)\quad,\quad v_2=(1,0,\varepsilon,0)\quad,\quad v_2=(1,0,0,\varepsilon)$$

avec $\varepsilon=10^{-10}$ (par exemple). Que constatez-vous ?

L'algorithme de Gram-Schmidt modifié consiste à calculer chaque u_k pour $k\geqslant 2$ à l'aide des formules suivantes (je vous laisse le soin de vérifier que, mathématiquement, cela donne la même chose) :

$$\begin{array}{lcl} u_k^{(0)} &=& v_k\\ u_k^{(1)} &=& v_k-p_{u_1}(v_k)\,,\\ u_k^{(2)} &=& u_k^{(1)}-p_{u_2}(u_k^{(1)})\,,\\ &\vdots&\\ u_k^{(k-2)} &=& u_k^{(k-3)}-p_{u_{k-2}}(u_k^{(k-3)})\\ u_k &=& u_k^{(k-2)}-p_{u_{k-1}}(u_k^{(k-2)})\end{array}$$

où p_u désigne le projeté orthogonal sur la droite de base u, c'est-à-dire $p_u(v)=\dfrac{\langle u\,|\,v\rangle}{\|u\|^2}\,u$.

Implémenter cet algorithme et comparer avec le résultat précédent.

B. Orthonormalisation de polynômes.

Un polynôme $\displaystyle\sum_{k=1}^{n}a_kX^k$ sera représenté en Python par une liste de longueur $n+1$ contenant ses coefficients.

Le but de l'exercice est d'écrire un programme qui orthonormalise par le procédé de Schmidt la famille de polynômes $(1,X,X^2,\dots,X^n)$ (n entier quelconque) lorsqu'on munit $\mathbb{R}[X]$ du produit scalaire

$$\langle P\,|\,Q\rangle=\int_{-1}^{1}P(t)Q(t)\,\mathrm{d}t.$$

Pour cela on sera amené à écrire quelques procédures utiles pour travailler sur les polynômes. J'en cite quelques-unes, mais la liste n'est pas exhaustive :

- une procédure `valeur(P, x)` qui calcule la valeur d'un polynôme P en x, par l'algorithme de Hörner ;
- une procédure `mult_ext(c, P)` qui multiplie un polynôme P par une constante c ;
- une procédure `add_poly(P, Q)` qui additionne deux polynômes P et Q ;
- une procédure `mult_poly(P, Q)` qui multiplie deux polynômes P et Q ;
- une procédure `primitive(P)` qui renvoie une primitive du polynôme P ;
- une procédure `affiche_poly(P)` pour ceux qui veulent faire un joli affichage....
- une procédure `prod_scal(P, Q)` pour calculer le produit scalire de P et Q ;
- etc..

Remarque : il existe un package `numpy.polynomial` *qui contient ce genre de procédures, mais nous ne l'utiliseraons pas ici, le but de ce TD étant de vous faire programmer !*

C. Application à la recherche d'un polynôme annulateur d'une matrice.

Soit A une matrice de $\mathcal{M}_n(\mathbb{R})$. Puisque $\mathcal{M}_n(\mathbb{R})$ est un espace vectoriel de dimension n^2, la famille des puissances de A, $\left\{A^k,\ 0 \leqslant k \leqslant n^2\right\}$ est liée puisqu'elle comporte n^2+1 vecteurs. Il existe donc une relation de dépendance linéaire entre ces puissances, c'est-à-dire un polynôme P tel que $P(A)=0$. P s'appelle un polynôme annulateur de A.

L'espace vectoriel $\mathcal{M}_n(\mathbb{R})$ sera muni du produit scalaire canonique donné par

$$\langle A \,|\, B\rangle = \operatorname{tr}({}^tAB) = \sum_{i,j} a_{ij}b_{ij}.$$

Si l'on applique l'algorithme de Gram-Schmidt à la famille formée des puissances successives de A, cette famille étant liée, il existe nécessairement un rang k où, en reprenant les notations du début de l'exercice, le vecteur u_k est nul, c'est-à-dire où le vecteur v_k est combinaison linéaire des précédents. Le but de l'exercice est d'utiliser cette idée afin de trouver un polynôme annulateur d'une matrice A donnée, c'est à dire un plus petit entier k tel que A^k s'exprime comme combinaison linéaire des A^j pour $0 \leqslant j \leqslant k-1$.

Comme exemple, vous pourrez considérer la matrice $A = \begin{pmatrix} 3 & -4 & 0 & 2 \\ 4 & -5 & -2 & 4 \\ 0 & 0 & 3 & -2 \\ 0 & 0 & 2 & -1 \end{pmatrix}$ et vérifier que $A^4 = 2A^2 - I_4$.