Révisions

1 Chaînes et conversions

Exercice 1. *Représentation binaire*. La fonction `bin` permet d'obtenir la représentation binaire d'un entier, sous la forme d'une chaîne de caractères, préfixée par `'0b'`. Par exemple, `bin(11)` fournit `'0b1011'`. Écrire une fonction permettant d'obtenir à l'aide de `bin` la liste des bits d'un entier, du poids fort au poids faible. Par exemple, `binaire(11)` renverra `[1, 0, 1, 1]`. Objectif : une ligne !

Exercice 2. *Construction de chaîne*. Écrire une fonction permettant de construire la chaîne de caractères contenant tous les entiers de 0 à $N-1$: par exemple `chaine(12)` renverra `'01234567891011'`.

Exercice 3. *Affichage à l'écran*. Écrire une fonction affichant à l'écran un triangle d'étoiles, comme ceci :

```
>>> triangle(5)
*
**
***
****
*****
```

Exercice 4. *Suite de Conway*. La suite de Conway s'obtient à partir du premier terme $u_0 = 1$ comme suit : à chaque étape, on lit à voix haute u_n pour obtenir u_{n+1}. Les premiers termes sont les suivants :

- $u_0 = 1$;
- $u_1 = 11$ (car u_0 se lit « un 1 ») ;
- $u_2 = 21$ (car u_1 se lit « deux 1 ») ;
- $u_3 = 1211$ (car u_2 se lit « un 2, un 1 »)
- $u_4 = 111221$ (car u_3 se lit « un 1, un 2, deux 1 »)
- etc...

Écrire une fonction `suivant(s)` prenant en entrée un terme de la suite sous forme de chaîne de caractères, et renvoyant le suivant. Par exemple `suivant("1211")` renverra `"111221"`.

2 Algorithmique impérative : listes, boucles, conditions etc...

2.1 Exercices rapides

Exercice 5. *Création de listes*. Indiquer comment créer les listes suivantes, de plusieurs manières éventuellement (on ne demande pas d'écrire de fonction) :

1. liste contenant un nombre de zéros donnés (disons 100) ;
2. liste contenant tous les entiers de 1 à 100 ;
3. liste contenant tous les entiers de 1 à 100, répétés 5 fois (la liste débute par 1, 1, 1, 1, 1, 2,...) ;
4. liste contenant les entiers positifs inférieurs à 1000 divisibles par 3 ;
5. liste contenant les entiers positifs inférieurs à 1000 congrus à 1 modulo 3 ;
6. liste contenant les entiers positifs inférieurs à 1000 divisibles par 3 ou par 5 mais pas par 15.

Exercice 6. *Extraction de portions*. On part de la liste `L` contenant les entiers de 0 à 99. Utiliser un slicing pour obtenir :

1. les entiers de 23 à 35 ;
2. les entiers pairs entre 24 et 40 ;
3. les éléments de `L` divisibles par 10 ;

4. les éléments de `L` congrus à 5 modulo 10 (5, 15, etc...) ;
5. les éléments de `L` à l'envers (de 99 à 0) ;
6. les éléments de `L` à l'envers, qui sont pairs (98, 96, etc...)

Exercice 7. *Instruction conditionnelle.* Écrire une fonction `mois(m)` prenant en entrée un entier entre 1 et 12 indiquant un mois de l'année et renvoyant son nombre de jours (on suppose l'année non bissextile).

Exercice 8. *Instruction conditionnelle.* Écrire une fonction `choix(n)` prenant en entrée un entier n :
— l'affichant à l'écran s'il est positif ;
— si de plus, il est pair, il est renvoyé ;
— si l'entier est impair alors la la fonction renvoie $n+1$ s'il est divisible par 3 et $n+2$ s'il est congru à 1 modulo 3 ;
— la fonction ne renvoie rien dans les autres cas.

Exercice 9. *Boucle simple.* Écrire des fonctions calculant, avec une complexité $O(1)$ *en mémoire* :
1. la factorielle d'un entier ;
2. le n-ème terme de la suite de Fibonacci défini par $F_0 = F_1 = 1$ et $F_n = F_{n-1} + F_{n-2}$ pour $n \geq 2$.
3. l'évaluation d'un polynôme $P = \sum_{k=0}^{n-1} a_k X^k$ donné par la liste de ses coefficients `[`$a_0, a_1, \ldots, a_{n-1}$`]` en un réel x.
4. la même chose que précédemment, en utilisant le schéma de Hörner :

$$P(x) = a_0 + x \times (a_1 + x \times (a_2 + x \times (\cdots + xa_{n-1}) \cdots))$$

Exercice 10. *Plateau.* Écrire une fonction `plateau(L)` prenant en entrée une liste non vide et déterminant la longueur maximale d'un « plateau », c'est à dire le nombre maximal d'éléments consécutifs égaux. Attention en particulier à ne pas faire de dépassement d'indice et à traiter correctement un plateau situé à la fin de la liste.

Exercice 11. *Insertion.* Écrire une fonction `insere(L,x)` prenant en entrée une liste `L` croissante, et insérant x dans `L` de sorte que la liste reste croissante.

Exercice 12. *Décomposition en binaire.* Écrire une fonction permettant d'obtenir la décomposition en binaire de l'entier passé en paramètre, en *little endian* (bits de poids faibles à gauche) :

```
>>> binaire(11)
[1, 1, 0, 1]
```

Exercice 13. *Boucles imbriquées : polynômes de Lagrange.* On rappelle que si $a_0, \ldots, a_{n-1}$ sont des réels distincts, et $b_0, \ldots, b_{n-1}$ des réels quelconques, il existe un unique polynôme P de degré au plus $n-1$ vérifiant $P(a_i) = b_i$ pour tout i, de formule

$$P(x) = \sum_{i=0}^{n-1} b_i \prod_{\substack{0 \leq j \leq n-1 \\ j \neq i}} \frac{x - a_j}{a_i - a_j}$$

Écrire une fonction `Lagrange(A, B, x)` prenant en entrée deux listes de même taille n contenant les a_i et les b_i, et un réel x, et évaluant P en x. Donner la complexité en fonction de n.

Exercice 14. *Liste de listes.* Écrire une fonction `grille(n)` construisant une liste de n listes de taille n contenant tous les entiers entre 0 et $n^2 - 1$, comme suit :

```
>>> grille(3)
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
```

2.2 Cours

Exercice 15. Soit `L` une liste strictement croissante, et x un élément tel que `L[0]` $\leq x <$ `L[-1]`. Écrire une fonction inspirée de la recherche dichotomique cherchant l'unique indice i tel que `L[i]` $\leq x <$ `L[i+1]`. Justifier votre approche en exhibant un variant et un invariant de boucle. Quelle est sa complexité ?

Exercice 16. Savez vous réécrire le tri par insertion ? Quelle est sa complexité dans le pire cas, dans le meilleur cas ?

Exercice 17. Savez vous réécrire le tri à bulles ? Quelle est sa complexité dans le pire cas, dans le meilleur cas ?

Exercice 18. Savez vous réécrire le tri par sélection ? Quelle est sa complexité dans le pire cas, dans le meilleur cas ?

2.3 Exercices plus relevés

Exercice 19. *Crible d'Eratosthène (Mines 2019).* Le crible d'Ératosthène est un algorithme qui permet de déterminer la liste des nombres premiers appartenant à l'intervalle $[\![0, N-1]\!]$. Son pseudo-code s'écrit comme suit :

Algorithme 1 : Algorithme d'Eratosthène

Entrée : Un entier $N \geq 1$
Sortie : La liste des nombres premiers strictement inférieurs à N
$B \leftarrow$ liste de N booléens, tous vrais;
Marquer $B[0]$ et $B[1]$ comme faux;
pour *tout* i *de* 2 *à* $\lfloor\sqrt{N}\rfloor$ **faire**
  si $B[i]$ *est vrai* **alors**
    Marquer comme faux tous les multiples de i différents de i strictement inférieurs à N
Renvoyer la liste des p tels que $B[p]$ est vrai

Implémenter l'algorithme d'Eratosthène. On admet que

$$\sum_{\substack{p \text{ premier} \\ p<N}} \frac{1}{p} = O(\log\log N)$$

Montrer que la complexité de l'algorithme est en $O(N \log\log N)$ (et le reprendre si vous n'avez pas cette complexité !)

Exercice 20. *Calcul d'une plus longue sous séquence commune.* Dans ce problème, on se donne deux chaînes de caractères `s` et `t` dont on souhaite calculer une *plus longue sous séquence commune*, c'est-à-dire une chaîne de caractères `m` dont les caractères apparaissent dans le même ordre dans `s` comme dans `t`, maximale pour cette propriété. Par exemple, pour les chaînes `s = "XMJYAUZ"` et `t = "MZJAWXU"`, la chaîne `"MJAU"` est une plus longue sous séquence commune : les caractères `"M"`, `"J"`, `"A"` et `"U"` apparaissent dans cet ordre dans `s` et dans `t`, et on ne peut pas trouver plus long. Ce problème intervient fréquemment, par exemple en biologie dans l'analyse du génome pour décrire la proximité de deux individus : on cherche des plus longues sous-séquences communes dans des séquences d'ADN, décrites par leurs bases nucléiques (A, C, T et G).

Dans tout l'exercice, on note n la longueur de `s` et m la longueur de `t`. Pour $0 \leq i < n$ et $0 \leq j < m$, on note également s_i et t_j les i-ème et j-ème lettre de `s` et `t`, indicées à partir de 0 (ainsi `s` est constituée des lettres $s_0, \ldots, s_{n-1}$ et `t` des lettres $t_0, \ldots, t_{m-1}$). En suivant la notation Python, on note également `s[0:i]` et `t[0:j]` les préfixes de taille i et j de `s` et `t` (constitués des lettres jusqu'à s_{i-1} et t_{j-1}, en particulier `s[0:0]` et `t[0:0]` sont des chaînes vides). Enfin, on note $\ell_{i,j}$ la longueur maximale d'une sous-séquence commune à `s[0:i]` et `t[0:j]`, de sorte que le problème se résume à calculer une sous-séquence commune à `s` et `t` de taille $\ell_{n,m}$. La résolution du problème consiste à calculer d'abord $\ell_{n,m}$, puis une séquence associée.

1. Donner le nombre de couples (s', t') de sous-séquences de s et t. Vérifier qu'une recherche exhaustive est hors de propos sauf pour n et m très petits.
2. Montrer *soigneusement* que, pour tout $(i, j) \in [\![1, n]\!] \times [\![1, m]\!]$:

$$\ell_{i,j} = \begin{cases} 1 + \ell_{i-1,j-1} & \text{si } s_{i-1} = t_{j-1} \\ \max(\ell_{i-1,j}, \ell_{i,j-1}) & \text{sinon.} \end{cases}$$

3. Que vaut $\ell_{i,j}$ lorsque $i = 0$ ou $j = 0$?
4. En déduire (et l'écrire) une fonction récursive `longueur_plssc(s,t)` de calcul de $\ell_{n,m}$.

```
>>> longueur_plssc("XMJYAUZ","MZJAWXU")
4
```

5. Que pensez-vous de l'efficacité pratique de cet algorithme ?
6. Un algorithme plus efficace de calcul de $\ell_{n,m}$ consiste à utiliser un tableau pour stocker toutes les longueurs $(\ell_{i,j})_{0 \leq i \leq n, 0 \leq j \leq m}$, qu'on remplit de manière itérative (avec des boucles). Écrire `longueurs_plssc(s,t)`, une fonction renvoyant une liste `L` de $n+1$ listes à $m+1$ éléments, de sorte que `L[i][j]` soit égal à $\ell_{i,j}$. On pourra utiliser librement la fonction suivante de création d'une liste de $n+1$ listes de taille $m+1$:

```
def creer_tab(n,m):
    return [[0]*(m+1) for i in range(n+1)]
```

Voici, le résultat obtenu avec les deux chaînes prises en exemple :

```
>>> afficher_tab(longueurs_plssc("XMJYAUZ","MZJAWXU"))
0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1
0 1 1 1 1 1 1 1
0 1 1 2 2 2 2 2
0 1 1 2 2 2 2 2
0 1 1 2 3 3 3 3
0 1 1 2 3 3 3 4
0 1 2 2 3 3 3 4
```

La fonction `afficher_tab` étant une fonction qui affiche joliment une liste de listes sous forme de tableau.

7. Donner la complexité de `longueurs_plssc(s,t)` en fonction des tailles n et m des listes `s` et `t`.

8. On passe maintenant au calcul effectif d'une sous-chaîne commune. Un moyen simple est d'utiliser le tableau `L` calculé par `longueurs_plssc`, en partant de la case (n, m) jusqu'à remonter à la ligne $i = 0$ ou à la colonne $j = 0$. En notant $m_{i,j}$ une sous-séquence commune à `s[0:i]` et `t[0:j]` de taille maximale, on peut prendre pour $i \geq 1$ et $j \geq 1$:

$$m_{i,j} = \begin{cases} m_{i-1,j} & \text{si L[i][j] = L[i-1][j]} \\ m_{i,j-1} & \text{si L[i][j] = L[i][j-1]} \\ m_{i-1,j-1}\texttt{+}s_{i-1} & \text{sinon, car } s_{i-1} = t_{j-1} \text{ (+ est la concaténation de chaînes ici).} \end{cases}$$

Écrire une fonction `calcul_plssc(s,t)` suivant ce principe. On utilisera une boucle `while`. On pourra utiliser aussi le fait que `S.reverse()` permet de renverser les éléments d'une liste et `"".join(S)` permet de réunir par concaténation une liste de chaînes de caractères.

```
>>> calcul_plssc("XMJYAUZ","MZJAWXU")
'MJAU'
>>> calcul_plssc("ACTAAGCGCGTGAGAGTC", "CTGACGCGATGAAT")
'CTACGCGTGAAT'
```

Exercice 21. *somme maximale d'une portion.* Soit `L` une liste de taille n, constituée d'entiers relatifs. On cherche deux indices $i \leq j$ tels que la somme $s_{i,j} =$ `L[i]` $+$ `L[i+1]` $+ \cdots +$ `L[j]` soit maximale. Par exemple, pour la liste `[-1,2,1,4,-3,-5,6,2,-1,6,0,-1,-2,2]`, la portion maximale est de taille 4, et comprise entre les indices 6 et 9 inclus, c'est `[6, 2, -1, 6]`.

1. Combien y a-t-il de sommes de la forme $s_{i,j}$? Proposer une solution en temps $O(n^3)$. On ne demande pas de coder.

2. L'améliorer en une solution en $O(n^2)$.

3. On va utiliser la programmation dynamique pour obtenir une solution en $O(n)$. On note $S_j = \max\{s_{i,j} \mid 0 \leq i \leq j\}$. Trouver une relation entre S_j et S_{j-1} pour $j \geq 1$, permettant de calculer tous les (S_j) en temps $O(n)$.

4. Adapter votre solution pour trouver effectivement deux indices i et j tels que $s_{i,j}$ soit maximale. Coder effectivement une fonction `portion_max(L)` de complexité linéaire en la taille de la liste, renvoyant le couple d'indice (i, j) en question.

```
>>> portion_max([-1,2,1,4,-3,-5,6,2,-1,6,0,-1,-2,2])
(6, 9)
```

Exercice 22. *Carré de zéros dans une matrice binaire.* On considère une matrice de taille $n \times m$, constituée de zéros et de uns. On cherche à déterminer la taille maximale d'un carré dans la matrice, constitué uniquement de zéros. Par exemple, la matrice suivante possède un carré de zéros de taille 3×3 :

```
1 0 0 1 1 0 1 1 0
0 1 1 1 0 1 1 0 0
0 0 0 1 0 1 0 0 0
0 1 1 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0
1 1 0 0 1 0 1 1 0
```

Pour $0 \leq i < n$ et $0 \leq j < m$ on note $t_{i,j}$ la taille maximale d'un carré de zéro dans la matrice, dont le coin en bas à droite est indexé par (i, j) (on a donc $t_{i,j} = 0$ si le coefficient en case (i, j) de la matrice est 1).

1. Relier $t_{i,j}$ à $t_{i-1,j}$, $t_{i,j-1}$ et $t_{i-1,j-1}$ pour $i, j \geq 1$.
2. En déduire une méthode efficace pour calculer la taille maximale d'un carré de zéros dans la matrice.
3. L'implémenter en Python et donner sa complexité.

Exercice 23. *Rendu de monnaie*. On se donne $v_0, \dots, v_{n-1}$ des valeurs de pièces de monnaie, avec $v_0 = 1$. On peut supposer les v_i croissantes. On se donne une somme d'argent S à décomposer en les v_i (il y a au moins une solution car $v_0 = 1$), et on cherche à minimiser le nombre de pièces impliquées. En clair, on cherche à résoudre le problème :

$$\text{Trouver } (\alpha_0, \dots, \alpha_{n-1}) \text{ tel que } S = \sum_{i=0}^{n-1} \alpha_i v_i \quad \text{et} \quad \sum_{i=0}^{n-1} \alpha_i \text{ minimal.}$$

Dans un système monétaire usuel, il suffit de procéder en rendant le plus possible de pièces v_{n-1}, puis de pièces v_{n-2}, etc... Ce n'est pas le cas par exemple pour $(v_0, v_1, v_2) = (1, 3, 4)$, où pour $S = 6$ il vaut mieux utiliser deux pièces 3 que trois pièces 1, 1 et 4. Chercher une méthode de résolution du problème avec une complexité $O(Sn)$.

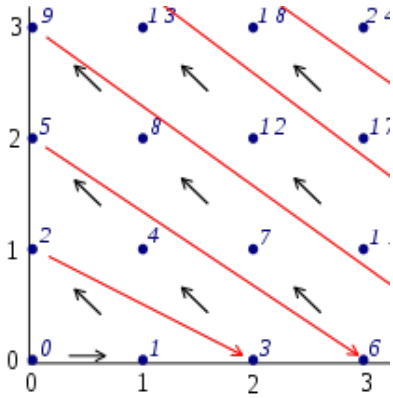
3 Récursivité

Exercice 24. Écrire une fonction récursive `sdc(n)` retournant la somme des chiffres de l'entier supposé naturel passé en argument.

```
>>> sdc(1234)
10
```

Exercice 25. Écrire un tri par sélection récursif. Analyser sa complexité.

Exercice 26. *Diagonale de Cantor*. On numérote les couples d'entiers naturels de la façon suivante.



Écrire une fonction récursive `enum(x,y)` associant à un couple d'entiers son numéro dans l'énumération ci-dessus. Réciproquement, écrire une fonction récursive `couple` renvoyant le couple associé à un entier.

```
>>> enum(1,2)
8
>>> couple(8)
(1, 2)
```

Remarque : ne cherchez pas une formule explicite : contentez vous de suivre les flèches !

Exercice 27. *Listes de 0 et de 1*. Le but de cet exercice est de produire toutes les listes d'une taille N donnée, constituées de 0 et de 1. Il y en a 2^N.

1. Proposer une approche itérative. On pourra s'inspirer de l'addition en binaire. Coder effectivement la fonction `toutes_listes(N)`.

```
>>> toutes_listes(3)
[[0, 0, 0], [1, 0, 0], [0, 1, 0], [1, 1, 0], [0, 0, 1], [1, 0, 1], [0, 1, 1], [1, 1, 1]]
```

2. Proposer une approche récursive, et donner son code Python (l'ordre dans la liste résultat n'a pas d'importance).

Exercice 28. *Tri rapide pas en place.* Écrire une fonction `partition(L,x)` prenant en entrée une liste `L`, et renvoyant deux listes `L1` et `L2`, telle que `L1` est constituée des éléments de `L` inférieurs ou égaux à `x` et `L2` des éléments strictement supérieurs à `x`. En déduire une fonction `tri_rapide(L)` renvoyant une copie triée de `L`. Quelle est la complexité dans le pire cas ? le meilleur cas ? le cas moyen ?

Exercice 29. *Tri fusion.* Savez vous réécrire le tri fusion ? Quelle est sa complexité dans le pire cas ? le meilleur cas ? le cas moyen ?

Exercice 30. *D'après Centrale 19.* On considère dans cet exercice une structure de données mélangeant entiers et listes. Un élément de cette structure est soit un entier, soit une liste d'éléments de la structure (la définition est récursive). Par exemple, `3`, `[4, [2, 5]]` ou `[[[2], 1], [3, 4, [5]], 0]` sont de tels éléments.

1. Écrire une fonction `somme(x)` prenant en entrée un élément de la structure, et renvoyant la somme de tous les entiers présents. On pourra effectuer un test du type `type(x) == int` ou `type(x) == list` (qui s'évalue en un booléen assez évocateur).

```
>>> somme(3), somme([4, [2, 5]]), somme([[[2], 1], [3, 4, [5]], 0])
(3, 11, 15)
```

2. De même, écrire une fonction faisant une copie « en profondeur » de l'élément de la structure. Pour mémoire :

— si `x` est un entier, il est immuable : pas besoin de copie !

— si `x` est une liste, alors `x[:]` est une copie de `x`. Mais si `x` contient elle-même des listes, celles-ci ne sont pas copiées.

```
>>> L=[[[2], 1], [3, 4, [5]], 0]
>>> M=copie(L)
>>> M[0][0][0] = 42
>>> L, M
([[[2], 1], [3, 4, [5]], 0], [[[42], 1], [3, 4, [5]], 0])
```

Remarque : la fonction `deepcopy` du module `copy` fonctionne sur ce principe.

3. Reprendre la question 1 sans récursivité. On pourra faire usage d'une liste vue comme une pile.

4 Analyse numérique

Exercice 31. *D'après Centrale 19.*

1. On travaille avec une représentation des flottants usuelle, c'est à dire sur 64 bits. Quel est le nombre de bits alloués à la mantisse ? (Question subsidiaire, qu'est-ce que la mantisse ?)
2. Quel est le nombre de chiffres significatifs décimaux avec cette représentation ?
3. On peut approcher la dérivée $f'(x)$ d'une fonction f en $x \neq 0$ via la formule suivante : $f'(x) \simeq \frac{f(x(1+h))-f(x)}{xh}$, avec h petit. $h = 10^{-15}$ est-il acceptable ? $h = 1$ l'est-il ? Justifier.
4. On cherche un h optimal dans cette question.
   (a) À l'aide d'un développement limité, donner un équivalent de l'erreur commise mathématiquement en fonction de $f''(x)$. (On suppose f de classe $\mathcal{C}^2$)
   (b) Lorsqu'on additionne deux nombres flottants a et b, on commet une erreur du à la représentation flottante majorée par $\varepsilon(|a| + |b|)$, où $\varepsilon = 2^{-53}$ ici. Donner une majoration de l'erreur commise à cause de la représentation flottante dans la formule donnant $f'(x)$ (on négligera les erreurs résultant de multiplications).
   (c) Trouver h tel que la somme des deux erreurs précédentes soit minimale. On supposera $f(x) = x^2$ pour le calcul.

Exercice 32. Que savez vous de la méthode de Newton et de la méthode dichotomique pour chercher un zéro d'une fonction de la forme $f(x) = 0$?

Exercice 33. *Rang d'une matrice.* On applique la méthode du pivot de Gauss (similaire à celle du cours de mathématiques) pour calculer le rang d'une matrice, on obtient une fonction `rang`. On exécute le code suivant :

```
>>> L1=[168, 168, 168]
>>> L2=[48, 51, 67]
>>> L3=[216, 219, 235]
>>> rang([L1,L2,L3])
3
```

Le rang de la matrice est 2 puisque `L3` est la somme des deux autres lignes. Qu'a-t-il pu se produire ? Expliquer.

Exercice 34. *Moindres carrés.* On se donne un nuage de points (x_i, y_i) du plan (les x_i sont distincts). On cherche la droite $y = ax + b$ qui minimise la somme des carrés des distances entre un point du nuage et sa projection sur la droite parallèlement à (Oy), à savoir la quantité $S = \sum_{i=0}^{n-1}(y_i - ax_i - b)^2$. Écrire les dérivées partielles par rapport à a et b. On admet que le point (a, b) annulant ces deux dérivées partielles correspond au minimum cherché. Résoudre le système obtenu et écrire un code Python `reg_lin(X,Y)` prenant en entrée les deux listes de points (x_i) et (y_i) et renvoyant (a, b). Quelle est sa complexité ?

5 Tableaux Numpy et utilisation des modules

Remarque : à travailler un peu pour Centrale et CCP Modélisation (PC), moins important pour X et Mines. Voici l'annexe qu'on trouve dans le sujet Centrale 19 :

Fonctions

- `range(n)` renvoie la séquence des n premiers entiers $(0 \to n-1)$
  `list(range(5))` → `[0, 1, 2, 3, 4]`
- `random.randrange(a, b)` renvoie un entier aléatoire compris entre a et $b-1$ inclus (a et b entiers)
- `random.random()` renvoie un nombre flottant tiré aléatoirement dans $[0, 1[$ suivant une distribution uniforme
- `random.shuffle(u)` permute aléatoirement les éléments de la liste `u` (modifie `u`)
- `random.sample(u, n)` renvoie une liste de n éléments distincts de la liste `u` choisis aléatoirement, si $n >$ `len(u)`, déclenche l'exception `ValueError`
- `math.sqrt(x)` calcule la racine carrée du nombre x
- `round(n)` arrondit le nombre n à l'entier le plus proche
- `math.ceil(x)` renvoie le plus petit entier supérieur ou égal à x
- `math.floor(x)` renvoie le plus grand entier inférieur ou égal à x

Opérations sur les listes

- `len(u)` donne le nombre d'éléments de la liste `u` :
  `len([1, 2, 3])` → `3` ; `len([[1,2], [3,4]])` → `2`
- `u + v` construit une liste constituée de la concaténation des listes `u` et `v` :
  `[1, 2] + [3, 4, 5]` → `[1, 2, 3, 4, 5]`
- `n * u` construit une liste constituée de la liste u concaténée n fois avec elle-même :
  `3 * [1, 2]` → `[1, 2, 1, 2, 1, 2]`
- `e in u` et `e not in u` déterminent si l'objet `e` figure dans la liste `u`, cette opération a une complexité temporelle en $O($`len(u)`$)$
  `2 in [1, 2, 3]` → `True` ; `2 not in [1, 2, 3]` → `False`
- `u.append(e)` ajoute l'élément `e` à la fin de la liste `u` (similaire [1] à `u = u + [e]`)
- `u.pop(i)` : renvoie l'élément à l'indice i de la liste `u` et le supprime
- `del u[i]` supprime de la liste `u` son élément d'indice i
- `del u[i:j]` supprime de la liste `u` tous ses éléments dont les indices sont compris dans l'intervalle $[i, j[$.
- `u.remove(e)` supprime de la liste `u` le premier élément qui a pour valeur `e`, déclenche l'exception `ValueError` si `e` ne figure pas dans `u`, cette opération a une complexité temporelle en $O($`len(u)`$)$
- `u.insert(i, e)` insère l'élément `e` à la position d'indice i dans la liste `u` (en décalant les éléments suivants) ; si $i >=$ `len(u)`, `e` est ajouté en fin de liste
- `u[i], u[j] = u[j], u[i]` permute les éléments d'indice i et j dans la liste `u`

1. Remarque de ma part : la complexité de l'opération `u=u+[e]` est linéaire en le nombre d'éléments du résultat, car une autre liste est créée. C'est très mauvais et à éviter.

Opérations sur les tableaux (np.ndarray)

- `np.array(u)` crée un nouveau tableau contenant les éléments de la séquence `u`. La taille et le type des éléments de ce tableau sont déduits du contenu de `u`
- `np.empty(n, dtype)`, `np.empty((n, m), dtype)` crée respectivement un vecteur à n éléments ou un tableau à n lignes et m colonnes dont les éléments, de valeurs indéterminées, sont de type `dtype` qui peut être un type standard (`bool`, `int`, `float`, ...) ou un type spécifique numpy (`np.int16`, `np.float32`, ...). Si le paramètre dtype n'est pas précisé, il prend la valeur `float` par défaut
- `np.zeros(n, dtype)`, `np.zeros((n, m), dtype)` fonctionne comme `np.empty` en initialisant chaque élément à la valeur zéro pour les types numériques ou `False` pour les types booléens
- `a.ndim` nombre de dimensions du tableau `a`
- `a.shape` tuple donnant la taille du tableau `a` pour chacune de ses dimensions
- `len(a)` taille du tableau `a` dans sa première dimension, équivalent à `a.shape[0]`
- `a.size` nombre total d'éléments du tableau `a`
- `a.flat` itérateur sur tous les éléments du tableau `a`
- `np.ndenumerate(a)` itérateur sur tous les couples (index, élément) du tableau `a` où « index » est un tuple de `a.ndim` entiers donnant les indices de l'élément
- `a.min()`, `a.max()` renvoie la valeur du plus petit (respectivement plus grand) élément du tableau `a` ; ces opérations ont une complexité temporelle en $O($`a.size`$)$
- `b in a` détermine si `b` est un élément du tableau `a` ; si `b` est un scalaire, vérifie si `b` est un élément de `a` ; si `b` est un vecteur ou une liste et `a` un tableau à deux dimensions, détermine si `b` est une ligne de `a`
- `np.concatenate((a1, a2))` construit un nouveau tableau en concaténant deux tableaux selon leur première dimension ; `a1` et `a2` doivent avoir le même nombre de dimensions et la même taille à l'exception de leur taille dans la première dimension (deux tableaux à deux dimensions doivent avoir le même nombre de colonnes pour pouvoir être concaténés) :

```
>>> np.concatenate((np.array([[1,2],[3,4]]),np.array([[2,3],[5,6]])))
array([[1, 2],
       [3, 4],
       [2, 3],
       [5, 6]])
```

  On peut utiliser le paramètre optionnel `axis` pour concaténer « horizontalement » (`axis=0` est la valeur par défaut) :

```
>>> np.concatenate((np.array([[1,2],[3,4]]),np.array([[2,3],[5,6]])), axis=1)
array([[1, 2, 2, 3],
       [3, 4, 5, 6]])
```

- `np.transpose(a)` renvoie le transposé du tableau `a`
- `np.dot(a, b)` calcule le produit matriciel des tableaux `a` et `b`
- `np.linalg.inv(a)` renvoie l'inverse du tableau `a`, lève l'exception `ValueError` si `a` n'est pas un tableau carré à deux dimensions et `LinAlgError` si `a` n'est pas inversible

Exercice 35. Écrire une fonction `modif(t,k)` prenant en entrée un tableau Numpy à une dimension de taille n contenant des réels de $[0, 1[$ et un entier $k \leq n$, et modifiant aléatoirement k coefficients consécutifs de `t` (la position de la plage d'indices modifiés sera également choisie au hasard).

Exercice 36. *Génération aléatoire, d'après Centrale 19*. Écrire une fonction `tab_aleatoire(n,c)` prenant en entrée deux entiers n et c strictement positifs, et renvoyant un tableau Numpy de taille $n \times 2$ constitués d'entiers aléatoires de $[\![0, c-1]\!]$, tel que les lignes soient deux à deux distinctes.

Exercice 37. *D'après Centrale 18*. D'une photographie en noir et blanc on peut obtenir en Python un tableau Numpy d'éléments de $[\![0, 255]\!]$: type `uint8`, pour entier (*int*) non signé (*Unsigned*, c'est-à-dire positif), sur 8 bits. Que fait la fonction suivante, prenant en entrée un tel tableau :

```
def h(t):
    L=np.zeros(256)
    for x in t.flat:
        L[x]+=1
    plt.plot(list(range(256),L))
```

où `matplotlib.pyplot` a classiquement été importé sous le nom `plt`.

Exercice 38. *Remplissage d'un tableau.* Si `M` est un tableau Numpy à deux dimensions, `M[i,:]=s` permet de remplacer la i-ème ligne de `M` par les éléments de la séquence `s` (sous réserve qu'elle est autant d'éléments que `M` a de colonnes). De même, `M[:,i]=s` permet de remplacer tous les éléments d'une colonne. On rappelle aussi qu'on peut agir sur un tableau Numpy via les opérations usuelles (+, -,etc...) et un scalaire, les opérations s'appliquent terme à terme. Écrire une fonction `VDM(L)` prenant en entrée une liste d'éléments $x_0, \dots, x_{n-1}$ et renvoyant la matrice de Van Der Monde associée (la i-ème colonne, contient $(x_i^j)_{0\leq j<n}$).

Exercice 39. Si `t` est un tableau Numpy, une opération comme `t >= 0` produit un tableau Numpy de même taille que `t`, contenant des booléens (les résultats de chaque test `v >= 0` pour `v` une valeur du tableau. De plus, si `b` est un tableau de booléens, `b.any()` (resp. `b.all()`) renvoie un booléen indiquant s'il y a au moins un `True` dans `b` (resp. si toutes les valeurs de `b` sont `True`). En utilisant cette syntaxe, écrire des instructions (courtes !) permettant de tester :

— si toutes les valeurs de `t` sont positives ;

— si toutes les valeurs de `t` sont entre deux réels a et b fixés ;

— s'il existe une valeur de `t` inférieure à $M/2$, où M est le maximum de `t`.

Exercice 40. *Création par blocs.* À l'aide de `np.concatenate` écrire une fonction `double(M)` prenant en entrée une matrice M et renvoyant la matrice par blocs :

$$\begin{pmatrix} M & 2M \\ 2M & 3M \end{pmatrix}$$

Exercice 41. *Retour sur les moindres carrés.* Voici un extrait de l'aide de la fonction `scipy.optimize.leastsq` :

```
leastsq(func, x0, ...)
    Minimize the sum of squares of a set of equations.

    ::

        x = arg min(sum(func(y)**2,axis=0))
                 y

    Parameters
    ----------
    func : callable
        should take at least one (possibly length N vector) argument and
        returns M floating point numbers. It must not return NaNs or
        fitting might fail.
    x0 : ndarray
        The starting estimate for the minimization.

...

    Returns
    -------
    x : ndarray
        The solution (or the result of the last iteration for an unsuccessful
        call).
    cov_x : ndarray
        Uses the fjac and ipvt optional outputs to construct an
        estimate of the jacobian around the solution. None if a
        singular matrix encountered (indicates very flat curvature in
        some direction).  This matrix must be multiplied by the
        residual variance to get the covariance of the
        parameter estimates -- see curve_fit.
```

À l'aide de cette fonction, écrire une fonction trouvant les coefficient de la droite $x \mapsto ax + b$ approchant au mieux un ensemble de points $((x_i, y_i))_{0\leq i<n}$, au sens ou la quantité $\sum_{i=0}^{n-1}(y_i - ax_i - b)^2$ est minimale.

```
>>> moindre_carres([[0,1], [2,4]]) #la droite passant par ces deux points est x -> 1.5x +1
array([ 1.5,  1. ])
>>> moindre_carres([[0,1], [2,4], [-2,1]]) #la plus proche des trois points !
array([ 0.75,  2.  ])
```