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 !

Corrigé. En une ligne.

```
def binaire(n): return [int(x) for x in bin(n)[2:]]
```

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 `chaîne(12)` renverra `'01234567891011'`.

Corrigé. Deux versions :

- `+` permet la concaténation de chaînes de caractères ;
- si `s` est une chaîne et `L` une liste de chaînes, `s.join(L)` permet de réunir les éléments de `L` avec `s` comme séparateur. En particulier avec `s` la chaîne vide, cela concatène les chaînes de `L`.

```
def chaîne(n):
    s=""
    for i in range(n):
        s+=str(i)
    return s
```

```
def chaîne(n):
    L=[]
    for i in range(n):
        L.append(str(i))
    return "".join(L)
```

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

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

Corrigé. `print` effectue par défaut un saut de ligne après avoir affiché son (ses) argument(s) à l'écran.

```
def triangle(n):
    for i in range(1,n+1):
        print("*"*i)
```

Autre version n'utilisant qu'une fois `print` :

```
def triangle(n):
    L=[]
    for i in range(1,n+1):
        L.append("*"*i)
    print("\n".join(L))
```

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

Corrigé. On progresse dans la chaîne, en regardant à chaque fois si le caractère courant est égal au précédent ou non. Pour ne pas traiter à part le dernier caractère, la boucle va un cran plus loin, mais on fait attention à ne pas accéder à `s[i]` pour $i = \text{len}(s)$: on traite ce cas comme si le caractère était différent du précédent.

```
def suisvant(s):
    L=[]
    x=s[0]
    nb=1
    for i in range(1, len(s)+1):
        if i==len(s) or s[i]!=x:
            L.append(str(nb)+x)
            if i<len(s):
                x,nb=s[i],1
        else:
            nb+=1
    return "".join(L)
```

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.

Corrigé.

```
# 100 zéros
L = [0]*100

# 3 méthodes pour la liste des entiers de 1 à 100
L = list(range(1,101))
L = [i for i in range(1,101)]
L = []
for i in range(1,101):
    L.append(i)

# 2 méthodes pour les entiers répétés 5 fois
L = []
for i in range(1,101):
    for j in range(5):
        L.append(i)
L = []
for i in range(1,101):
    L+=[i]*5 #ou L.extend([i]*5)

#Par compréhension pour les questions 4-5-6
L = [i for i in range(1001) if i%3==0]
L = [i for i in range(1001) if i%3==1]
L = [i for i in range(1001) if (i%3==0 or i%5==0) and i%15!=0]
```

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...)

Corrigé. On rappelle que $L[i:j:p]$ construit la liste des éléments $L[i]$, $L[i+p]$, $L[i+2p]$... avec $i + kp < j$ si $p > 0$ et $i + kp > j$ si $p < 0$. Si i (resp. j) n'est pas renseigné, il vaut 0 (resp. $\text{len}(L)$) si $p > 0$, et $\text{len}(L)-1$ (resp. -1) si $p < 0$.

```
L[23:36]
L[24:41:2]
L[::10]
L[5::10]
L[::-1]
L[-2::-2]
```

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).

Corrigé. Par exemple :

```
def mois(m):
    if m==2:
        return 28
    elif m in [1,3,5,7,8,10,12]:
        return 31
    else:
        return 30
```

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.

Corrigé.

```
def choix(n):
    if n>=0:
        print(n)
        if n%2==0:
            return n
    if n%2==1:
        if n%3==0:
            return n+1
        elif n%3==1:
            return n+2
```

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 $[a_0, a_1, \dots, a_{n-1}]$ en un réel x .
4. la même chose que précédemment, en utilisant le schéma de Hörner :

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

Corrigé. Voici $O(1)$ en mémoire signifie en particulier qu'on ne peut utiliser de liste pour répondre à la question (stockant les termes de la suite de Fibonacci ou les puissances de x).

```
def fact(n):
    p=1
    for i in range(1,n+1):
        p*=i
    return p

def fibo(n):
    a,b=1,1
    for i in range(n):
        a,b=a+b,a
    return b

def evalue(P,x):
    p, s = 1, 0
    for i in range(len(P)):
        s+=P[i]*p
        p*=x
    return s

def evalue(P,x):
    s = 0
    for i in range(len(P)-1,-1,-1):
        s=s*x+P[i]
    return s
```

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.

Corrigé. Les deux solutions suivantes sont en $O(n)$ avec n la taille de la liste.

```
def plateau(L):
    v,k = L[0], 1 #valeur, longueur du plateau courant
    kmax = 1 #longueur du plateau maximal
    for i in range(1,len(L)):
        if L[i]==L[i-1]:
            k+=1
            if k>kmax:
                kmax=k
        else:
            v,k=L[i],1
    return kmax
```

```
def plateau(L):
    i=1
    while i<len(L) and L[i]==L[i-1]:
        i+=1
    kmax = i #premier plateau
    j=i
    while j<len(L):
        i=j
        j+=1
        while j<len(L) and L[j]==L[j-1]:
            j+=1
        kmax = max(kmax, j-i)
    return kmax
```

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.

Corrigé. En s'inspirant du tri par insertion (mais on permute comme dans le tri à bulles) :

```
def insere(L,x):
    L.append(x)
```

```
i=len(L)-1
while i>0 and L[i-1]>L[i]:
    L[i], L[i-1] = L[i-1], L[i]
    i-=1
```

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]
```

Corrigé. Classique, à savoir faire sans réfléchir.

```
def binaire(n):
    L=[]
    while n>0:
        L.append(n%2)
        n=n//2
    return L
```

Exercice 13. *Boucles imbriquées : polynômes de Lagrange.* On rappelle que si $a_0, \dots, a_{n-1}$ sont des réels distincts, et $b_0, \dots, b_{n-1}$ 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 .

Corrigé. Complexité $O(n^2)$, cohérente avec la formule :

```
def Lagrange(A, B, x):
    s=0
    n=len(A)
    for i in range(n):
        p=1
        for j in range(n):
            if j!=i:
                p*=(x-A[j])/(A[i]-A[j])
        s+=p*B[i]
    return s
```

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]]
```

Corrigé. Par exemple.

```
def grille(n):
    L=[]
    for i in range(n):
        U=[]
        for j in range(n*i, n*(i+1)):
            U.append(j)
        L.append(U)
    return L

def grille(n):
    return [list(range(n*i, n*(i+1))) for i in range(n)]
```

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 ?

Corrigé.

```
def dichot(L, x):
    a, b = 0, len(L) - 1
    while b - a > 1:
        m = (a + b) // 2
        if L[m] <= x:
            a = m
        else:
            b = m
    return a
```

- Variant de boucle : $b - a$ (en effet, lors d'un tour de boucle, $b - a \geq 2$, et soit a soit b prend la valeur m vérifiant $a < m < b$).
- Invariant de boucle : « $L[a] \leq x$ et $L[b] > x$. Le fait que ce soit un invariant est clair. Après la boucle, on a donc $b - a \leq 1$ et $L[a] \leq x < L[b]$. Donc $b = a + 1$ et a est l'entier cherché.
- Complexité : $O(\log n)$ avec n la taille de L . En effet, en notant p la plus petite puissance de 2 supérieure ou égale à $b - a$, on vérifie que la quantité $b - a$ est inférieure à $\frac{p}{2^k}$ après k étapes. Or cette quantité vaut 1 pour $k = \log_2(p) = O(\log n)$.

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 ?

Corrigé. cf cours !

2.3 Exercices plus relevés

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

Algorithme 1 : Algorithme d'Ératosthène

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'Ératosthène. 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!)

Corrigé. On utilise une boucle **while** pour éviter de faire un calcul de racine carrée, mais une boucle **for** parcourant tous les entiers de 2 à $N - 1$ ne changeait en fait rien à la complexité.

```
def eratosthene(N):
    B=[True]*N
    B[0]=False
    B[1]=False
    i=2
    while i**2<N:
        if B[i]:
            for m in range(2*i, N, i):
                B[m]=False
            i+=1
    return [p for p in range(N) if B[p]]
```

pour chaque p premier, la boucle `for` fait $O(\frac{N}{p})$ étapes. La somme des complexité de ces boucles est donc

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

À cela s'ajoute un $O(N)$ du à la boucle `while`, qui ne change rien.

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 $s = \text{"XMJYAUZ"}$ et $t = \text{"MZJAWXU"}$, 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 $s_0, \dots, s_{n-1}$ et t des lettres $t_0, \dots, t_{m-1}$). 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 $(i, j) \in \llbracket 1, n \rrbracket \times \llbracket 1, m \rrbracket$:

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

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 $j \geq 1$:

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

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'
```

Corrigé. 1. Une sous-séquence non vide de s s'obtient en sélectionnant les éléments de la séquence : pour chaque caractère il y a deux choix (appartient ou n'appartient pas), soit 2^n . Il y a donc 2^{n+m} couples de sous-séquences possibles. Examiner tous ces couples n'est donc pas faisable pour des raisons de complexité, sauf si n et m sont petits.

2.
 - Supposons $s_{i-1} = t_{j-1}$. À partir d'une sous-séquence commune à $s[0:i-1]$ et $t[0:j-1]$ de longueur $l_{i-1,j-1}$, on obtient une sous-séquence commune à $s[0:i]$ et $t[0:j]$ en rajoutant s_{i-1} . D'où $l_{i,j} \geq 1 + l_{i-1,j-1}$. Réciproquement, une sous-séquence commune à $s[0:i]$ et $s[0:j]$ termine nécessairement par s_{i-1} , et supprimer ce caractère donne une sous-séquence commune à $s[0:i-1]$ et $s[0:j-1]$, d'où l'inégalité réciproque. Ainsi $l_{i,j} = 1 + l_{i-1,j-1}$ dans ce cas.
 - Supposons $s_{i-1} = t_{j-1}$. Déjà, $l_{i,j} \geq l_{i-1,j}$ est clair, de même que $l_{i,j} \geq l_{i,j-1}$, d'où $l_{i,j} \geq \max(l_{i-1,j}, l_{i,j-1})$. Réciproquement, une sous-séquence maximale commune à $s[0:i]$ et $t[0:j]$ ne termine soit pas par s_{i-1} , soit pas par t_{j-1} (soit les deux). Ainsi c'est une sous-séquence commune à $s[0:i]$ et $t[0:j-1]$ ou à $s[0:i-1]$ et $t[0:j]$, d'où l'inégalité réciproque. Ainsi $l_{i,j} = \max(l_{i-1,j}, l_{i,j-1})$ dans ce cas.
3. Si $i = 0$ ou $j = 0$, alors $l_{i,j} = 0$ car dans ce cas la seule séquence commune à $s[0:i]$ et $t[0:j]$ est la séquence vide.
4. Voici :

```
def longueur_plssc(s,t):
    if s=="" or t=="":
        return 0
    elif s[-1]==t[-1]:
        return 1+longueur_plssc(s[:-1],t[:-1])
    else:
        return max(longueur_plssc(s,t[:-1]), longueur_plssc(s[:-1],t))
```

5. Dans le pire cas, la complexité est exponentielle en la somme des tailles de s et t . Par exemple, si les caractères de s sont différents de ceux de t , il y a deux appels récursifs à chaque étape, et il y a évidemment chevauchement. En pratique cet algorithme n'est pas efficace.
6. On suit scrupuleusement la formule de la question 2 :

```
def longueurs_plssc(s,t):
    n,m=len(s), len(t)
    T=creer_tab(n,m)
    for i in range(1,n+1):
        for j in range(1,m+1):
            if s[i-1]==t[j-1]:
                T[i][j]=1+T[i-1][j-1]
            else:
                T[i][j]=max(T[i-1][j], T[i][j-1])
    return T
```

7. La complexité est $O(nm)$.

8. Voici (on s'arrête lorsque $T[i][j] = 0$, ce qui inclus le cas $i = 0$ ou $j = 0$)

```
def calcul_plssc(s,t):
    n,m=len(s), len(t)
    T=longueurs_plssc(s,t)
    S=[]
    i,j=n,m
    while T[i][j]>0:
        if s[i-1]==t[j-1]:
            S.append(s[i-1])
            i-=1
            j-=1
        elif T[i][j]==T[i-1][j]:
            i-=1
        else:
            j-=1
    S.reverse()
    return "".join(S)
```

Remarque : en dehors du calcul des longueurs $\ell_{i,j}$, la fonction précédente est en complexité $O(n + m)$.

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 $s_{i,j} = L[i] + L[i+1] + \dots + L[j]$ 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)
```

Corrigé.

1. Il y en $\binom{n}{2} + n = \frac{n(n-1)}{2} + n = \frac{n(n+1)}{2}$. Chaque somme $s_{i,j}$ se calcule en temps $O(j - i + 1) = O(n)$, il y en a $O(n^2)$, d'où l'algorithme en $O(n^3)$ consistant à calculer toutes les $s_{i,j}$ et garder le maximum.
2. On remarque que pour $j > i$, $s_{i,j}$ se calcule en $O(1)$ à partir de $s_{i,j-1}$, car $s_{i,j} = s_{i,j-1} + L[j]$. On peut donc réduire la complexité précédente à $O(n^2)$.
3. On remarque que $S_j = L[j] + \max(S_{j-1}, 0)$. En effet, à partir d'une portion de poids S_{j-1} terminant à l'indice $j - 1$, on en construit une de poids $L[j] + S_{j-1}$ en rajoutant $L[j]$. En prenant $L[j]$ seul, on obtient une portion de poids $L[j]$, d'où $S_j \geq L[j] + \max(S_{j-1}, 0)$. L'inégalité réciproque est assez immédiate. La relation permet bien de calculer tous les S_j en temps $O(n)$.
4. L'algorithme consiste d'abord à trouver j tel que S_j maximal, puis chercher i tel que $s_{i,j} = S_j$:

```

def portion_max(L):
    s, smax, jmax=L[0], L[0], 0
    #calcul du j tel que Sj soit maximal
    for j in range(1, len(L)):
        s=L[j]+max(s, 0)
        if s>smax:
            smax, jmax = s, j
    #calcul du i tel que s_{i, jmax}=s_max
    i=jmax
    s=L[i]
    while s<smax:
        i-=1
        s+=L[i]
    return i, jmax

```

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.

Corrigé.

1. On a

$$t_{i,j} = \begin{cases} 0 & \text{si } m_{i,j} = 1 \\ 1 + \min(t_{i-1,j}, t_{i,j-1}, t_{i-1,j-1}) & \text{sinon.} \end{cases}$$

En effet si a est tel que $t_{i-1,j} \geq a$, $t_{i,j-1} \geq a$ et $t_{i-1,j-1} \geq a$, alors $t_{i,j} \geq a + 1$. Réciproquement, si l'un des $t_{i-1,j}$, $t_{i,j-1}$ ou $t_{i-1,j-1}$ est strictement inférieur à a , alors le carré de taille $a + 1$ dont le coin en bas à droite est (i, j) contient nécessairement un 1.

2. Algorithme : calculer tous les $t_{i,j}$ via la relation précédente et prendre le maximum.
3. Avec n et m les dimensions de la matrice, l'algorithme est en $O(nm)$.

```

def max_zeros(M):
    n, m=len(M), len(M[0])
    tmax = 0
    T=[[0]*m for i in range(n)]
    for i in range(n):
        for j in range(m):
            if M[i][j]==0:
                T[i][j]=1+min(T[i-1][j], T[i][j-1], T[i-1][j-1])
                tmax = max(tmax, T[i][j])
    return tmax

```

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 :

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

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 $O(Sn)$.

Corrigé. Pour tout $1 \leq s \leq S$, on peut calculer le nombre minimal de pièces nécessaires, noté m_s : $m_s = 1 + \min_{v \in V} m_{s-v}$, le minimum portant sur l'ensemble des pièces $v \in V$ telles que $s - v \geq 0$, avec $m_0 = 0$. Le code suivant calcule pour chaque s ce minimum ainsi qu'un v tel que $m_s = 1 + m_{s-v}$. Il est ensuite facile de reconstruire l'ensemble des pièces nécessaires pour décomposer S , en remontant de S à 0 :

```
def rendu_monnaie(V,S):
    n=len(V)
    #dans M, calcul du nombre minimal de pièces
    M=[(S,1)]*(S+1)
    M[0]=(0,1)
    for s in range(1,S+1):
        for x in V:
            if s-x>=0 and M[s-x][0]+1<M[s][0]:
                M[s]=(M[s-x][0]+1,x)
    P=[] #liste des pièces à utiliser
    s=S
    while s>0:
        t,x=M[s]
        P.append(x)
        s=s-x
    return P
```

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
```

Corrigé. Rappel : `//` est la division entière. Par exemple $1234//10$ vaut 123.

```
def sdc(n):
    if n==0:
        return 0
    else:
        return n%10 + sdc(n//10)
```

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

Corrigé. La fonction suivante renvoie une copie triée de la liste passée en paramètre. Elle fait usage d'une fonction `min_reste(L)` prenant en entrée une liste non vide, et renvoyant son minimum, et l'ensemble des éléments de L privés de ce minimum.

```
def min_reste(L):
    im=0 #indice du min
    for i in range(1,len(L)):
        if L[i]<L[im]:
            im=i
    return L[im], L[:im]+L[im+1:]

def tri_selection(L):
    if L==[]:
        return []
    else:
        m,r=min_reste(L)
        return [m]+tri_selection(r)
```

La complexité vérifiée avec n la taille de L : $C(n) = C(n-1) + O(n)$ pour $n \geq 1$ (car `min_reste(L)` est linéaire en la taille de la liste. D'où par télescopage $C(n) = O(n^2)$).

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


```
def addition(L):
    i=0
    while L[i]==1:
        L[i]=0
        i+=1
    L[i]=1

def toutes_listes(N):
    L=[0]*N
    T=[L[:]]
    for i in range(2**N-1):
        addition(L)
        T.append(L[:])
    return T
```

Remarque : `addition` modifie `L`, il faut donc faire des copies dans `toutes_listes`.

- Les listes fournies sont dans le même ordre que précédemment. Pour le cas de base ($N = 0$), il y a une liste qui convient, la liste vide.

```
def toutes_listes(N):
    if N==0:
        return [[]]
    else:
        A=toutes_listes(N-1)
        return [x+[0] for x in A]+[x+[1] for x in A]
```

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 ?

Corrigé. La fonction `partition` effectue deux passages sur `L` (une version avec une boucle et `append` est possible et ne fait qu'un seul passage). Pour le tri rapide, si la taille de la liste est au moins deux, on effectue une partition du reste de la liste autour du premier élément.

```
def partition(L,x):
    return [y for y in L if y<=x], [y for y in L if y>x]

def tri_rapide(L):
    if len(L)<=1:
        return L[:]
    else:
        a,b=partition(L[1:],L[0])
        return tri_rapide(a)+L[0]+tri_rapide(b)
```

La complexité est la même que la version du cours : $O(n^2)$ dans le pire cas, $O(n \log n)$ dans le meilleur cas et 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 ?

Corrigé. cf cours!

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.

- 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.

Corrigé.

```
1. def somme(x):
    if type(x)==int:
        return x
    else:
        return sum([somme(y) for y in x])
```

2. Même schéma que la fonction précédente :

```
def copie(x):
    if type(x)==int:
        return x
    else:
        return [copie(y) for y in x]
```

3. L'idée est d'applatir la liste petit à petit : dans `L` sont stockés les entiers. `reste` contient les éléments restants : à chaque étape, on met les éléments de `reste` qui sont des entiers dans `L`, et on concatène les éléments de `reste` qui sont des listes. Le processus s'arrête lorsque `reste` est vide. On démarre avec `reste=[x]` pour ne pas traiter à part le cas où x est un entier.

```
def somme(x):
    L=[]
    reste=[x]
    while reste!=[]:
        r=[]
        for y in reste:
            if type(y)==int:
                L.append(y)
            else:
                r+=y
        reste=r
    return sum(L)
```

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 due 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.

Corrigé.

1. Il y a 52 bits alloués à la mantisse (1 bit de signe, 11 bits d'exposant). Rappel : si $M_1, \dots, M_{52}$ sont les bits de mantisse, s est le bit de signe et $E_0, \dots, E_{10}$ sont les bits d'exposant (décalé), le flottant représenté est (si les bits E_i ne sont ni tous nuls, ni tous égaux à 1) :

$$(-1)^s \times \overline{1.M_1 \dots M_{52}} \times 2^{E-D}$$

avec $E = \overline{E_{10} \dots E_0}^2$ et $D = 2^{10} - 1 = 1023$.

2. $2^{-52} \simeq 2.2 \times 10^{-16}$. On a donc 15 chiffres significatifs décimaux.
3. À priori $h = 1$ n'est pas acceptable car mathématiquement trop grand, par exemple pour $f : x \mapsto \cos(\pi x)$, $f'(1/2) = -\pi$ alors que l'approximation donne $f'(1) \simeq 0$. $h = 10^{-15}$ n'est pas acceptable non plus, car trop petit. En effet, on n'a pas assez de précision. Par exemple :

```
>>> def f(x): return x**2
>>> def deriv(x,h): return (f(x*(1+h))-f(x))/(x*h)
>>> deriv(1,10**-15) #dérivée en 1 de f : 2.
2.220446049250313
```

4. (a) $f(x(1+h)) \underset{h \rightarrow 0}{=} f(x) + xhf'(x) + \frac{x^2h^2}{2}f''(x) + o(h^2)$. D'où $\frac{f(x(1+h))-f(x)}{xh} - f'(x) = \frac{xh}{2}f''(x) + o(h)$
- (b) L'erreur dans l'évaluation de $f(x(1+h)) - f(x)$ est majorée par $\varepsilon(|f(x(1+h))| + |f(x)|) \simeq 2\varepsilon|f(x)|$. D'où une erreur de l'ordre de $\left| \frac{2\varepsilon f(x)}{xh} \right|$ à cause de l'approximation flottante.
- (c) Avec $f : x \mapsto x^2$, on a $f'(x) = 2x$ et $f''(x) = 2$. D'où la somme des deux erreurs égale à $|xh| + \frac{|2\varepsilon x|}{h} = |x|(h + \frac{2\varepsilon}{h})$. Une petite étude de fonction montre qu'il faut prendre $h = \sqrt{2\varepsilon} = 2^{-26} \simeq 1.5 \times 10^{-8}$. Testons :

```
>>> deriv(1,1.5*10**-8)
2.0000000026480316
```

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 $f(x) = 0$?

Corrigé.

- Méthode dichotomique : valable sur $\mathbb{R}$ pour une fonction continue sur $[a, b]$ telle que $f(a)f(b) < 0$, pour trouver un zéro de $[a, b]$. Lente mais fonctionne à tous les coups !
- Méthode de Newton : valable pour une fonction $\mathcal{C}^1$, pour trouver un zéro de f . Fonctionne si on démarre suffisamment près d'un zéro de f tel que $f'(z) \neq 0$, sinon la convergence n'est pas assurée. Rapide, s'étend à des fonctions définies sur $\mathbb{R}^k$ pour $k \geq 2$ (la jacobienne remplace la dérivée).

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.

Corrigé. Lors du déroulement de l'algorithme du pivot, une opération devant produire zéro n'a pas exactement produit zéro à cause des erreurs d'arrondi. Voilà à quoi ressemble la matrice après réduction :

```
array([[ 1.68000000e+02,  1.68000000e+02,  1.68000000e+02],
       [ 0.00000000e+00,  3.00000000e+00,  1.90000000e+01],
       [-2.84217094e-14,  0.00000000e+00,  1.49213975e-13]])
```

On voit les coefficients très petits (mais non nuls) sur la dernière ligne.

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 ?

Corrigé. On introduit x et y les vecteurs de composantes les x_i et y_i . On pose aussi $s_x = \sum_{i=0}^{n-1} x_i$ et $s_y = \sum_{i=0}^{n-1} y_i$

- $\frac{\partial S}{\partial a} = \sum_{i=0}^{n-1} -2x_i(y_i - ax_i - b) = -2\langle x, y \rangle + 2a\|x\|^2 + 2bs_x$ avec $\|x\|^2 = \sum_{i=0}^{n-1} x_i^2$.
- $\frac{\partial S}{\partial b} = \sum_{i=0}^{n-1} -2(y_i - ax_i - b) = -2s_y + 2as_x + 2nb$.

Après résolution, on trouve : $a = \frac{n\langle x, y \rangle - s_x s_y}{n\|x\|^2 - s_x^2}$, $b = \frac{\|x\|^2 s_y - s_x \langle x, y \rangle}{n\|x\|^2 - s_x^2}$. D'où le code Python :

```
def reg_lin(X, Y):
    n=len(X)
    sx=sum(X)
    sy=sum(Y)
    nx2=sum([x**2 for x in X])
    ps=sum([X[i]*Y[i] for i in range(n)])
    return (n*ps-sx*sy)/(n*nx2-sx**2), (nx2*sy-sx*ps)/(n*nx2-sx**2)
```

Remarque : en notant X et Y des variables aléatoires telles que $\mathbb{P}((X, Y) = (x_i, y_i)) = \frac{1}{n}$, on peut écrire :

$$a = \frac{n^2\mathbb{E}(XY) - n^2\mathbb{E}(X)\mathbb{E}(Y)}{n^2\mathbb{E}(X^2) - n^2\mathbb{E}(X)^2} = \frac{\text{Cov}(X, Y)}{\mathbb{V}(X)} \quad \text{et} \quad b = \frac{\mathbb{E}(X^2)\mathbb{E}(Y) - \mathbb{E}(X)\mathbb{E}(XY)}{\mathbb{V}(X)}$$

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 ($0 \rightarrow n - 1$)
`list(range(5))` $\rightarrow$ `[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 $n > \text{len}(u)$, 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])` $\rightarrow$ `3`; `len([[1,2], [3,4]])` $\rightarrow$ `2`
- $u + v$ construit une liste constituée de la concaténation des listes u et v :
`[1, 2] + [3, 4, 5]` $\rightarrow$ `[1, 2, 3, 4, 5]`
- $n * u$ construit une liste constituée de la liste u concaténée n fois avec elle-même :
`3 * [1, 2]` $\rightarrow$ `[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 $O(\text{len}(u))$
`2 in [1, 2, 3]` $\rightarrow$ `True`; `2 not in [1, 2, 3]` $\rightarrow$ `False`

- `u.append(e)` ajoute l'élément `e` à la fin de la liste `u` (similaire¹ 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 $O(\text{len}(u))$
- `u.insert(i, e)` insère l'élément `e` à la position d'indice `i` dans la liste `u` (en décalant les éléments suivants); si $i \geq \text{len}(u)$, `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`

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 $O(\text{a.size})$
- `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

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, en tout cas de manière répétée.

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).

Corrigé. Il y a $n - k + 1$ indices possibles pour le début de la plage d'indices.

```
def modif(t,k):
    n=len(t)
    i=random.randrange(0,n-k+1)
    for j in range(i,i+k):
        t[j]=random.random()
```

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.

Corrigé. Le plus simple est de créer une liste que l'on convertira en tableau Numpy à la fin.

```
def tab_aleatoire(n,c):
    L=[]
    while len(L)<n:
        x,y=random.randrange(0,c),random.randrange(0,c)
        if not [x,y] in L:
            L.append([x,y])
    return np.array(L)
```

Remarques : lorsque vous voulez générer un tableau Numpy, le plus simple est :

- d'utiliser `np.zeros` si la taille est fixée ;
- d'utiliser une liste et la modifier si la taille doit évoluer, puis convertir en tableau Numpy à la fin.

Ici, il est un peu compliqué de tester l'appartenance d'une ligne si on construit le tableau d'un coup (il faudrait l'initialiser avec des valeurs « interdites », comme des -1 partout).

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`.

Corrigé. Cette fonction calcule et trace *l'histogramme* de l'image : on compte pour toutes les valeurs entre 0 (noir) et 255 (blanc) le nombre de pixels de cette intensité.

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 $(x_i^j)_{0 \leq j < n}$).

Corrigé.

```
def VDM(L):
    n=len(L)
    M=np.zeros((n,n))
    for i in range(n):
        M[:,i]=[L[i]**j for j in range(n)]
    return M
```

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

- si toutes les valeurs de $\mathbf{t}$ sont positives ;
- si toutes les valeurs de $\mathbf{t}$ sont entre deux réels a et b fixés ;
- s'il existe une valeur de $\mathbf{t}$ inférieure à $M/2$, où M est le maximum de $\mathbf{t}$.

Corrigé. Les tests suivants conviennent :

```
(t>=0).all()
(t>=a).all() and (t<=b).all()
(t<t.max()/2).any()
```

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 :

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

Corrigé.

```
def double(M):
    A, B = 2*M, 3*M
    U=np.concatenate((M,A))
    V=np.concatenate((A,B))
    return np.concatenate((U,V), axis=1)
```

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 où 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.  ])
```

Corrigé. Remarque : toute l'aide n'a pas été affichée, on a seulement mis les paramètres obligatoires, il y a plein d'autres paramètres optionnels. La fonction f prend en paramètre au moins un argument y (qui peut-être un tableau Numpy), et renvoie M flottants $(f(y))_{1 \leq i \leq M}$. `leastsq` calcule un x vérifiant $\sum_{i=1}^M (f(x)_i)^2 = \min_y \sum_{i=1}^M (f(y)_i)^2$. Pour résoudre le problème de regression linéaire, la fonction f doit prendre en paramètre un tableau Numpy à deux dimensions $((a, b))$, et retourner le N -uplet de valeurs $(ax_i + b - y_i)_{0 \leq i < N}$:

```
def moindre_carres(X):
    def f(t):
        a,b=t
        return [a*x[0]+b-x[1] for x in X]
    return scipy.optimize.leastsq(f,np.array([0,0]))[0]
```

La fonction prend en paramètre un « point de départ », qu'on fixe arbitrairement à $(0, 0)$. Puisque `leastsq` renvoie un couple (solution, covariance), on ne garde que la solution, d'où le `[0]`.