■ ■ ■

Table des matières

I. Organisation des Concours

Lors des concours, l'informatique est évaluée à l'écrit, et à l'oral.

- **À l'écrit.** Des épreuves écrites (sans ordinateurs) sont proposées par tous les concours : CCP (3h), Centrale (3h), Mines (1h30), X-ENS (2h).
- **À l'oral.** La seconde épreuve de mathématiques du concours Centrale combine questions de mathématiques et écriture de programmes. L'environnement de programmation proposé est Pyzo.

I.1 Pyzo

Pyzo est un environnement de programmation libre et multi-plateforme pour Python. Les informations concernant son installation sont disponibles : http://www.pyzo.org/
Ce programme est le seul disponible lors de l'oral du concours de Centrale. Il convient donc de se familiariser avec celui-ci le plus tôt possible.
Lors d'un TP, il est important de :

- Dimensionner la fenêtre de travail pour qu'elle occupe une grande partie de l'écran et d'augmenter la police pour rendre le travail lisible.
- Commencer par **enregistrer** le fichier de travail (sans espaces ou accents dans le nom). Par exemple, choisir `monnom_tp_1.py`.
- **Évaluer** le code tapé sur l'éditeur en utilisant la combinaison `Ctrl + Shift + E`. Ceci permet d'effectuer une réinitialisation de la mémoire lors de chaque évaluation.
- Lors de l'utilisation du **terminal**, les touches ↑ et ↓ permettent de naviguer dans les commandes déjà évaluées (et d'éviter ainsi des saisies clavier laborieuses).
- D'inclure des **tests**, en nombre suffisant, et **insérés** dans le code.

I.2 Liens utiles

- La documentation Python : https://docs.python.org/fr/3/
- La documentation Numpy et Scipy : https://docs.scipy.org/doc/
- La documentation disponible aux oraux de Centrale : https://www.concours-centrale-supelec.fr/CentraleSupelec/SujetsOral/PSI
- Pour visualiser l'utilisation de l'espace mémoire par Python : http://www.pythontutor.com/visualize.html#mode=edit
- Des problèmes d'algorithmique parfois avancés : https://projecteuler.net/
- La documentation des requêtes SQL : http://sql.sh
- Manipuler en ligne les bases de données à l'adresse : https://sqliteonline.com

II. Conseils

II.1 Présentation des écrits

- Les commentaires doivent précéder les programmes. Ces derniers peuvent être commentés (♯).
- Utiliser une barre verticale pour matérialiser les indentations.

II.2 Des beugs ?

`SyntaxError` Le signe `=` est un symbole d'affectation et il n'est pas symétrique. Le nom de l'objet est à gauche, son contenu à droite.

`IndexError` Les bornes de la fonction `range`. L'appel `range(a, b, pas)` crée un itérateur qui parcourt les entiers qui s'écrivent sous la forme $\texttt{a} \leqslant \texttt{a} + k \cdot \texttt{pas} < \texttt{b}$.
Les indices des éléments de la liste `liste` vont de `0` à `len(liste) - 1`.

`TypeError` L'appel à la fonction `f` s'effectue via `f(2)`. L'accès aux indices de la liste (ou chaîne de caractères) `liste` via `liste[2]`. Les erreurs de type surviennent également si les opérandes ne sont pas correctes, par exemple `2 + [3]`.

- L'indentation de `return` dans une boucle ou conditionnelle.
- L'oubli de `return` dans une fonction. Le résultat de l'évaluation de cette fonction est alors `None`.
- Lors de la manipulation des listes, les copies avec alias. Une copie sans alias peut s'effectuer via `n_liste = liste.copy()` ou `n_liste = liste[:]`.
- La fonction `print` permet d'afficher des informations dans l'interpréteur. Il est judicieux de l'utiliser pour **débeuguer** des programmes en demandant par exemple d'afficher les valeurs successives associées à une variable donnée.

III. Complexité

III.1 Recommandations

- Pour ajouter un élément à une liste, la méthode `liste.append(element)` est en temps constant, contrairement à `liste = liste + [element]` qui s'effectue en temps linéaire (voir Partie VIII.2).
- Un stockage vaut mieux que plusieurs appels de fonction !
- On peut parfois préférer les boucles conditionnelles `while` aux boucles `for` pour minimiser le nombre d'itérations. En Python, l'instruction `return` interrompt la fonction et permet d'obtenir un comportement analogue.

III.2 Classes de complexité

Si T_n est le nombre d'opérations (la nature des opérations dépend du contexte : multiplications, additions, comparaisons, appels de fonctions,...) effectuées pour un argument de taille n et a_n est le terme général d'une suite,

$$T_n = O(a_n) \Leftrightarrow \exists\, (M, n_0) \in \mathbb{R}_+^\star \times \mathbb{N}\ ;\ \forall\, n \geqslant n_0,\ T_n \leqslant M a_n.$$
$$T_n = \Theta(a_n) \Leftrightarrow T_n = O(a_n) \text{ et } a_n = O(T_n).$$

- $\Theta(1)$: en *temps constant*.
  Ex. : Échange de deux valeurs.
- $\Theta(\ln n)$: *logarithmique*.
  Ex. : Recherche dichotomique.
- $\Theta(n)$: *linéaire*.
  Ex. : Recherche du maximum.
- $\Theta(n \ln n)$: *quasi-linéaire*.
  Ex. : Tri fusion.
- $\Theta(n^2)$: *quadratique*.
  Ex. : Tri par insertion.
- $\Theta(n^\beta)$: *polynomiale* ($\beta > 1$).
  Ex. : Multiplication matricielle.
- $\Theta(a^n)$: *exponentielle*.
  Ex. : Tours de Hanoï.

III.3 Complexité sur les listes

Lors de la création d'une liste en Python, une *grande* place mémoire est allouée. Certaines opérations sont alors relativement efficaces, tant que cette place n'est pas toute utilisée... Les primitives Python sur les listes ont les complexités suivantes (source : https://wiki.python.org/moin/TimeComplexity) :

length	O(1)	Parcours	O(n)
Lecture d'un item	O(1)	Copie	O(n)
Modification d'un item	O(1)	Insertion d'un élément	O(n)
append(1)	O(1)	Suppression d'un élément	O(n)

(1) Complexité en moyenne, l'opération peut être parfois beaucoup plus complexe.

III.4 Exemples de calculs de complexité

```
def horner(P,x) :
    n = len(P)
    Px = P[n-1]
    for i in range(1,n) :
        Px = Px * x + P[n-i-1]
    return Px
```

À chaque passage dans la boucle :
- 1 multiplication,
- 1 addition.

Le nombre d'opérations vaut alors

$$\sum_{i=1}^{\mathtt{n}-1} 2 = 2(\mathtt{n}-1)$$

```
def fibo(n):
    if n==0 or n==1 :
        return 1
    else:
        return fibo(n-1) + fibo(n-2)
```

En notant T_n le nombre d'appels pour évaluer `fibo(n)`,

$$T_0 = 1$$
$$T_1 = 1$$
$$T_n = T_{n-1} + T_{n-2} + 1$$

On montre alors que $\frac{1+T_n}{2}$ est le $n$ème nombre de Fibonacci.

IV. Preuves de programmes : 3 exemples

IV.1 Programme itératif

L'invariant doit être vrai en début de boucle, maintenu par le corps de boucle et sa validité en sortie de boucle doit assurer la correction de la fonction.

```
def horner(P,x) :
    n = len(P) - 1
    Px = P[n]
    for i in range(1,n+1) :
        Px = Px * x + P[n-i]
    return Px
```

On note `P = [`$a_0, \dots, a_n$`]`. L'invariant de boucle : En entrée de l'itération `i`, `Px` contient $\sum_{k=0}^{\mathtt{i}-1} a_{n-k}\mathtt{x}^{\mathtt{i}-1-k}$.

- **Précondition.** Avant l'entrée en boucle, `Px` contient $a_n = \sum_{k=0}^{0} a_{n-k}\mathtt{x}^{0-k}$.
- **Hérédité.** Supposons qu'à l'entrée de la `i`-ème itéation, `Px` contienne $\sum_{k=0}^{\mathtt{i}-1} a_{n-k}\mathtt{x}^{\mathtt{i}-1-k}$.

  Alors, après le corps d'instructions, `Px` contient $\mathtt{x}\cdot\sum_{k=0}^{\mathtt{i}-1} a_{n-k}\mathtt{x}^{\mathtt{i}-1-k} + a_{n-1-\mathtt{i}-1} = \sum_{k=0}^{\mathtt{i}} a_{n-k}\mathtt{x}^{\mathtt{i}-k}$..
- **Postcondition.** À la fin de la boucle, `i` contient n et `Px` contient $\sum_{k=0}^{n} a_{n-k}\mathtt{x}^{n-k} = \sum_{k=0}^{n} a_k\mathtt{x}^k$.

Programme avec boucle conditionnelle

```
def racine_entiere(n) :
    m = 0
    while (m+1) * (m+1) <= n :
        m = m + 1
    return m
```

Correction. Invariant I : $\mathtt{n} - \mathtt{m}^2 \geqslant 0$.
- Avant l'entrée dans la boucle, $\mathtt{n}-\mathtt{m}^2 = \mathtt{n} \geqslant 0$.
- Supposons que $\mathtt{n} - (\mathtt{m}+1)^2 \geqslant 0$. Dans le corps de boucle, `m` est incrémenté, donc en sortie $\mathtt{n} - \mathtt{m}^2 \geqslant 0$.
- En sortie, $\mathtt{n}-\mathtt{m}^2 \geqslant 0$ et $\mathtt{n}-(\mathtt{m}+1)^2 < 0$, soit

$$\mathtt{m} = \lfloor \sqrt{n} \rfloor .$$

Terminaison. Variant V : $\mathtt{n} - \mathtt{m}^2 \geqslant 0$.
- V est à valeurs entières.
- V décroît de 1 à chaque passage dans la boucle.
- Le programme termine.

Programme récursif

```
def f_47(n, p):
    if n = 0:
        return 47
    else:
        return f_47(n-1, f_47(n-1, p+7))
```

On montre par récurrence sur `n` que pour tout `p` entier, `f_47(n, p)` termine et renvoie 47.
Initialisation. Si $\mathtt{n} = 0$, alors pour tout `p`, `f_47(n, p)` renvoie 47.
Hérédité. Soit $\mathtt{n} \in \mathbb{N}^*$. On suppose que la propriété est vraie pour tout entier inférieur ou égal à `n`. Soit `p` un entier. Comme $\mathtt{n}-1 < \mathtt{n}$, alors `f_47(n-1, p+7)` renvoie 47. Ainsi, `f_47(n-1, 47)` termine et renvoie 47. Finalement, `f_47(n, p)` renvoie 47.

V. Types

V.1 Booléens

- Contient uniquement les éléments `True` et `False`.
- À ne pas confondre avec les chaînes de caractères `"True"` et `"False"`.
- Si `b` est un booléen, le test `b == True` peut être remplacé par `b` et le test `b == False` par `not b`.

V.2 Nombres

En Python, le type `int` permet d'utiliser des entiers arbitrairement grands.

- En général, dans la représentation par complément à 2, si les entiers sont stockés sur n bits, on peut représenter les entiers de $[\![-2^{n-1}, 2^{n-1}-1]\!]$ par :
  - Si $-2^{n-1} \leqslant x < 0$, il est représenté par $\widetilde{x} = x + 2^n$.
  - Si $0 \leqslant x < 2^{n-1}$, il est représenté par $\widetilde{x} = x$.

  Les entiers plus grands ou plus petits sont translatés dans cet intervalle.

En Python, le type `float` permet d'utiliser des nombres flottants.

- Dans le standard IEE-754, un nombre $x = \varepsilon \cdot 1.m_1 \cdots m_{p-1} \cdot 2^e$ à *virgule flottante* est représenté par :
  - son signe ε codé par 0 si le nombre est positif, 1 s'il est négatif,
  - sa mantisse à $p-1$ composantes telle que $m_0 = 1$ (non stocké),
  - un exposant non signé mais *biaisé* pour faciliter la comparaison entre flottants,
  - un nombre de bits de stockage fixé à 32 bits (*simple précision*) ou 64 bits (*double précision*) :

	signe	exposant	mantisse	biais d'exposant
32 bits	1 bit	8 bits	23 bits	$2^7 - 1 = 127$
64 bits	1 bit	11 bits	52 bits	$2^{10} - 1 = 1023$

- Les phénomènes pouvant être rencontrés : dépassement de capacité, échec de l'égalité à 0, présence de chiffres non significatifs à l'issue d'une soustraction,...

V.3 Listes, Tuples, Chaînes de caractères

- Les listes sont délimitées par des crochets, les tuples par des parenthèses, les chaînes de caractères par des guillemets.
- On peut accéder aux contenus des listes, tuples et chaînes de caractères en spécifiant l'indice concerné : `truc[i]`.
- Les listes sont des objets mutables : l'opération `liste[0] = 3` permet de modifier le contenu de `liste[0]` et de le remplacer par la valeur `3`.
- Les tuples et chaînes de caractères ne sont pas mutables. La seule façon de les modifier est de les écraser avec une nouvelle valeur, par exemple : `chaine = "he", chaine = chaine + "llo"`.
- Les dictionnaires sont des structures de données intéressantes !

V.4 Piles

Les piles sont des structures de données sur lesquelles il n'est permis que 3 opérations :

- Tester si la pile est vide.
- `push` : Ajouter un élément sur le dessus de la pile.
- `pop` : Supprimer et renvoyer l'élément sur le dessus de la pile.

Les piles peuvent être implémentées grâce à des listes.

VI. Programmes au Programme

VI.1 Autour des itérables

Recherche dans une liste

```
def est_present(liste, el):
    trouve, i = False, 0
    while not(trouve) and i < len(liste):
        if liste[i] == el:
            trouve = True
        i += 1
    return trouve
```

Remarque.
- Complexité linéaire dans le pire des cas.

Recherche du maximum

```
def maximum(liste):
    m, im = liste[0], 0
    for i in range(1,len(liste)):
        if liste[i] > m:
            m = liste[i]
            im = i
    return (m, im)
```

Remarques.
- Complexité linéaire.
- Prendre garde aux indices du range.

Exercice 1.

1. Modifier la fonction `maximum` pour qu'elle retourne uniquement le maximum.

2. Écrire une fonction `minimum`.

Remarque.
Les fonctions `index`, `max` et `min` sont implémentées en Python.

Moyenne

```
def moyenne(liste):
    s = 0
    for el in liste:
        s += el
    return s / len(liste)
```

Variance

```
def variance(liste):
    ecarts = 0
    moy = moyenne(liste)
    for el in liste:
        ecarts += (el - moy)**2
    return ecarts / len(liste)
```

Remarques.
- La division n'est effectuée qu'en dernier pour minimiser les erreurs de calculs.
- Les variables sont initialisées **avant** l'entrée dans la boucle.

Recherche d'un mot dans une chaîne de caractères

```
def est_present(chaine, mot):
    trouve, i = False, 0
    while not(trouve) and i < len(chaine) - len(mot) + 1:
        commune = (chaine[i] == mot[0])
        j = 1
        while commune and j < len(mot) and i + j < len(chaine):
            commune = (chaine[i+j] == mot[j])
            j += 1
        if j == len(mot) and commune:
            trouve = True
        i += 1
    return trouve
```

Remarques.
- Complexité en $O(m \cdot n)$ où m est la taille du mot et n celle de la chaîne.
- En utilisant les fonctionnalités de Python, la seconde boucle `while` pourrait être écrite : `trouve = chaine[i:i+m] == mot`.
- Il existe des algorithmes plus efficaces en moyenne, par exemple Rabin-Karp (H.P.).

Recherche dichotomique dans une liste triée

```
def recherche_dicho(liste, el):
    def recloc(liste, el, g, d):
        n = len(liste)
        m = (g + d) // 2
        if d < g:
            return False
        elif liste[m] == el:
            return True
        elif liste[m] < el:
            return recloc(liste,el,m+1,d)
        else:
            return recloc(liste,el,g,m-1)
    return recloc(liste,el,0,n-1)
```

```
def recherche_dicho(liste, el):
    g, d = 0, len(liste) - 1
    while g <= d:
        m = (g + d) // 2
        if el < liste[m]:
            d = m - 1
        elif el > liste[m]:
            g = m + 1
        else:
            return True

    return False
```

Remarques.
- Prendre garde au critère d'arrêt.
- La présence des indices `m-1` et `m+1` dans l'appel récursif assurent la stricte décroissance de la longueur de la liste passée en argument.

VI.2 Résolution d'équations

Dichotomie

```
def dichotomie_defaut(f, a, b, eps):
    while (b - a) > eps:
        c = (a + b) / 2
        if f(a) * f(c) <= 0:
            a, b = a, c
        else:
            a, b = c, b
    return a
```

Remarques.
- Une condition suffisante de terminaison est fournie pas le théorème des valeurs intermédiaires.
- Le critère d'arrêt est précis bien que parfois surestimé.
- Ce programme peut être écrit en récursif.
- La méthode est linéaire, i.e. le nombre de décimales exactes augmente d'une unité à chaque itération.

Newton

```
def newton(f, df, x0, eps):
    x1 = x0 - f(x0) / df(x0)
    while abs(x1 - x0) > eps:
        x0, x1 = x1, x1 - f(x1) / df(x1)
    return x0
```

Remarques.
- Interprétation géométrique : Prendre la tangente !
- Le calcul nécessite la connaissance de la dérivée (ou on code une dérivée numérique).
- Le critère d'arrêt est peu fiable si la fonction varie beaucoup près du zéro.
- Le critère d'arrêt peut être remplacé par `abs(f(x1) - f(x0)) > eps` mais reste peu fiable si la fonction varie peu autour du zéro.
- La méthode converge vers un zéro de `f`, mais il est difficile à localiser...
- Pour qu'elle converge, la méthode nécessite des hypothèses sur `f`...
- La convergence est quadratique, i.e. le nombre de décimales exactes double à chaque itération.

VI.3 Intégration numérique

Méthode des rectangles à gauche

```
def rectangles_gauche(f, a, b, n):
    pas, hauteur, x = (b - a) / n, 0, a
    for i in range(n):
        hauteur += f(x)
        x += pas
    integrale = pas * hauteur
    return integrale
```

Remarques.
- Erreur en $O\left(\frac{1}{n}\right)$ pour les fonctions de classe $\mathscr{C}^1$.
- Erreur atteinte pour des fonctions affines.
- Méthode exacte pour les fonctions constantes.
- Voir la méthode des rectangles à droite.
- La méthode des rectangles médians (H.P.) est plus efficace.

Méthode des trapèzes

```
def trapezes(f, a, b, n):
    pas = (b - a) / n
    hauteur = 0
    x, y = a, a + pas
    for i in range(n):
        hauteur += (f(x)+f(y))/2
        x, y = x + pas, y + pas
    integrale = pas * hauteur
    return integrale
```

Remarques.
- Erreur en $O\left(\frac{1}{n^2}\right)$ pour les fonctions de classe $\mathscr{C}^2$.
- Erreur atteinte pour des fonctions polynomiales de degré 2.
- Méthode exacte pour les fonctions affines.

Exercice 2.

1. Modifier ces fonctions pour que le quatrième paramètre soit le pas et non le nombre d'intervalles.

2. Implémenter la méthode des rectangles médians.

3. Utiliser ces méthodes sur des échantillonnages de vitesses pour déterminer les distances parcourues.

Méthode d'Euler explicite

On considère l'équation différentielle $y'(t) = f(t, y(t))$.

```
def euler_explicite(f, t0, t1, y0, pas):
    y, t = [y0], [t0]
    while t[-1] < t1:
        y.append(y[-1] +
                 pas * f(t[-1], y[-1]))
        t.append(t[-1]+pas)
    return t, y
```

Remarques.
- Interprétation géométrique : Prendre la tangente !
- `y[-1]` permet de retourner le dernier élément d'une liste.
- L'erreur est en $O\left(\frac{1}{n}\right)$.
- La fonction `f` doit être suffisamment régulière pour qu'il y ait existence et unicité de la solution de l'équation différentielle.

Exercice 3.

1. Modifier la fonction précédente pour pouvoir résoudre de manière approchée...

a) des équations où y et f sont à valeurs dans $\mathbb{R}^n$.

b) les équations différentielles d'ordre 2.

2. Réécrire le programme précédent en le comprenant !

VI.4 Résolution de systèmes linéaires

La stratégie est la suivante : **1.** Triangulation **2.** Élimination.

Système triangulaire

Si le système est de la forme $Ux = b$, où U est une matrice supérieure, il est résolu par élimination en commençant par les dernières composantes.

```
import numpy as np

def solve_upper(U, b):
    nl, nc = np.shape(U)
    x = np.zeros((nl,1))
    x[nl-1] = b[nl-1] / U[nl-1,nl-1]
    for i in range(nl-2, -1, -1) :
        s = 0
        for k in range(i+1,nl) :
            s = s + U[i, k] * x[k]
        x[i] = (b[i] - s)/U[i, i]
    return x
```

Coût.

$$1+\sum_{i=1}^{n-1}(\underbrace{n-i}_{\text{produits}} + \underbrace{n-i-1}_{\text{somme}} + \underbrace{1}_{\text{soustraction}} + \underbrace{1}_{\text{division}}) = n^2.$$

Pivot de Gauss

La fonction `echange_lignes` échange les lignes d'un tableau sans créer d'alias. La fonction `op` permet d'effectuer une opération élémentaire sur les lignes d'une matrice.

```
def echange_lignes(A, i, j):
    tmp = np.copy(A[i,:])
    A[i,:] = np.copy(A[j,:])
    A[j,:] = np.copy(tmp)

def op(A, i, j, pivot):
    nl, nc = np.shape(A)
    for k in range(i,nc):
        A[j,k] = A[j,k] - A[i,k] * pivot
```

La résolution par pivot de Gauss consiste à chercher un pivot non nul, puis à éliminer les composantes pour obtenir une matrice triangulaire.

```
def resolution(A, b):
    nl, nc = np.shape(A)
    for i in range(nl):
        j = i
        while A[j,i] == 0:
            j = j + 1
        p = A[j,i]
        echange_lignes(A,i,j)
        echange_lignes(b,i,j)
        for j in range(i+1,nl):
            ag = A[j,i]/p
            op(A, i, j, ag)
            b[j] = b[j] - b[i] * ag
    x = solve_upper(A, b)
    return x
```

Coût maximal de recherche du pivot.

$$\sum_{i=0}^{n-1}(n-i-1) = \frac{(n-1)n}{2}.$$

Coût des opérations élémentaires.

$$\sum_{k=0}^{n-1}\sum_{i=k+1}^{n}\left(1+\sum_{j=k+1}^{n}2\right) = \sum_{k=1}^{n}(k+2k^2) \sim \frac{2}{3}n^3.$$

Coût de l'élimination.

$$n^2.$$

Coût total.

$$\sim \frac{2}{3}n^3.$$

Remarques.

- Les opérations effectuées sur A sont à effectuer sur le second membre b.
- La méthode du pivot partiel consiste à chercher le pivot maximal parmi les lignes restantes. Il permet de minimiser les erreurs de calculs mais augmente la complexité moyenne de l'algorithme.

VI.5 Tris

Une fois la liste triée, il est très rapide d'en déterminer le maximum, le minimum et la médiane. On peut montrer qu'il n'existe pas d'algorithme de tri, basé uniquement sur des comparaisons, qui soit de complexité dans le pire des cas meilleure que $\Theta(n\ln(n))$.

Tri par Insertion

On insère successivement, à leur place, les éléments de la liste.

```
def tri_partiel(liste, i):
    j = i
    while j>0 and liste[j]<liste[j-1]:
        tmp = liste[j]
        liste[j] = liste[j-1]
        liste[j-1] = tmp
        j = j - 1
```

```
def tri_insertion(liste):
    for i in range(1,len(liste)):
        tri_partiel(liste, i)
```

Remarque.
- Complexité linéaire dans le meilleur et quadratique le pire des cas.
- Cet algorithme modifie la liste passée en argument en utilisant le caractère mutable des listes.

Tri Fusion

On trie indépendamment les première et seconde moitiés de la liste puis on les fusionne en temps linéaire en la taille des listes.

```
def fusion(liste, g, d):
  aux = []
  m = (g + d)//2
  # Parcourt des deux listes
  i1, i2  = g, m
  # les elements d'indices g,..,i1-1 sont
     # deja fusionnes
  # les elements d'indices m,..,i2-1 sont
     # deja fusionnes
  while i1 < m and i2 < d:
    if liste[i1] <= liste[i2]:
      aux.append(liste[i1])
      i1 += 1
    else :
      aux.append(liste[i2])
      i2 += 1
  # Une des listes a ete videe
  if i1 == m:
    aux.extend(liste[i2:d])
  elif i2 == d:
    aux.extend(liste[i1:m])
  for i in range(len(aux)):
    liste[g+i] = aux[i]
```

```
def tri_fusion(liste):
    def aux(liste, g, d):
        if d - g > 1:
            m = (g+d)//2
            aux(liste, g, m)
            aux(liste, m, d)
            fusion(liste, g, d)
    aux(liste, 0, len(liste))
```

Remarques.
- Il est essentiel que la fusion s'effectue en temps linéaire.
- Le tri a été programmé de manière récursive.

Tri Rapide

Un pivot est choisi (ici le premier élément de la liste). On sépare (en temps linéaire) les éléments de la liste qui sont inférieurs au pivot de ceux qui lui sont supérieurs. On continue récursivement. On propose ci-dessous deux implémentations (cf. remarque suivante).

```
def tri_rapide(liste):
    if len(liste) < 1 :
        return liste
    else:
        pivot = liste[0]
        petits, egaux, grands = [],[],[]
        for i in range(0,len(liste)):
            if liste[i] < pivot:
                petits.append(liste[i])
            elif liste[i] > pivot:
                grands.append(liste[i])
            else:
                egaux.append(liste[i])
        return tri_rapide(petits) +\
                egaux +\
                tri_rapide(grands)
```

```
def tri_local(liste, g, d):
    if d - g <= 1:
        return None
    pivot = liste[g]
    g1, i, d1 = g, g, d-1
    while i <= d1:
        if liste[i] > pivot:
            echange(liste, i, d1)
            d1 = d1 - 1
        elif liste[i] == pivot:
            i += 1
        elif liste[i] < pivot:
            echange(liste, g1, i)
            g1 = g1 + 1
            i = i + 1
    tri_local(liste, g, g1)
    tri_local(liste, d1+1, d)

def tri_rapide(liste):
    tri_local(liste, 0, len(liste))
```

Remarques.
■ Complexité en $\Theta(n\ln(n))$ en moyenne, quadratique dans le pire des cas.
■ Le choix du pivot peut être rendu aléatoire pour éviter le pire des cas.
■ Le code de gauche est plus facile à écrire mais utilise des listes auxiliaires. Le code de droite modifie les listes en place et permet d'atteindre la complexité annoncée.
Il repose sur l'invariant suivant :

- Les éléments d'indice g,...,g1-1 sont strictement inférieurs au pivot.
- Les éléments d'indice g1,...,i sont égaux au pivot.
- Les éléments d'indice d1+1,...,d-1 sont strictement supérieurs au pivot.

VII. Bases de données

VII.1 Syntaxe

La syntaxe d'une requête sur une base de données est de la forme suivante

```
SELECT <liste d attributs>
FROM <liste de tables>
WHERE <conditions>
GROUP BY <liste d attributs>
HAVING <conditions>
ORDER BY <liste d attributs>
LIMIT <entier1> OFFSET <entier2>
```

où

- `SELECT` désigne les attributs (colonnes) qui seront affichés en sortie ;
  Variante : `SELECT DISTINCT` n'affiche que des lignes distinctes.
- `FROM` désigne les tables dans lesquelles sont sélectionnés les attributs ;
  Jointure naturelle : `<tab1> JOIN <tab2> ON <tab1.att1> = <tab2.att2>`
- `WHERE` désigne les conditions que doivent satisfaire les n-uplets affichés ;
  Comparateurs : `<>`, `=`, `<`, `>`, `<=`, `>=`, `BETWEEN`, `AND`, `OR`, `IS NULL`, `IS NOT NULL`, `IN`, `NOT IN`, `LIKE`.
- `GROUP BY` désigne les regroupements effectués sur ces n-uplets ;
- `HAVING` désigne des conditions nécessitant des fonctions d'agrégation.
- `ORDER BY` désigne la manière dont sont triés les résultats.
  Paramètres : `<att1> ASC` ou `<att1> DESC`.
- `LIMIT <entier1> OFFSET <entier2>` affiche uniquement `<entier1>` lignes en n'affichant pas les `<entier2>` premières.

Plutôt qu'un attribut, on peut afficher une de ses statistiques via une des fonctions d'agrégations suivantes : `MAX`, `MIN`, `SUM`, `AVG` et `COUNT`. Pour cette dernière, `COUNT(*)` permet de compter le nombre de lignes dans une table.
Enfin, étant données deux tables, on peut effectuer leur réunion (`UNION`), leur intersection (`INTERSECT`) ou la première exclue des entrées de la seconde (`EXCEPT`).

Plus d'informations à l'adresse : http://sql.sh.
Tester ses requêtes en ligne à l'adresse : https://sqliteonline.com.

VII.2 Méthodologie

(i) **Sélection simple.**

(a) Identifier les attributs qui doivent être affichés et les tables les contenant.

(b) Indiquer ces tables après `FROM` avec éventuellement un alias (`AS`).

(c) Indiquer les attributs à afficher après `SELECT`, préfixés du nom de la table ; ajouter éventuellement un alias à l'attribut.

(d) Identifier les fonctions à calculer, les attributs nécessaires pour leur calcul et ajouter éventuellement les tables après `FROM` puis la fonction après `SELECT`.

(ii) **Groupes & Restrictions.**

(a) Identifier les attributs utilisés pour le partitionnement des lignes et les indiquer après `GROUP BY`.

(b) Compléter si nécessaire les tables après `FROM`

(c) Compléter `SELECT` avec les attributs utilisés pour le partitionnement.

VII.3 Algèbre relationnelle

Exemples pris sur une base de données d'une bibliothèque constituée des tables
■ `livres` d'attributs `titre`, `auteur`, `parution`, `isbn`, `nombre` ;
■ `emprunts` d'attributs `isbn`, `emprunteur`, `date` ;
■ `abonnes` d'attributs `nom`, `prenom`, `naissance`, `id`, `contact`.
La clé primaire d'une table est un attribut qui identifie de manière unique un $n$uplet (Ex. l'attribut `isbn` dans la table `livres`).

- **Sélection.** σ : Supprime des n-uplets (i.e. des lignes dans la relation).

  Livres dont l'auteur est Agnesi

  $$\sigma_{auteur=\text{"Agnesi"}}(livres)$$

```
SELECT *
FROM livres
WHERE auteur="Agnesi"
```

- **Projection.** π : Supprime des attributs (i.e. des colonnes dans la relation).

  Liste des ISBN des livres dont l'auteur est Agnesi

  $$\pi_{isbn}(\sigma_{auteur=\text{"agnesi"}}(livres))$$

```
SELECT isbn
FROM livres
WHERE auteur="Agnesi"
```

- **Groupements & Agrégats.** γ : Regroupe des n-uplets selon la valeur de leur attribut et calcule une fonction.

  Auteur et première année de parution par auteur.

```
SELECT auteur, MIN(parution)
FROM livres
GROUP BY auteur
```

  $$\pi_{auteur,MIN(parution)}({}_{auteur}\gamma_{MIN(parution)}(livres))$$

- **Renommage.** ρ : Change le nom d'un attribut

  Auteur et première année de parution par auteur.

```
SELECT auteur, MIN(parution) AS m
FROM livres
GROUP BY auteur
```

  $$\pi_{auteur,m}(\rho_{MIN(parution)\to m}({}_{auteur}\gamma_{MIN(parution)}(livres)))$$

- **Jointure.** $\bowtie$: Combine des relations selon des attributs communs.

  ISBN des livres d'Agnesi empruntés.

  $$\pi_{isbn}(\sigma_{auteur=\text{"agnesi"}}(livres \bowtie emprunts))$$

```
SELECT livres.isbn
FROM livres
JOIN emprunts
ON livres.isbn=emprunts.isbn
WHERE auteur="Agnesi"
```

- **Produit cartésien.** $\times$: Équivalent à une jointure naturelle sur des tables qui n'ont aucun attribut en commun.

  Ensemble des emprunts possibles.

  $$livres \times abonnes$$

```
SELECT *
FROM livres
JOIN abonnes
```

VIII. F.A.Q.

VIII.1 `range`, `arange` et `linspace` ?

- `range` est une primitive du langage et l'appel `range(a, b, p)` crée un itérateur sur les entiers de la forme $(a + kp)_{k=0,1,...}$ de l'intervalle $[a, b[$. Le pas `p` est un entier (éventuellement négatif).
  À utiliser quand on ne travaille qu'avec des entiers.
- `arange` et `linspace` sont des fonctions disponibles dans le module `numpy`.
  L'appel `numpy.arange(a, b, p)` crée un tableau `numpy` dont les éléments sont les flottants $(a + kp)_{k=0,1,...}$ de l'intervalle $[a, b[$. Le pas `p` est un entier (éventuellement négatif).
  À utiliser avec des flottants, quand on connaît le pas entre deux éléments consécutifs.
  L'appel `numpy.linspace(a, b, num)` crée un tableau `numpy` constitué de `num` éléments équirépartis de l'intervalle $[a, b]$. *À utiliser avec des flottants, quand on connaît le nombre de points.*

VIII.2 Ajouter un élément en fin de liste ?

- La méthode `append`. Sa complexité amortie est en temps constant et elle modifie la liste en place.

```
>>> L = [1, 2, 3]
>>> id(L)
140150518994624
>>> L.append(4)
>>> L
[1, 2, 3, 4]
>>> id(L)
140150518994624
```

  À utiliser SANS *modération !*
- La syntaxe `+=`. Elle a les mêmes caractéristiques que la méthode `append` mais sa signature est différente.

```
>>> L = [1, 2, 3]
>>> id(L)
140150518994112
>>> L += [4]
>>> L
[1, 2, 3, 4]
>>> id(L)
140150518994112
```

  Peut être utilisée, mais la méthode .append est plus orientée objet...
- La concaténation `+`. La liste est copiée dans un nouvel emplacement mémoire, en ajoutant un dernier élément. Sa complexité est linéaire.

```
>>> L = [1, 2, 3]
>>> id(L)
140150518994624
>>> L = L + [4]
>>> L
[1, 2, 3, 4]
>>> id(L)
140150518994048
```

  Cette stratégie est à éviter !

VIII.3 Tracer un graphique en 5 lignes ?

Le module `import matplotlib.pyplot as plt` doit être chargé.

- Créer la figure via `plt.figure()`.
- Commencer par créer le vecteur `X` des abscisses en utilisant `range`, `arange` ou `linspace` par exemple.
- Coninuer en créant le vecteur `Y` des ordonnées. Si on trace le graphe de $x \mapsto f(x)$, on pourra utiliser une définition par compréhension de la liste : `Y = [f(x) for x in X]`.
- Construire le graphique en utilisant `plt.plot(X, Y)`.

- Afficher le graphique à l'aide de `plt.plot()`. On pourra remplacer cette ligne par `plt.savefig("monfichier.png")` pour sauvegarder le graphique.

Pour améliorer le graphique en insérant une légende, on pourra utiliser

- `plt.plot(X, Y, label = "f")`
- `plt.legend()` avant l'affichage.

VIII.4 Importer un module ?

3 options s'offrent à vous.

- On importe tout le contenu du module.

```
from math import *
```

Il faut ensuite être très prudents !

```
>>> pi
3.141592653589793
>>> pi = 4
>>> pi
4
```

Cette stratégie est à éviter !

- On importe toutes les fonctions sans charger toutes ses fonctionnalités. Il faut ensuite faire référence au module pour aller chercher ses éléments.

```
import math
```

On peut alors faire coexister différents noms de variables.

```
>>> math.pi
3.141592653589793
>>> pi = 4
>>> pi, math.pi
(4, 3.141592653589793)
```

Stratégie à utiliser tant que le nom du module est court !

- On importe le module avec un alias.

```
import math as m
```

On peut alors faire coexister différents noms de variables.

```
>>> m.pi
3.141592653589793
```

Stratégie à utiliser !

VIII.5 Comment calculer les coefficients binomiaux ?

Commençons par les fausses bonnes idées.

- Programmation par récurrence, en utilisant la formule du triangle de Pascal.

```
def b(n, p):
  if p < 0 or p > n:
    return 0
  elif p == 0 or p == n:
    return 1
  else:
    return (b(n-1, p) + b(n-1, p-1))
```

En notant $T_{n,p}$ le nombre d'appels nécessaires pour évaluer `b(n, p)`, on montre que $T_{n,p}$ satisfait $T_{n,0} = 1$ et $T_{n,p} = T_{n-1,p} + T_{n-1,p-1}$. Ainsi, $T_{n,p} = \binom{n}{p}$ et, en particulier, d'après la formule de Stirling, $T_{2n,n} \sim \frac{4^n}{\sqrt{\pi n}}$.

- Avec la forme factorielle.

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

  def b(n, p):
  if p < 0 or p > n:
    return 0
  else :
    return fact(n) // (fact(p) * fact(n-p))
```

On remarquera l'utilisation d'une division (euclidienne) entière pour obtenir un entier. La complexité est linéaire en nombre de multiplications. Cependant, les nombres en jeu sont très grands. En Python, les entiers sont arbitrairement grands. Dans d'autres langages, ce n'est pas le cas et cette stratégie n'est pas du tout adaptée.

- Avec la formule du capitaine.

```
def b(n, p):
  if p < 0 or p > n:
    return 0
  elif p == 0 or p == n:
    return 1
  else:
    (n * b(n-1, p-1)) // p
```

Ici, la complexité est linéaire en $\min\{n, p\}$. De plus, les nombres calculés sont aussi petits que possible. On notera également l'usage d'une division entière.

- Si on doit stocker tous les coefficients binomiaux, on utilisera la formule du triangle de Pascal.

```
def b_list(n):
  L = [[1]]
  for i in range(1, n+1)
    M = [[1]]
    for k in range(1,i):
      M.append(L[-1][i] + L[-1][i-1]])
    L.append(M)
  return L
```

Ici, la complexité est linéaire. On a cependant utilisé une complexité spatiale quadratique.

VIII.6 Calculer une intégrale à paramètre ?

Le module `import scipy.integrate as integr` doit être chargé. On cherche à évaluer $I_n = \displaystyle\int_a^b f_n(t)\mathrm{d}t$.

- Commencer par créer la fonction. Par exemple,

```
def f(n, x):
  return n * x
```

- Redéfinir la fonction d'un seul paramètre.

```
def I(n):
  def g(t):
    return f(n, t)
  return integr.quad(g, 0, 1)[0]
```

Si vous connaissez les fonctions anonymes, vous pouvez écrire, plus élégamment

```
def I(n):
  return integr.quad(lambda t : f(n, t), 0, 1)[0]
```

VIII.7 Appeler des fonctions ?

Lors d'un **appel** de fonction, la fonction est **appelée**... En particulier, si un appel est effectué plusieurs fois, les fonctions seront exécutées plusieurs fois.
On comparera les comportements suivants :

```
import numpy.random as rd

def Y(n):
  return sum(rd.binomial(3, 0.5, n))
```

```
def seuil1(M):
  if Y(100) > M:
    return (1, Y(100))
  elif Y(100) == M:
    return (2, Y(100))
  else:
    return (3, Y(100))

>>> seuil1(150)
(1, 154)
```

```
def seuil2(M):
  y = Y(100)
  if y > M:
    return (1, y)
  elif y == M:
    return (2, y)
  else:
    return (3, y)

>>> seuil2(150)
(3, 131)
```

Ce choix de programmation peut s'avérer problématique en termes de complexité.

```
def f(n):
  if n == 0:
    return 2
  else:
    return f(n-1) * f(n-1)
```

```
def g(n):
  if n == 0:
    return 2
  else:
    tmp = g(n-1)
    return tmp * tmp
```

Les deux fonctions renvoient 2^{2^n}. La première fonction effectue $2^n - 1$ multiplications ; la seconde effectue n multiplications !

VIII.8 `int`, `float` ?

En Python, les entiers sont codés avec un précision *infinie* (limitée par les capacités de l'ordinateur) alors que le comportement des flottants sont régis par le standard IEE-754.
En particulier, il existe un plus grand flottant...

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

>>> n = 171
>>> F1, F2 = fact(n), fact(n-1)
>>> f1, f2 = fact(float(n)), fact(float(n)-1)
>>> print(F1//F2, f1/f2)

>>> 171 inf
```

De plus, les tests d'égalité à `0.` ne sont pas pertinents avec les flottants.

```
def decroit(s):
    while s != 0.:
        s = s - 0.1
    return True

>>> print(decroit(0.1))
True
>>> print(decroit(0.3))
```

Le premier appel termine alors que le second ne termine pas!!
Autant que possible, nous préférerons le travail algorithmique avec des entiers!!