TP - Parcours séquentiel d'un tableau

> Dans presque tous les calculs, une grande variété d'arrangements pour la succession des processus est possible, et plusieurs considérations doivent influencer la sélection. L'une des considérations essentielle est de choisir l'arrangement qui tend à réduire au minimum le temps de calcul.
>
> Ada Lovelace

Ouvrez le fichier « TP-parcours-tabeau.py », et exécutez-le pour vérifier l'absence d'erreur. Toutes les fonctions créées dans ce TP seront enregistrées dans ce fichier et pourront être utilisées si besoin dans la suite du TP.

I. Coût en temps d'un algorithme

Le temps d'exécution d'un algorithme (ou de son implémentation en Python) est le produit du nombre d'instructions qui seront exécutées tout au long de son déroulement par le temps d'exécution d'une instruction.

temps d'exécution = nombre d'instructions × temps d'exécution d'une instruction

Activité 1.
On considère 2 algorithmes renvoyant le même résultat à partir du même nombre n de données.

- L'algorithme A1 effectue n^2 successions d'instructions de base.
- L'algorithme A2 effectue $10 \times n$ successions d'instructions de base.

On utilise deux machines aux capacités différentes :

- La machine M1 effectue 1 000 d'instructions de base par seconde (Intel 80286 20 MHz - 1982)
- La machine M2 effectue 100 000 000 d'instructions de base par seconde (Intel i7 3,5 GHz - 2010)

1. Déterminez le temps de calcul pour chaque machine et pour chaque algorithme si le nombre de données à traiter est n = 10, n = 1 000, n = 1 000 000 et n = 1 000 000 000. (Complétez les tableaux page 2)

2. L'ordre de grandeur (la puissance de 10) du nombre de secondes dans une journée est 10^5 ; L'ordre de grandeur du nombre de secondes dans un mois est 10^6 ; L'ordre de grandeur du nombre de secondes dans une année est 10^7.
Toutes ces données peuvent-elles être traitées dans un temps raisonnable ?

Pour l'algorithme A1 :

n =	10	1 000	1 000 000	1 000 000 000
M1				
M2				

Pour l'algorithme A2 :

n =	10	1 000	1 000 000	1 000 000 000
M1				
M2				

Conclusion : une machine puissante n'est rien sans un bon algorithme.

II. Outils pour mesurer le coût d'un algorithme

II. 1. Générer des tableaux aléatoires

Dans le module `random` de Python, il existe une fonction `randint` qui renvoie un nombre compris entre deux bornes, avec la même probabilité.

1. Quelle instruction permet d'importer la fonction *randint* du module *random* ?

   Saisissez cette instruction tout en haut de votre fichier, avant la déclaration des fonctions.

2. L'usage de `help(randint)` fournit tous les renseignements utiles à l'utilisation de cette fonction.
   Quelle instruction permet de générer un nombre aléatoire compris entre 1 et 50 inclus ?

3. Vérifiez votre réponse en tapant ce code dans la console Python.

4. Créez une fonction *remplissage_aleatoire*, prenant un entier n en argument, et renvoyant une liste contenant n entiers générés de façon aléatoire entre 0 et 100. On préfèrera le remplissage d'une liste par compréhension.
   Pour vérifier la bonne exécution de cette fonction, on vérifie que la longueur de la liste renvoyée est égale au paramètre n de la fonction *remplissage_aleatoire*.

```
>>> len(remplissage_aleatoire(50))
50
```

   Insérez ce jeu de test (pour une utilisation avec le module doctest) dans la documentation.

II. 2. Tracé de courbes

Pour tracer des courbes en Python, nous utiliserons le module matplotlib.pyplot. Pour cela, il suffit de l'importer. Afin d'éviter de réécrire le nom du module en entier à chaque fois, on peut l'importer avec la ligne suivante : `import matplotlib.pyplot as plt`.

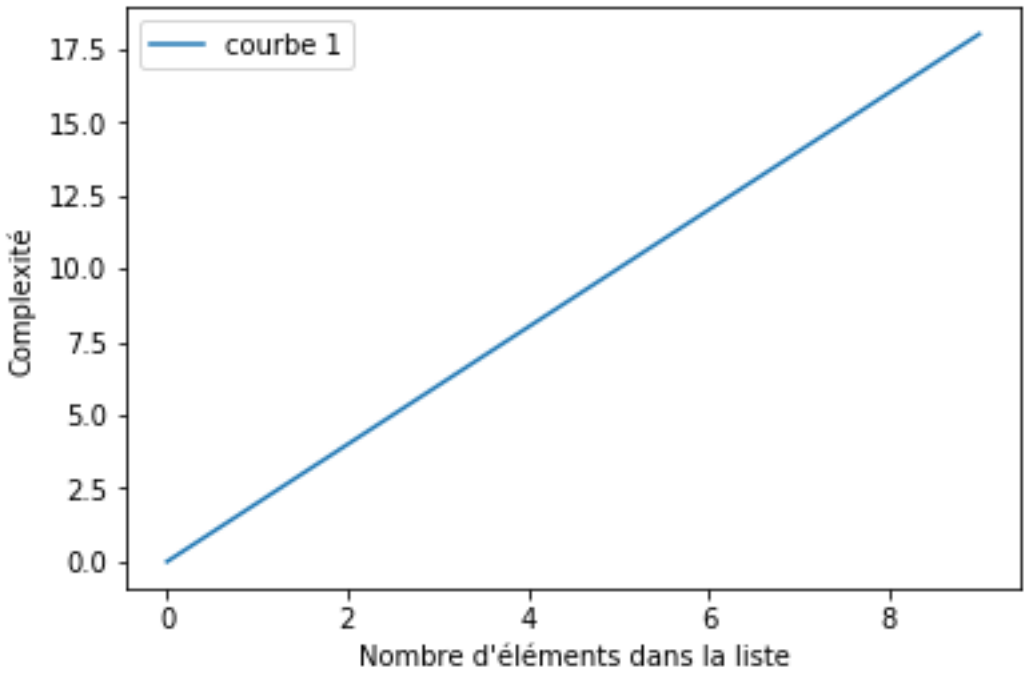
Les fonctions utiles pour ce TP du module matplotlib.pyplot sont les suivantes :

- `plt.plot(x, y, label=name)` quand x et y sont des listes de la même taille, génère la courbe qui passe par les points $(x[i], y[i])$. C'est la fonction usuelle pour tracer des courbes. Cette courbe aura pour étiquette name.
- `plt.xlabel(name)` : donne à l'axe (x) la légende name.
- `plt.ylabel(name)`p : donne à l'axe (y) la légende name.
- `plt.legend()` : met les légendes sur les axes (à appeler après les deux fonction fixant les légendes des axes).
- `plt.show()` : affiche le graphique (à appeler après tous les autres appels).

La documentation officielle de cette bibliothèque est au lien suivant : `https://matplotlib.org/index.html`.

1. Saisissez l'instruction d'import du module *matplotlib* tout en haut de votre fichier, avant la déclaration des fonctions.
2. Créez une fonction *trace* prenant x, y et *legende* en argument, et affichant le graphique donnant $y = f(x)$.

3. L'exécution de `trace(abscisses, ordonnees, "courbe 1")` donne le graphique suivant :



Vérifiez la bonne exécution de votre fonction.

II. 3. Le module time

Le module `time` fournit différentes fonctions liées au temps. La documentation est disponible à l'adresse : `https://docs.python.org/fr/3/library/time.html`

L'heure Unix ou heure Posix (aussi appelée Unix timestamp) est une mesure du temps basée sur le nombre de secondes écoulées depuis le 1er janvier 1970 00 :00 :00 UTC. Elle est utilisée principalement dans les systèmes qui respectent la norme POSIX, dont les systèmes de type Unix, d'où son nom. C'est la représentation POSIX du temps[1].
Le 1er janvier 1970 00 :00 :00 est appelé *epoch*.
Pour utiliser le module time, il suffit de l'importer avec la ligne suivante : `import time`.

La fonction `time.time()` renvoie le temps en secondes depuis *epoch* sous forme de nombre à virgule flottante.

```
>>> time.time()
1576949367.5409153
```

1. Saisissez l'instruction d'import du module time tout en haut de votre ficher, avant la déclaration des fonctions.
2. Dans la console Python, saisissez l'instruction permettant de déterminer le nombre de secondes écoulées depuis *epoch*.
3. On souhaite connaître le temps d'exécution de la fonction *remplissage_aleatoire*. Écrivez une fonction *temps_execution_aleatoire*, sans argument d'entrée, renvoyant la durée d'exécution de la fonction *remplissage_aleatoire* lorsque n prend la valeur 100. Basez vous sur l'exemple d'exécution ci-dessous (la valeur renvoyée dépend de la vitesse d'horologe de chaque ordinateur).

```
>>> temps_execution_aleatoire()
0.00043010711669921875
```

III. Moyenne des éléments d'un tableau

Rappel : Python identifie listes et tableaux.

III. 1. Écriture des postconditions

1. Quelle est la moyenne de la liste *abscisses* ?
2. Quelle est la moyenne de la liste *ordonnes* ?

III. 2. Implémentation en Python

On s'intéresse à la fonction suivante :

```
def moyenne_liste(liste):
    """fonction qui calcule la moyenne de tous les éléments d'une liste"""
    somme = 0
    for element in liste :
        somme = somme + element
    moyenne = somme / len(liste)
    return moyenne
```

1. Utilisez les deux exemples d'exécution précédents dans la documentation de la fonction, pour une utilisation avec le module `doctest`.

1. Source Wikipedia `https://fr.wikipedia.org/wiki/Heure_Unix`

2. Créez deux assertions permettant de s'assurer que la liste n'est pas vide, et que le premier élément est bien un nombre (on admettra que tous les éléments sont du même type).

3. Exécutez le fichier pour s'assurer de l'absence d'erreur.

III. 3. Tracé du temps d'exécution en fonction de n

On souhaite montrer que le coût en temps de cette fonction est linéaire.

Pour voir l'évolution du temps d'exécution en fonction du nombre n d'éléments dans une liste, on va réaliser la succession d'opérations suivantes :

- ⊛ créer une liste *liste_temps* contenant 100 fois la valeur 0,
- ⊛ créer une liste *liste_n* contenant [1, 2, ..., 100],
- ⊛ utiliser la fonction *remplissage_aleatoire* pour créer une liste de taille n,
- ⊛ utiliser la fonction *time* du module `time` pour déterminer le temps d'exécution de la fonction *moyenne_liste* pour une liste de taille n.
- ⊛ insérer dans *liste_temps*, la valeur du temps d'exécution précédemment trouvé. **N'oubliez pas que pour n = 1, on remplit** *liste_temps*[0],
- ⊛ **répéter** les deux dernières opérations pour toutes les valeurs de n comprises entre 1 et 100 inclus,
- ⊛ utiliser la fonction *trace* pour tracer le graphique donnant le temps d'exécution en fonction du nombre d'éléments dans la liste : $temps\_execution = f(n)$.

Écrivez une fonction *trace_temps_moyenne*, sans argument, permettant de réaliser l'ensemble de ces opérations.

1. Quelle est la forme du graphique ?
2. Le coût en temps est-il linéaire ?
3. Exécutez plusieurs fois cette fonction. Le graphique obtenu est-il toujours le même ? Proposez une justification.

IV. Recherche d'un élément dans une liste

IV. 1. algorithme de recherche

On se propose d'étudier un algorithme qui prend en argument *liste* et x, et renvoyant le plus petit indice i tel que $liste[i] = x$, et -1 si l'élément n'est pas présent dans la liste.

Algorithme : recherche(liste,x)

```
/* algorithme qui recherche l'élément x dans une liste                    */
```
Entrées : $liste, x$: la liste et l'élément recherché
Sorties : $indice$: la valeur du plus petit indice tel que $liste[i] = x$

```
1  liste : une liste d'entiers;
2  x, i, indice : des entiers;
3  début
4      pour i allant de 1 à longueur de liste inclus par pas de 1 faire
5          si liste[indice] = x alors
6              retourner indice
7          fin si
8      fin pour
9      retourner −1
10 fin
```

1. Que renvoie cet algorithme si *liste* prend la valeur [1, 3 8] et x prend la valeur 3 ?
2. Que renvoie cet algorithme si *liste* prend la valeur [1, 3 8] et x prend la valeur 2 ?
3. Cet algorithme prend-t-il toujours fin ?

IV. 2. Implémentation en Python

Écrivez une fonction *recherche* prenant *liste* et x en argument et renvoyant le plus petit indice i tel que $liste[i] = x$, et -1 si l'élément n'est pas présent dans la liste.
L'exécution de la fonction donne les résultats suivants :

```
>>> recherche(abscisses, 9)
9
>>> recherche(abscisses, 0)
0
>>> recherche(abscisses, 10)
-1
```

Insérez ce jeu de test (pour une utilisation avec le module doctest) dans la documentation.

IV. 3. Tracé du temps d'exécution en fonction de n

1. En vous basant sur l'exemple précédent (partie III), écrivez une fonction *trace_temps_recherche*, sans argument, permettant de tracer le temps d'exécution en fonction de n : $temps\_execution = f(n)$.
2. Quelle est la forme du graphique ?
3. Le coût en temps est-il linéaire ?
4. Exécutez plusieurs fois cette fonction. Le graphique obtenu est-il toujours le même ? Proposez une justification.

V. Recherche d'un extremum dans une liste

V. 1. algorithme de recherche

On se propose de réaliser un algorithme qui prend en argument *liste*, et renvoyant le plus petit élément ET le plus grand élément de la liste.

1. Que renvoie cet algorithme si *liste* prend la valeur [1, 3 8].
2. Proposez un algorithme répondant au cahier des charges.

V. 2. Implémentation en Python

Écrivez une fonction *extremum* prenant *liste* en argument et renvoyant le plus petit ET le plus grand élément de la liste.
L'exécution de la fonction donne les résultats suivants :

```
>>> recherche(abscisses)
(0, 9)
>>> recherche(ordonnees)
(0, 18)
```

Insérez ce jeu de test (pour une utilisation avec le module doctest) dans la documentation.

V. 3. Tracé du temps d'exécution en fonction de n

1. En vous basant sur les deux exemples précédents, écrivez une fonction *trace_temps_extremum*, sans argument, permettant de tracer le temps d'exécution en fonction de n : $temps\_execution = f(n)$.
2. Quelle est la forme du graphique ?
3. Le coût en temps est-il linéaire ?