TP 5 – COMPLEXITÉ

Info1.Algo1 - 2022-2023 Semestre Impair

Lors de ce TP, **il est nécessaire de :**

- **Exécuter le code sur machine** (c'est-à-dire *hors ligne*), sinon les mesures de temps d'exécution seraient impossibles ou faussées.
- **Conserver une trace écrite** (sur papier ou sur fichier texte/python) :
  - des **résultats expérimentaux** observés
  - des **interprétations** réalisées.

Rappel

La mesure du temps d'exécution d'une instruction s'effectue de la façon suivante :

```
import time # à écrire une seule fois en tout début de fichier

tic = time.time()
... # écrire ici l'intruction à mesurer
tac = time.time()
print('Instruction exécutée en',tac-tic,'secondes')
```

Exercice 1 - Somme des entiers ★

Dans cet exercice on mesure la complexité en temps des deux fonctions `somme_entiers_1` et `somme_entiers_2` qui calculent et retournent la somme des entiers de `1` à `n`, où `n` est un entier positif donné en paramètre.

1) a) Dans le fichier **ex01_somme_entiers.py**, écrire un programme permettant de mesurer le temps d'exécution de l'appel `somme_entiers_1(n)`pour des valeurs de `n` comprises entre 10^0 et 10^{12}. **Relever les temps d'exécution** obtenus.

b) Comment évolue le temps d'exécution de l'appel `somme_entiers_1(n)` lorsque le nombre `n` passé en paramètre est multiplié par `10` ? **Interpréter**.

2) a) Modifier le fichier **ex01_somme_entiers.py** afin de mesurer cette fois-ci le temps d'exécution des appels `somme_entiers_2(n)`. **Relever les temps d'exécution** obtenus.

b) Comment semble évoluer le temps d'exécution de l'appel `somme_entiers_2(n)` lorsque le nombre `n` passé en paramètre est multiplié par `10` ? **Interpréter**.

Exercice 2 - Puissance ★

Dans cet exercice, on mesure la complexité en temps des deux fonctions `puissance_A` et `puissance_B` qui acceptent en paramètres deux entiers naturels `a` et `n` et retournent la valeur de `a**n`.

Dans cet exercice, on choisira une valeur de `a` arbitraire qui restera constante pendant toutes les mesures.

1) a) Dans le fichier **ex02__puissance.py**, écrire un programme permettant de mesurer le temps d'exécution de l'appel `puissance_A(a,n)` pour des valeurs de `n` comprises entre et 10^0 et 10^{12}. **Relever les temps d'exécution** obtenus.

b) Comment semble évoluer le temps d'exécution de l'appel `puissance_A(a,n)` lorsque `n` est multiplié par `10` ?

2) a) Modifier le corps du programme du fichier **ex02__puissance.py** de manière à mesurer le temps d'exécution des appels `puissance_B(a,n)`. **Relever les temps d'exécution** obtenus.

b) Comment évolue le temps d'exécution de l'appel `puissance_B(tab)` lorsque `n` est multiplié par `10` ?

Exercice 3 - Somme maximale ★★

1) Le fichier **ex03__somme__maximale__q1.py** fournit la fonction `somme_maximale` qui accepte en paramètre un tableau `tab` de taille `n` au moins égale à 2 et détermine la plus grande somme de deux éléments d'indices distincts de `tab`.

a) Exprimer en fonction de `n` le nombre de comparaisons effectuées dans la fonction `somme_maximale`. En déduire la complexité en temps de cet algorithme.

b) Compléter le programme afin de mesurer le temps d'exécution de l'appel `somme_maximale(tab)`, où `tab` est un tableau aléatoire de taille `n` fourni par l'appel `tableau_aleatoire(n)`. Choisir les valeurs de `n` selon des puissances croissantes de 10.

c) Relever les temps d'exécution obtenus. Comment évolue le temps d'exécution de l'appel `somme_maximale(tab)` lorsque la taille `n` du tableau passé en paramètre est multipliée par `10` ? **Interpréter**.

2) On se propose d'améliorer la complexité en temps **en ne parcourant qu'une seule fois le tableau** `tab`. L'idée qui permet de réaliser cette amélioration est de déterminer au fur et à mesure de ce parcours la première et la deuxième plus grandes valeurs du tableau.

a) Dans le fichier **ex03_somme_maximale_q2.py**, compléter la fonction `deux_maxima` qui accepte en paramètre un tableau d'entiers `tab` de longueur `n` au moins égale à `2` et retourne un tuple `(max1,max2)`, où `max1` est la plus grande valeur du tableau (c'est-à-dire le maximum) et `max2` la deuxième plus grande valeur du tableau.

Modèle de solution : À chaque nouvelle valeur rencontrée :

- Si elle remplace le maximum actuel alors le maximum actuel prend la place de la deuxième plus grande valeur.
- Sinon elle peut éventuellement prendre la place de la deuxième plus grande valeur.

Rappel des contraintes :

- N'effectuer qu'un seul parcours du tableau.
- Respecter l'abstraction tableau.
- Ne pas modifier le tableau.
- Ne pas utiliser d'autre tableau que celui donné en paramètre.
- Ne pas utiliser de fonctions élaborée de Python pour déterminer le maximum (`max`, `sorted`, ...).

b) Toujours dans le fichier **ex03_somme_maximale_q2.py**, compléter la fonction `somme_maximale` avec un appel de la fonction `deux_maxima` et vérifier expérimentalement que que la complexité en temps de cette nouvelle solution est bien celle attendue.

Exercice 4 - Meilleure plus-value ★★

Dans cet exercice, `tab` est un tableau **non vide** d'entiers représentant l'évolution du cours en bourse (en €) d'une action pendant une série de jours consécutifs. Au jour `i`, la valeur de cette action est donc `tab[i]`.

On appelle plus-value sur une action le gain réalisé entre l'achat et la revente d'une action. Cette plus-value est égale au prix de vente diminué du prix d'achat (*si la différence entre le prix de vente et le prix d'achat fait apparaître une perte, on parle alors de moins-value*).

L'objectif de cet exercice est d'écrire la fonction `meilleure_plus_value` qui accepte en paramètre le tableau `tab`, et retourne la **meilleure plus-value** que l'on aurait pu effectuer au cours de l'évolution du cours en bourse représentée par `tab`.

> **On cherche donc la valeur maximale des différences `tab[j]-tab[i]`, où `i` et `j` sont deux entiers tels que `0<=i<=j<len(tab)`.**

Exemples :

Entrée	Sortie
`[11,15,9,7,8,14]`	`7`
`[124,112,65,63,51,43]`	`0`

- Dans le premier exemple, la meilleure plus-value `tab[j]-tab[i]` est obtenue pour `i=3` et `j=5`, et vaut `14-7`. La valeur de `tab[j]-tab[i]` obtenue pour `i=3` et `j=1` est certes plus grande (elle vaut `8`) mais elle ne respecte pas la condition `i<=j`.
- Dans le second exemple, l'action est toujours à la baisse, on ne peut faire aucune plus-value. La meilleure plus-value `tab[j]-tab[i]` est obtenue dans le cas où `i==j` (par exemple `i=0` et `j=0`).

1) Dans le fichier **ex04__meilleure__plus__value__q1.py** sont fournies deux fonctions `meilleure_plus_value_A` et `meilleure_plus_value_B` qui répondent au problème posé.

a) Écrire un programme permettant de mesurer le temps d'exécution de l'appel `meilleure_plus_value_A(tab)`, où `tab` est un tableau aléatoire de taille `n` fourni par l'appel `tableau_aleatoire(n)`. Choisir les valeurs de `n` selon des puissances croissantes de 10. **Relever les temps d'exécution** obtenus.

b) Comment évolue le temps d'exécution de l'appel `meilleure_plus_value_A(tab)` lorsque la taille `n` du tableau passé en paramètre est multipliée par `10` ? **Interpréter**.

c) Modifier le fichier afin de mesurer cette fois-ci le temps d'exécution des appels `meilleure_plus_value_B(tab)`. **Relever les temps d'exécution** obtenus.

d) Comment semble évoluer le temps d'exécution de l'appel `meilleure_plus_value_B(tab)` lorsque la taille `n` du tableau passé en paramètre est multipliée par `10` ? **Interpréter**.

2) Dans cette question, on se propose d'améliorer la complexité en temps de la fonction `meilleure_plus_value` **en ne parcourant qu'une seule fois** le tableau `tab`. L'idée qui permet de réaliser cette amélioration consite à conserver au fur et à mesure de cet unique parcours :

- le **minimum** des éléments lus jusqu'à présent.
- la **meilleure plus value** possible avec les valeurs déjà rencontrées.

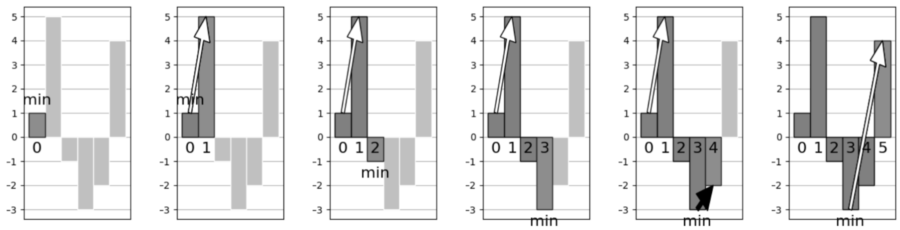
À chaque nouvel élément lu, on teste si cet élément :

- doit remplacer le minimum.
- permet une meilleure plus value.

On actualise alors le minimum ou la meilleure plus value courants si nécessaire.

Exemple de déroulé :

La figure ci-dessous donne le déroulé de l'algorithme lorsque `tab` est `[1,5,-1,-3,-2,4]`. Les flèches blanches représentent la meilleure plus-value observée à chaque étape, la flèche noire une plus-value non conservée.



La trace de l'algorithme est alors la suivante :

indice lecture	minimum	meilleure plus-value
`0`	`1`	`0`
`1`	`1`	`4`
`2`	`-1`	`4`
`3`	`-3`	`4`
`4`	`-3`	`4` (la plus-value `1` est ignorée)
`5`	`-3`	`7`

a) Dans le fichier **ex04_meilleure_plus_value_q2.py**, compléter le corps de la fonction `meilleure_plus_value_C` de telle sorte qu'elle passe la fonction de test `test_meilleure_plus_value`.

Rappel des contraintes :

La fonction doit :

- Respecter l'abstraction tableau.
- Ne pas utiliser d'autre tableau que celui donné en entrée.
- N'effectuer qu'un seul parcours du tableau.
- Ne pas utiliser de fonctions élaborée de Python pour déterminer le minimum ou le maximum (`min`, `max`, `sorted`, …).

b) Écrire un programme permettant de mesurer le temps d'exécution de l'appel `meilleure_plus_value_C(tab)`, où `tab` est un tableau aléatoire de taille `n` fourni par l'appel `tableau_aleatoire(n)`. **Relever les temps d'exécution** obtenus. **Interpréter**.

Exercice 5 - Opérateur bit à bit ★★

On appelle **opérateur bit à bit** un opérateur agissant sur la représentation en binaire des entiers sur lesquels il opère (ses **opérandes**). En Python il existe entre autres les opérateurs :

- `&` : **et** bit à bit
- `|` : **ou** bit à bit
- `^` : **ou exclusif** bit à bit

Exemple :

Expression	**Représentation binaire**	**Interprétation**
26	...00011010	
22	...00010110	
26&22	...00010010	26&22 vaut donc 18
26|22	...00011110	26|22 vaut donc 30
26^22	...00001100	26^22 vaut donc 12

1) Dans le fichier **ex05_maximum_et.py**, compléter la fonction `maximum_et_bit_a_bit` qui accepte en entrée un tableau d'entiers positifs de taille `n` au moins 2 et retourne la valeur maximale de `tableau[i]&tableau[j]`, où `i` et `j` sont deux indices différents dans le tableau.

Contrainte : Ne pas modifier la structure des deux boucles `for` déjà fournies dans le fichier.

2) a) Exprimer en fonction de `n` le nombre de passage dans la boucle `for j in ...`.

b) Écrire un programme permettant de mesurer le temps d'exécution de l'appel `maximum_et_bit_a_bit(tableau)`, où `tableau` est un tableau aléatoire de taille `n` fourni par l'appel `tableau_aleatoire(n)`. Choisir les valeurs de `n` selon des puissances croissantes de 10. **Relever les temps d'exécution** obtenus.

c) Comment évolue le temps d'exécution de l'appel `maximum_et_bit_a_bit(tableau)` lorsque la taille `n` du tableau passé en paramètre est multipliée par `10` ? Cela est-il cohérent avec le résultat de la question **2a**?

3) Si l'on cherche à améliorer cette complexité il faudrait trouver une relation qui permette d'identifier plus efficacement les candidats susceptibles d'atteindre le maximum. On pourrait par exemple chercher à identifier l'élément le plus prometteur (qui a le plus de bits à `1` dans sa représentation binaire ?) et chercher à optimiser le second argument. Nous n'avons pas trouvé de solution à ce problème, la question 3) est donc une question ouverte !