TP Informatique 5 : Investissements boursiers

1 Introduction

1.1 Présentation du problème

Un trader a acheté, puis revendu quelques temps plus tard, une action, réalisant une certaine plus-value. Il s'interroge toutefois sur son « efficacité », et souhaite comparer sa plus-value à la plus grande plus-value qu'il aurait été possible de faire avec un achat puis une revente de cette même action, à des moments potentiellement différents.

Pour répondre à cette question, on dispose des évolutions de l'action, minute par minute, sous la forme d'une liste de valeurs représentant ses hausses (valeurs positives) et ses baisses (valeurs négatives). Sur une année, cela représente une liste de 100000 valeurs.

Pour simplifier et éviter de transférer une liste aussi longue, nous allons la générer aléatoirement. Cependant, l'inconvénient d'une génération aléatoire est que les valeurs seront différentes à chaque exécution du programme, rendant la recherche des erreurs et la comparaison des résultats difficiles.

Heureusement, les langages de programmation disposent d'une solution pour résoudre ce problème. Les fonctions retournant des valeurs dites « aléatoires », telles que `random.randint` ou `random.gauss`, ne retournent en fait pas des valeurs réellement aléatoires, mais font utilisent des fonctions suffisamment chaotiques pour que les valeurs fournies apparaissent suffisamment imprévisibles pour que l'on ne puisse les distinguer, à première vue, de valeurs aléatoires (on parle de générateur pseudo-aléatoire).

Ces fonctions sont cependant déterministes, et les valeurs sortent toujours dans le même ordre. Le « point de départ » de la série, en revanche, est généralement choisi aléatoirement (en se basant par exemple sur l'heure du système). Mais il est possible de choisir ce point de départ en utilisant la fonction `random.seed`. Ainsi, les trois lignes suivantes :

```
from random import gauss, seed

seed('Hello World!')

L = [ round(gauss(0.0, 1.0), 2) for _ in range(100000) ]
```

génèrent une liste de 100000 valeurs réelles pseudo-aléatoires (la fonction `gauss(mu, sigma)` retourne une valeur issue d'un processus pseudo-aléatoire gaussien de moyenne `mu` et d'écart-type `sigma`), arrondies à deux décimales, mais qui seront toujours les mêmes ! On peut le constater en affichant les premières valeurs de L.

Si l'on souhaite d'autres valeurs aléatoires, il suffira de changer l'argument de la fonction `seed` (on y a ici placé une chaîne de caractères, mais on peut y mettre à peu près n'importe quoi).

1.2 Préparation

1. Construire la liste `L` comme indiquée ci-dessus, et vérifier que `sum(L)=-407.62`.

Dans la suite, on cherchera donc à déterminer le gain maximal que l'on pouvait effectuer, soit en terme mathématiques

$$\max_{0\leq i<j\leq 100000}\left(\sum_{k=i}^{j-1} \mathrm{L}[k]\right)$$

Comme la liste L est très longue, les calculs sur cette liste peuvent prendre beaucoup de temps, et la vérification du résultat être malcommode...

2. Construire donc, grâce à un slice, quatre listes supplémentaires, `L1` contenant les dix premières valeurs de `L`, `L2` contenant les cent premières valeurs de `L`, `L3` contenant les mille premières valeurs de `L` et `L4` contenant les dix mille premières valeurs de `L`.

1.3 Objectifs

Le but de cette séance de travaux pratique est de montrer que, s'il existe différents algorithmes qui permettent d'obtenir le résultat que l'on cherche, certains sont plus intéressants que d'autres, car ils permettent par exemple d'obtenir ce résultat en un temps raisonnable, tandis que d'autres pourraient bien mettre des millions d'années pour y parvenir...

Le temps de calcul n'est pas le seul critère qui entre en ligne de compte. On peut aussi parfois vouloir ne pas utiliser trop de mémoire, par exemple, or certains algorithmes très rapides sont parfois gourmands en mémoire. Mais dans les deux années qui viennent, c'est principalement le temps de calcul que l'on cherchera à maîtriser.

Dans la suite, nous mettrons en œuvre trois façons distinctes de calculer le résultat recherché, et comparerons leur efficacité en terme de temps de calcul.

2 Une première approche

Puisque l'on cherche à déterminer le maximum, pour tout i et j, de l'expression

$$\sum_{k=i}^{j-1} \mathrm{L}[k]$$

on peut commencer par écrire une fonction calculant ladite somme.

1.a Écrire une fonction `Somme(L, i, j)` qui retourne la somme précédente.

1.b Combien de sommes effectue-t-on dans cette fonction, selon les valeurs de `i` et `j` ?

2.a En déduire une fonction `GainMax_1(L)` qui calcule le gain maximal recherché, en appelant la fonction précédente pour toutes les valeurs acceptables de `i` et `j`. On pourra supposer que la liste n'est pas vide (peu importe si l'algorithme retourne une erreur si ce n'est pas le cas).

2.b Vérifier son bon fonctionnement sur `L1`.

2.c Déterminer précisément le nombre de sommes effectuées en fonction du nombre n d'éléments de `L`.

On rappelle, pour ce faire, que $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$ et $\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$.

En pratique, on s'intéresse surtout au terme dominant, ici n^3, qui sera le plus important lorsque n sera grand. On dira que l'algorithme auquel on a affaire ici a « une complexité en n^3 », c'est-à-dire que le temps est grossièrement proportionnel au nombre de données à traiter élevé au cube.

On va chercher à déterminer exactement le temps nécessaire pour traiter les données de la liste `L1`. Pour ce faire, nous allons utiliser la fonction `time` du module `time`, qui retourne une valeur exprimée en seconde qui correspond en gros à l'heure courante.

Ainsi, afin de déterminer le temps d'exécution d'un appel à une fonction, on mémorise l'« heure » avant d'appeler la fonction, puis l'« heure » une fois que le résultat a été obtenu, et on calcule la différence entre ces deux instants. On pourra procéder de la sorte :

```
from time import time

t_debut = time()
res = GainMax_1(L2)
t_fin = time()
print("Le temps écoulé est de", t_fin-t_debut, "secondes.")
```

À noter que si l'on utilise l'interpréteur IPhython dans Pyzo (reconnaissable à son invite « `In` »), c'est plus simple, il existe une « magic function » pour mesurer le temps d'exécution d'une fonction :

```
%timeit -n1 GainMax_1(L2)
```

3.a Mesurer le temps d'exécution de la fonction `GainMax_1` pour la liste `L2` et `L3`.

3.b Prévoir (sans essayer) le temps qui serait nécessaire pour utiliser la fonction sur les listes `L4` et `L`.

Données à traiter	L2	L3	L4	L
Temps	mesuré :	mesuré :	estimé :	estimé :

3 Un algorithme un peu plus intelligent

1.a Montrer qu'il est possible de déterminer

$$\mathscr{A}(i) = \max_{i<j\leq 100000} \left(\sum_{k=i}^{j-1} \mathrm{L}[k] \right)$$

en effectuant seulement $n-1-i$ sommes et $n-1-i$ comparaisons (où n correspond à la longueur de la liste `L`).

1.b Sur ce principe, écrire une fonction `GainMaxAux(L, i)` qui détermine, en effectuant $n-1-i$ sommes et $n-1-i$ comparaisons, le gain maximal que l'on peut obtenir si l'on a acheté l'action au moment correspondant à `i`.

2.a En déduire une fonction `GainMax_2(L)` qui, grâce à la fonction `GainMaxAux`, détermine le gain maximal en effectuant nettement moins d'opérations, et montrer que le nombre d'opérations est s'exprime, cette fois, en fonction de n^2

2.b Utiliser la fonction `GainMax_2` sur les différentes listes et noter son temps d'exécution dans chaque cas. Vérifier que ces temps sont compatibles avec un nombre d'opérations proportionnel à n^2.

Données à traiter	L2	L3	L4	L
Temps				

4 Encore un peu plus efficace

Nous allons à présent montrer qu'il est possible de faire encore mieux, avec un nombre d'opérations proportionnel à n, c'est-à-dire en n'effectuant qu'un seul passage dans la liste passée en argument.

L'idée est la suivante : on reconstitue le cours de l'action à chaque instant (on pourra partir d'une valeur quelconque : comme on ne s'intéresse qu'à une plus-value et que l'on a des écarts à ajouter ou soustraire, cela n'aura pas d'importance), et on mémorise, pour chacun de ces instants, le cours le plus bas observé.

Si l'on vend à un instant quelconque, pour avoir la plus grande plus-value, il faudra avoir acheté justement à cet instant où le cours était au plus bas !

1. Écrire une fonction `GainMax_3(L)` qui parcourt une seule fois la liste fournie en argument, et en s'aidant des suggestions précédentes, détermine la plus grande plus-value qu'il était possible d'effectuer. Puis comparer les temps d'exécution et le lier à l'ordre de grandeur du nombre d'opérations effectuées, en fonction du nombre n de données dans la liste.

Données à traiter	L2	L3	L4	L
Temps				

5 Deux dimensions

On s'intéresse à présent à une matrice M de taille $n \times n$, contenant des valeurs $M_{i,j}$ réelles, et on s'intéresse au problème de la somme maximale d'une sous-matrice rectangulaire, c'est-à-dire à la quantité

$$\max_{\substack{0 \leqslant i_1 < j_1 \leqslant n \\ 0 \leqslant i_1 < j_1 \leqslant n}} \left(\sum_{k=i_1}^{j_1-1} \sum_{k'=i_2}^{j_2-1} M[k][k'] \right)$$

1. Quelle est la complexité que l'on peut obtenir naïvement, en calculant toutes les sommes, puis en en cherchant le maximum ?

2. En s'inspirant de ce qui a été fait sur le cas unidimensionnel, chercher une méthode dont le nombre d'opérations est de l'ordre de n^3.

Pour les tests, on pourra générer une matrice de taille 1000×1000 par exemple en utilisant l'instruction suivante :

```
M = [ [ round(gauss(0.0, 1.0), 2) for _ in range(1000) ]
        for _ in range(1000) ]
```