TD d'informatique

Révisions sur les algorithmes de tri

2025-2026

1. Préliminaires : simulations de choix aléatoires

1.a. Ecrire une fonction ayant pour paramètre d'entrée un entier n, et renvoyant en retour une liste de n flottants compris entre 0 et 1, choisis au hasard.

1.b. Ecrire une fonction ayant pour paramètres d'entrée deux entiers n et p, et renvoyant en retour une liste de n entiers compris entre 0 et p, choisis au hasard.

1.c. En utilisant le module time, écrire une deuxième version de la fonction précédente, renvoyant en outre une estimation de son temps d'exécution.

2. Implémentation de trois algorithmes de tri (ou plus…)

On rappelle (ou on donne) le principe du tri par sélection, du tri à bulles et du tri par insertion :

i. Le tri par sélection

On crée une liste T, vide. On cherche le plus petit élément de L. On le place dans la liste T, et on le supprime de la liste L. Puis on cherche le nouveau plus petit élément de L, que l'on ajoute à la liste T et que l'on supprime de la liste L. On répète l'opération jusqu'à ce que la liste L soit vide : T contient alors tous les éléments de L, rangés dans l'ordre croissant.

Raffinement : le tri par sélection en place

Créer une liste, lui ajouter des éléments, et surtout supprimer un élément d'une liste, sont des opérations coûteuses (à chaque suppression, la liste L est recrée). On peut éviter cet inconvénient en utilisant la variante suivante du tri par sélection :

On cherche le plus petit élément de L, et on l'échange avec $L[0]$.

Puis on cherche le plus petit élément de $[L[1], ..., L[n-1]]$, que l'on échange avec $L[1]$. On cherche ensuite le plus petit élément de $[L[2], ..., L[n-1]]$, on l'échange avec $L[2]$, et ainsi de suite…

Remarque

Il pourra être pratique d'écrire et d'utiliser un algorithme annexe, recherchant la place du plus petit élément d'une liste.

ii. Le tri à bulles

On parcourt la liste L. Pour $i \in [0, n-2]$, si $L[i]$ est supérieur à $L[i+1]$, on échange ces deux éléments.

On effectue cette opération de parcours-échanges jusqu'à ce que la liste soit entièrement triée.

Possibilité 1

Remarquons qu'à l'issue du premier parcours de la liste, le plus grand élément de L sera nécessairement à la bonne place (la dernière). Après deux parcours, on est sûr que les deux plus grands éléments de L seront bien placés, et ainsi de suite… en fin de compte, après $n-1$ parcours de la liste, les $n-1$ plus grands éléments de L sont bien placés, le plus petit élément de L l'est donc aussi : la liste L est triée.

Possibilité 2

On peut également recommencer le parcours jusqu'à ce que, lors de celui-ci, aucun échange ne soit plus constaté : c'est qu'alors la liste est triée.

iii. Le tri par insertion

Le principe

On crée une liste T, contenant uniquement $L[0]$. On y insère $L[1]$, en le plaçant correctement par rapport à $L[0]$. On ajoute ensuite $L[2]$, mis à la bonne place par rapport à $L[0]$ et $L[1]$, et ainsi de suite…

Remarque 1

Là encore, on peut améliorer l'algorithme en adoptant une méthode de tri par insertion en place : Si $L[1] < L[0]$, on échange ces deux éléments, la sous-liste $[L[0], L[1]]$ est alors triée

Ensuite, si $L[2] < L[1]$, on échange ces deux éléments, et, si nécessaire, on fait également l'échange avec $L[0]$, de sorte que la sous-liste $[L[0], L[1], L[2]]$ soit correctement triée. De manière générale, la sous-liste $[L[0],..., L[i-1]]$ étant triée, on fait en sorte, à l'aide d'échanges successifs, que la sous-liste $[L[0],..., L[i-1], L[i]]$ le soit.

Remarque 2

Quelle que soit la version du tri par insertion utilisée, on voit qu'une fonction annexe serait utile :

Etant donnée une liste $T = [T[0], ..., T[n-1]]$ telle que la sous-liste $[T[0], ..., T[n-2]]$ soit triée, cette fonction fera en sorte qu'à l'aide d'échanges successifs du dernier élément avec les autres, la liste T soit entièrement triée.

2.a. Ecrire une fonction **triselec** qui effectue le tri par sélection d'une liste d'entiers ou de flottants L passée en argument.

2.b. Ecrire une fonction **tribulle** qui effectue le tri à bulles d'une liste d'entiers ou de flottants L passée en argument.

2.c. Ecrire une fonction **triinsert** qui effectue le tri par insertion en place (on rappelle que le "en place" signifie qu'on modifie la liste passée en argument, sans en créer de nouvelle).

Si on a le temps :

2.d. Implémenter le tri fusion (cours de première année).

2.e. Implémenter le tri rapide.

3. On teste…

On désire tester le temps nécessaire à chacune des fonctions de tri précédentes pour classer une « grande » liste d'entiers ou de flottants choisis au hasard.

3.a. Ecrire une fonction permettant, une liste L étant donnée, de trier L à l'aide de chacune des fonctions écrite en **2.** .

La fonction copy du module copy pourra s'avérer utile… pourquoi ?

On est prié d'éviter de fastidieuses répétitions d'instructions, en les remplaçant par exemple par :

Tris = [tribulle, triselec, triinsert, trifusion, trirapide]

for tri in Tris :

3.b. En utilisant la fonction time du module time, modifier la fonction précédente de manière à ce qu'elle renvoie le temps moyen, sur N essais, mis par chacune des fonctions de tris pour trier une liste de taille n choisie au hasard.

3.c. Ecrire une fonction permettant, pour chacune des fonctions de tri, de représenter l'évolution de la complexité temporelle en fonction de la taille de la liste triée (*avec les notations précédentes, on pourra prendre* $N = 10$ *et faire varier* n *de* 100 à 500, *avec un pas de* 20).

4. Un autre exemple d'algorithme de tri

Tri par dénombrement, ou par comptage

On dispose d'une liste L de longueur N, dont les coefficients sont des entiers compris entre 0 et p. On se propose de trier cette liste sans faire de comparaison entre ses éléments. Pour cela, on compte, pour chaque entier $n \in \{0, ..., p\}$, combien de fois n apparaît dans L. On en déduit la liste triée.

4.a. Implémenter cet algorithme en Python.

4.b. Estimer sa complexité temporelle en fonction de N et de p.
On précisera d'abord les opérations élémentaires considérées.

5. Un problème de pesée

On dispose de n pièces d'or indexées de 1 à n, parmi lesquelles une est fausse, et donc plus légère que les autres. On dispose d'une balance à deux plateaux de type Roberval, qui permet uniquement de déterminer si l'un des deux plateaux est plus lourd que l'autre.
On cherche à détecter rapidement la pièce défectueuse, c'est-à dire déterminer son indice.

L'idée est de répartir les pièces en tas de $\left\lfloor \frac{n}{3} \right\rfloor$, $\left\lfloor \frac{n}{3} \right\rfloor$ et $n - 2\left\lfloor \frac{n}{3} \right\rfloor$ pièces…

5.a. Que fait – on ensuite ?

5.b. Ecrire précisément l'algorithme utilisé. Estimer sa complexité en nombre de pesées ; prouver sa terminaison et sa validité.

6. Le jeu de Taquin

Le jeu de taquin est une forme de puzzle, dans lequel le plateau de jeu est un carré découpé en $n \times n$ cases ; on peut déplacer ces cases grâce à l'une d'elle, laissée libre (on déplace l'un des carrés adjacents à cette case libre). On considérera qu'à l'origine, la case vide est située en bas à droite, et que le reste du plateau représente une figure. On mélange ensuite les cases, et le but est de reconstituer la figure initiale.

6.1. Ecrire une fonction permettant de modéliser le jeu à l'état initial.

6.2. Ecrire une fonction permettant de déterminer, à chaque stade, quelles sont les cases adjacentes à la case vide.

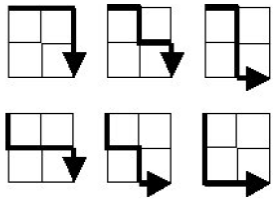
6.3. Ecrire une fonction permettant de déplacer un carré adjacent à la case vide, ce carré étant aléatoirement choisi.

6.4. Ecrire une fonction permettant de mélanger le jeu au hasard, sachant que chaque carré doit avoir quitté sa place initiale.

6.5. Ecrire une fonction permettant de partir d'un jeu mélangé, et de replacer les carrés à leur place initiale.

7. Dénombrement et mémoïzation

On dispose d'une grille composée de $n \times m$ cases. On part du coin supérieur gauche, et l'on ne peut se déplacer que vers la droite ou vers le bas. On cherche le nombre de chemins permettant d'atteindre le coin inférieur droit. Par exemple, pour une grille de 2×2 cases, il existe 6 chemins convenables :



On notera $\varphi(n, m)$ le nombre de solutions pour une grille de $n \times m$ cases.

7.a. Justifier les relations :

$$\varphi(n, m) = \begin{cases} 1 \text{ si } n = 0 \text{ ou } m = 0 \\ \varphi(n, m-1) + \varphi(n-1, m) \text{ si } m > 0 \text{ et } n > 0 \end{cases}.$$

Ecrire une fonction **Phi1** permettant de calculer $\varphi(n, m)$ sur ce principe. Prouver sa terminaison.

On note $C(n)$ le nombre d'appels de **Phi1** effectués lors de l'exécution de **Phi1(n,n)**.

Montrer que $C(n) \geq 2^n$.

7.b. Soient n, i, j tels que $n > i \geq j$.

Soit $A(n, i, j)$ le nombre de fois où **Phi1(i,j)** est appelée lors de l'exécution de **Phi1(n,n)**. Montrer que $A(n, i, j) \geq 2^{n-i}$. En déduire une nouvelle minoration plus performante de $C(n)$.

7.c. Soit $p \in \mathbb{N}$. On souhaite écrire une fonction **Phi2** permettant de calculer plus efficacement $\varphi(n, m)$ pour $n \leq p$ et $m \leq p$.

Pour cela, on utilise le principe de la mémoïzation : on crée un tableau de taille $(p+1) \times (p+1)$, où on place des 0 pour toutes les valeurs non calculées. A chaque appel à **Phi2(i,j)** , on commence par chercher dans le tableau des valeurs calculées, et :

- S'il y a dans le tableau quelque chose de strictement positif, on le renvoie.
- Sinon, on calcule récursivement $\varphi(i, j-1) + \varphi(i-1, j)$, on place la valeur obtenue dans le tableau de valeurs, et on renvoie le résultat.

Pour intialiser le tableau de valeurs, on pourra écrire :

tableau_valeurs=[[1]*p] + [[1] + [0]*p for i in range(p)]

Ecrire une telle fonction **Phi2** .

7.d. On note $D(n)$ le nombre d'appels de **Phi2** effectués lors de l'exécution de **Phi2(n,n)**. Montrer que $D(n) = o(C(n))$.

Préciser autant que possible ce résultat.