COMPTE RENDU : TRAVAIL INDIVIDUEL

Tri Par Insertion

MEZGHICHE AHMED

POUR LE 20/12/2023

1. Explication du tri par insertion

Le tri par insertion est un algorithme de tri simple et intuitif. Cela fonctionne en parcourant le tableau un par un et en insérant chaque élément dans la position correcte entre les éléments déjà triés.

A chaque itération, l'algorithme compare l'élément courant avec les éléments précédemment triés. Si l'élément actuel est plus petit, il sera déplacé vers la gauche jusqu'à atteindre la bonne position.

Ce processus de comparaison et de mouvement est répété pour chaque élément du tableau, formant une séquence progressivement triée.

Ainsi, voici à quoi ressemblerait un algorithme répondant aux critères du Tri par Insertion :

```
POUR k = 2 JUSQU'A longueur(tab) FAIRE
    m = k

    TANTQUE m > 1 ET tab[m] < tab[m-1] FAIRE

        temp = tab[m - 1]
        tab[m-1] = tab[m]
        tab[m] = temp

        SI m > 2 ALORS
            m = m - 1

    FINTANTQUE
FINPOUR

ECRIRE tab
```

Figure 1: Algorithme de tri en LARP

Malgré sa simplicité, le tri par insertion peut s'avérer inefficace lors du traitement de grandes quantités de données par rapport à d'autres algorithmes de tri plus sophistiqués.

Exemple/Illustration

Considérons le tableau non trié suivant : **[4, 2, 10, 8, 1].**

L'algorithme commence par considérer le deuxième élément, ici **2**, et le compare avec le premier élément **4**. Comme **2** est plus petit que **4**, ils sont échangés, donnant **[2, 4, 10, 8, 1].**

Ensuite, l'algorithme se déplace au troisième élément, **10**, et le compare avec les éléments triés précédents. Il reste à sa position car il est plus grand que **4**.

La même logique est appliquée à chaque élément suivant, triant progressivement le tableau. Après plusieurs itérations, le tableau devient trié : **[1, 2, 4, 8, 10].**

Chaque élément est inséré à sa position correcte parmi les éléments déjà triés, caractérisant le fonctionnement du tri par insertion.

2. Mesures de la complexité temporelle du Tri (En C et en Python) :

L'évaluation de la complexité temporelle d'un algorithme de tri par le biais de mesures du temps d'exécution pour différentes tailles de tableaux (représentées par N) permet ainsi d'appréhender son comportement face à des volumes croissants de données. Ces mesures fournissent des informations précieuses sur les performances de l'algorithme dans différentes situations, du meilleur au pire des cas, et contribuent à une compréhension plus approfondie de son efficacité et de son évolution en fonction de la taille des données traitées :

Algorithme (C) – Tri insertion

Taille(N)	100	200	1000	5000	7500	10000
$t_{mc}(N)$	0,0000s	0,0000s	0,0344s	0,1763s	0,2503s	0,3390s
$t_{m}(N)$	0,0000s	0,0000s	0,0355s	0,1776s	0,2933s	0,4170s
$t_{pc}(N)$	0,0032s	0,0094s	0,0430s	0,2030s	0,3336s	0,4746s

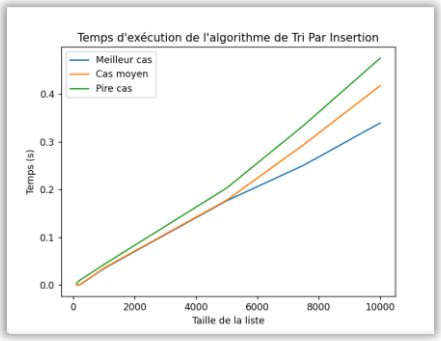

Figure 2: Graphique de complexité temporelle en C

Algorithme (Python) – Tri Insertion

Taille(N)	100	200	1000	5000	7500	10000
$t_{mc}(N)$	0,0000s	0,0038s	0,0040s	0,0280s	0,0482s	0,0692s
$t_{m}(N)$	0,0002s	0,0056s	0,0294s	0,5388s	1,3270s	2.0664s
$t_{pc}(N)$	0,0004s	0,0068s	0,0476s	0,9106s	2,1810s	3,6640s

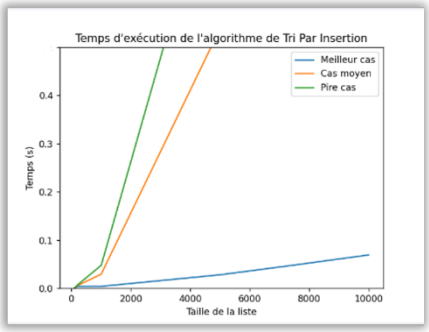

Figure 3: Graphique de complexité temporelle en Python

La comparaison des mesures du temps d'exécution du tri par insertion entre Python (interprété) et C (compilé) donne plusieurs observations. Premièrement, les mesures en C ont généralement des temps d'exécution plus rapides que les mesures en Python pour des tailles de données similaires. L'une des raisons à cela est que le code C étant compilé, il s'exécute plus rapidement que l'interprétation du code Python.

En termes de complexité temporelle, les deux langages n'affichent pas des tendances similaires, pour pouvoir distinguer correctement les différences de complexité, nous nous concentrerons principalement sur les mesures en Python. Le temps d'exécution optimal (tmc) est observé lorsque la taille des données est petite, démontrant l'efficacité du tri par insertion dans des situations favorables. Si le tableau est déjà trié, la complexité temporelle du tri par insertion est linéaire (O(n)). Cependant, dans le pire/moyen des cas, il peut atteindre une complexité quadratique ($O(n^2)$), surtout si le tableau est trié à l'envers. Cependant, à mesure que la taille des données augmente, le temps d'exécution augmente également, mettant en évidence la complexité quadratique du tri par insertion, surtout dans le pire des cas.

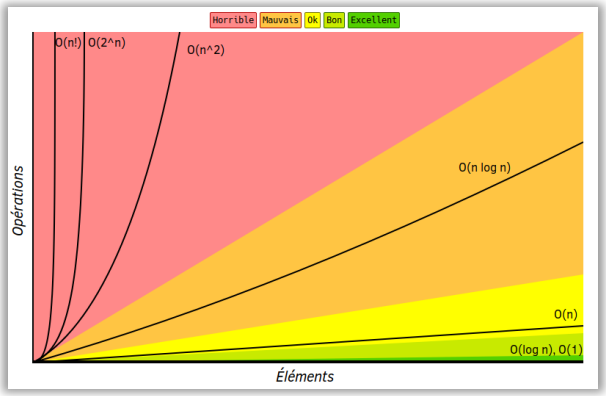

Figure 4 : Graphique présentant les plusieurs types de complexité

3. Point Annexe

Cas Moyen et Lecture de Tableau :

Afin de pouvoir manipuler les mêmes tableaux générés aléatoirement peu importe le langage mais aussi afin d'éviter des conditions inéquitables concernant les mesures de temps d'exécution, il fut nécessaire de générer chaque valeur (ligne par ligne) du tableau dans un fichier texte dans le but de permettre aux programmes (C et Python) de lire facilement chaque valeur et de l'attribuer à sa place respective dans le tableau, voici comment cela a été réalisé :

Générateur de tableau :

```c
int main() {

    // Initialisation du chemin d'accès
    const char *path = "C:\\Users\\but-info\\Documents\\BUT-INFO\\S1\\SAE\\SAE1.02\\TabAlea\\" ;

    // Initialisation de la variable n qui va générer un tableau aléatoire de taille N
    int n;

    // Demande à l'utilisateur la taille N du tableau
    printf("Nombres :");
    scanf("%d", &n);

    // Creation du nom du fichier avec le chemin d'accès + nom du fichier
    char filename[100];
    sprintf(filename, "%s%d.txt", path, n);


    // Ouverture du fichier
    FILE* fichier = fopen(filename, "w") ;

    int limite = n;
    srand( (unsigned int) time(NULL));
    printf("Tirage de 10 nombres aleatoires entre 0 et %d\n", limite);


    if (fichier != NULL){

        // Ecriture ligne par ligne de valeur générer  aléatoirement
        for (int i=0; i<n; i++) {
        // tirage d'un nombres aléatoire dans l'intervalle [0,limite[
            int nb = (int)(((float) rand()/ RAND_MAX) * limite);

            fprintf(fichier,"%d\n", nb);
        }
    }

            fclose(fichier);

    return 0;
```

Figure 5: Algorithme de générateur de tableau

Lecture et allocution de chaque valeur dans un tableau :

```python
# Définition de la fonction pour le cas Moyen
def MoyenCas(n, tab: list[int]):

    # Récupére le nombre entré par l'utilisateur et le transforme en chaine de charactère,
    # afin de lire le fichier n.txt corréspondant à ce que l'utilisateur a entré
    fichier = str(n)
    txt = ".txt"
    fichier += txt
    f = open(fichier, "r")

    # Lecture du fichier et range chaque valeur lue à sa place respective dans un tableau
    for i in range(n):
        content = f.readline()
        tab.append(int(content.split("\n")[0]))
    f.close()
    return tab
```

Figure 6: Fonction de lecture en Python

```c
int MoyenCas(int n)

    // Initialisation de la chaine de caractère nommée n.txt dont n est la valeur entrée par l'utilisateur

    char filename[1000];
    sprintf(filename, "%d.txt", n);


    int *tab = (int *) malloc(n * sizeof(int));

    // Récupére le nombre entré par l'utilisateur et le transforme en chaine de charactère,
    // afin de lire le fichier n.txt corréspondant à ce que l'utilisateur a entré
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Erreur lors de l'ouverture du fichier");
        return 1;  // Quitte le programme en cas d'erreur
    }

    int num_values = 0;  // Nombre de valeurs lues

    //  Allocution de chaque valeur lue à sa place respective dans le tableau
    while (fscanf(file, "%d,", &tab[num_values]) == 1) {
        num_values++;
    }
    fclose(file);  // Ferme le fichier après lecture
  return tab;
```

Figure 7: Fonction de lecture en C

Générateur Graphique :

Afin de pouvoir comparer correctement la complexité temporelle pour chacun des cas, nous avons également eu à programmer un générateur graphique en Python, voici comment cela a été réalisé :

```python
import pandas as pd
import matplotlib.pyplot as plt

# Meilleur cas
best_case = pd.DataFrame({
    "Taille(N)": [100, 200, 1000, 5000, 7500, 10000],
    "Temps (s)": [0.0000, 0.0038, 0.0040, 0.0280, 0.0482, 0.0692]
})

# Cas moyen
average_case = pd.DataFrame({
    "Taille(N)": [100, 200, 1000, 5000, 7500, 10000],
    "Temps (s)": [0.0002, 0.0056, 0.0294, 0.5388, 1.3270, 2.0664 ]
})

# Pire cas
worst_case = pd.DataFrame({
    "Taille(N)": [100, 200, 1000, 5000, 7500, 10000],
    "Temps (s)": [0.0004, 0.0068, 0.0476, 0.9106, 2.1810, 3.6640]
})

# Afficher les courbes
axes = plt.gca()
axes.plot(best_case["Taille(N)"], best_case["Temps (s)"], label="Meilleur cas")
axes.plot(average_case["Taille(N)"], average_case["Temps (s)"], label="Cas moyen")
axes.plot(worst_case["Taille(N)"], worst_case["Temps (s)"], label="Pire cas")

# Titre et étiquettes
plt.title("Temps d'exécution de l'algorithme de Tri Par Insertion")
plt.xlabel("Taille de la liste")
plt.ylabel("Temps (s)")

#Légende
plt.legend()

#Limiter les axes y
axes.set_ylim([0,0.5])
# Afficher le graphique
plt.show()
```

Figure 8: Programme générateur de tableau