INFO0250 - Programmation avancée

Projet 1 : Algorithmes de tri

Groupe 1: Martin RANDAXHE, Cyril RUSSE

Table des matières

1 Introduction

2 Algorithmes vus au cours

Dans cette section, nous allons analyser les différents algorithmes ayant été vus lors du cours théorique et implémenté dans le cadre de ce projet. Pour ce faire, nous allons présenter les résultats des calculs empiriques des temps d'exécution des 3 algorithmes concernés. Par la suite, nous comparerons l'évolution de ceux-ci par rapport aux complexités théoriques en fonction de la taille du tableau. Et finalement y apporter nos analyses vis à vis de l'ordre relatif de ces différents algorithmes.

2.1 Résultats empiriques de temps d'exécution

Voici, présenté dans la table ci-dessous, les valeurs de temps d'exécution des 3 algorithmes de tri vus au cours : InsertionSort, QuickSort et HeapSort.(Algorithmes respectivement implémentés dans les fichiers InsertionSort.c, QuickSort.c et HeapSort.C) Ces valeurs sont basées sur une moyenne de 10 temps d'execution pour des tableaux de taille 10^3, 10^4 et 10^5.

Type de tableau	aléatoire			croissant		
Taille	10^3	10^4	10^5	10^3	10^4	10^5
InsertionSort	0.001939	0.06914	6.65226	0.000017	0.000146	0.001397
QuickSort	0.000377	0.001536	0.044997	0.011423	0.33125	31.01124
HeapSort	0.000291	0.003594	0.009951	0.00008	0.000463	0.003292
PlaceSort	0.009042	0.479786	41.534067	0.00707	0.230783	22.825008

FIGURE 1 – Résultats empiriques de temps d'exécution

2.2 Analyse des tests en fonction des complexités théoriques des différents algorithmes

1. InsertionSort :
   — croissant : L'InsertionSort n'est pas réputé pour être un bon algorithme de tri pour des tableaux de grandes tailles mais il a pour seul avantage d'être très éfficace dans les cas où le tableau est déjà trié ou presque. Ce cas croissant correspond donc au meilleur cas de tableau pour cet algorithme où sa complexité théorique est $\Theta(n)$. Si nous observons les données de notre tableau, nous pouvons mettre en evidence le fait que lorsque qu'un facteur 10 sépare la taille des tableaux nous retrouvons effectivement ce facteur 10 au niveau du temps d'exécution dû à notre complexité $\Theta(n)$. Par exemple, pour des tableaux de taille 10^3 et 10^4, nous avons respectivement, en moyenne, 0.000146s et 0.001397s. 0.00146 se rapproche très fortement de 0.001397.
   — aléatoire : Le cas aléatoire correspond à un cas moyen pour cet algorithme qui est $\Theta(n^2)$. Nous observons bien entre les tableaux de tailles 10^3 à 10^4 ou de 10^4 à 10^5 un facteur 10^2 séparant nos moyennes temporelles.
2. QuickSort :
   — croissant : Le cas croissant est un des pires cas pour le QuickSort. En effet, celui-ci prend le dernier élément du tableau et l'utilise comme valeur de pivot. Malheureusement, le but de cet algorithme est de diviser le plus possible le tableau afin de réduire la tableau en plus petit tableau à gérer, hors dans le cas du cas croissant cette valeur de pivot sera toujours la valeur la plus grande par rapport aux autres et ne permettra pas de séparer le tableau en plusieurs tableaux qui permettrait d'avoir la meilleure complexité

possible pour des algorithmes de tri et notamment celui-ci $\Theta(n * log(n))$. Dans ce cas-ci, la complexité est $\Theta(n^2)$. Comme pour le cas aléatoire de l'InsertionSort, un facteur 10^2 sépare les moyennes de temps d'exécution entre des tableaux de taille 10 fois plus grand.

— aléatoire :

3. HeapSort :
   — croissant :
   — aléatoire :

3 PlaceSort

3.1 Pseudo-code

```
1         PLACE(A)
2             i=1
3             while(i<=A.length)
4                 m=1
5                 for j=1 to A.length
6                     if A[i]>A[j]
7                         m = m + 1
8                 if m!=i
9                     if A[i]==A[m]
10                        while (A[i]==A[m])
11                            m = m + 1
12                        swap(A[i], A[m])
13                        i = i + 1
14                    else
15                        swap(A[i], A[m])
16                else
17                i = i + 1
```