INFO0250 - Programmation avancée

Projet 1 : Algorithmes de tri

Groupe 1: Martin RANDAXHE, Cyril RUSSE

Table des matières

1 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.

1.1 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
RecSort	0.000813	0.002564	0.031851	0.004871	0.0127861	11.120391

FIGURE 1 – Résultats empiriques de temps d'exécution

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

1. InsertionSort :
   — Croissant : Nous retrouvons entre 10^3 et 10^4 un facteur 8,59 et entre 10^4 et 10^5 un facteur 9.57. En moyenne, cela nous fait $9.08 \simeq 10$. La complexité $\Theta(n)$ est vérifiée.
   — Aléatoire : Nous retrouvons entre 10^3 et 10^4 un facteur 35.66 et entre 10^4 et 10^5 un facteur 96.21. En moyenne, cela nous fait 65.93. Nous nous rapprochons bel et bien du facteur 10^2 attendu, d'autant plus notable entre les tableaux 10^4 et 10^5. La complexité $\Theta(n^2)$ est vérifiée.
2. QuickSort :
   — Croissant : Nous retrouvons entre 10^3 et 10^4 un facteur 93.62 et entre 10^4 et 10^5 un facteur 29. En moyenne, cela nous fait 61.31. Nous nous rapprochons bel et bien du facteur 10^2 attendu, d'autant plus notable entre les tableaux 10^4 et 10^5. La complexité $\Theta(n^2)$, étant donné que nous sommes dans le pire cas pour le QuickSort qui va toujours avoir sa valeur de pivot qui ne permettra pas de "diviser pour mieux régner", est vérifiée.
   — Aléatoire : Nous retrouvons entre 10^3 et 10^4 un facteur 4.07 et entre 10^4 et 10^5 un facteur 29.29. En moyenne, cela nous fait 16.68. Nous nous rapprochons bel et bien du facteur $10log(10)$ attendu, d'autant plus notable entre les tableaux 10^4 et 10^5. La complexité $\Theta(n * log(n))$ est vérifiée.
3. HeapSort :
   — Croissant : Nous retrouvons entre 10^3 et 10^4 un facteur 5.79 et entre 10^4 et 10^5 un facteur 7.11. En moyenne, cela nous fait 6.45. Théoriquement la complexité du heapsort est $\Theta(n * log(n))$, nous nous en rapprochons dans ce cas-ci, les résultats sont mêmes meilleurs que ceux attendus.

— Aléatoire : Nous retrouvons entre 10^3 et 10^4 un facteur 12.35 et entre 10^4 et 10^5 un facteur 2.77. En moyenne, cela nous fait 7.56. La complexité $\Theta(n * log(n))$ est vérifiée. Cet algorithme est de loin le plus performant quelque soit le cas, il respecte sa complexité théorique de $\Theta(n * log(n))$ quelque soit le cas et est même plus efficace que les autres algorithmes ayant cette même complexité.

1.3 Ordre relatif entre les différents algorithme

— Croissant : Pour le cas croissant il n'est pas difficile de donner un ordre entre ces algorithme car l'InsertionSort se trouve dans son meilleur cas qui est $\Theta(n)$ qui est le meilleur vu que les 2 autres sont dans leur meilleur cas $\Theta(n * log(n))$. Le cas croissant est le pire cas pour le QuickSort car sa valeur de pivot ne sera jamais à un endroit opportun pour séparé le tableau en plus petits tableaux. Sa complexité est dès lors $\Theta(n^2)$ et le HeapSort est quant à lui toujours $\Theta(n * log(n))$.
Nous avons donc dans l'ordre (Ordre confirmé par nos tests pour 10^5) :
InsertionSort(0.001397s)→ HeapSort(0.003292s)→ QuickSort(31.01124s).

— Aléatoire : Pour les 3 algorithmes, nous nous trouvons dans les cas moyens. Nous avons donc l'InsertionSort qui est le moins efficace avec une complexité théorique $\Theta(n^2)$.Ensuite, nous devons départager le QuickSort et le HeapSort qui sont dans ce cas-ci, tous les deux $\Theta(n * log(n))$. Par nos observations, nous pouvons déterminer que le HeapSort est légerement plus efficace. En effet, nous savons que ces 2 algorithmes sont les meilleurs algorithmes possible en terme d'algorithme de tri comparatif. En général, une préférence pour le quicksort qui se focalise un peu plus sur certaines parties du tableau tandis que le heapsort va "chercher" des valeurs un peu partout ce qui peu être moins avantageux d'un point de vue accès mémoire. Mais pour une analyse purement théorique au niveau des performances temporelles, le HeapSort est meilleur.
Nous avons donc dans l'ordre (Ordre confirmé par nos tests pour 10^5) :
HeapSort(0.009951s)→ QuickSort(0.044997s)→ InsertionSort(6.65226s).

2 PlaceSort

2.1 Pseudo-code

Voici le pseudo-code de la fonction Place. Celle-ci utilise A, le tableau à trier A[1...A.length]. La fonction swap, permute les valeurs du tableau qui lui sont données en argument. Nous allons utiliser m comme valeur permettant de définir la position de la valeur sur laquelle nous sommes actuellement qui sera notre à l'indice i.

```
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
```

La fonction Place(A) va placer l'élément A[i]($1 \leq i \leq$ A.length) à la position qu'il occuperait dans le tableau A s'il était trié. Soit m, l'emplacement correct de la valeur A[i] alors on permute A[i] et A[m]. On recommence en boucle avec la nouvelle valeur se trouvant en A[i] jusqu'à ce qu'il n'y ait pas de permutation possible où l'on va dès lors incrémenter i afin de trouver une nouvelle valeur qui n'est pas à sa place. Nous prenons également en compte le cas particulier où il existe plusieurs fois la même valeur dans le tableau à trier. Pour ce faire, si l'emplacement correct de la valeur est déjà occupé par un élément de cette même valeur, nous essayons avec la case d'après et ceci en boucle jusqu'à trouver un des cases suivantes ayant une valeur différente.