Université de Liège
Faculté des Sciences Appliquées

Rapport projet N°1

Dans le cadre du cours de structures des données et algorithmes

Année Académique 2016-2017

Moitroux Olivier

Étudiant en 2e Bac Ingénieur Civil
Le 24 mars 2017

1 Introduction

Ce rapport vise à familiariser l'étudiant avec certains algorithmes de tri. Il s'agira dans ce projet d'implémenter et/ou d'analyser pratiquement les performances des algorithmes "InsertionSort", "QuickSort", "RandomizedQuickSort", "HeapSort" et enfin "GSort" [1]. Ce rapport ne contiendra pas de code et se concentrera uniquement sur l'analyse de ces différents algorithmes de tri.

2 Analyse expérimentale

2.a. Temps d'exécution

Dans un premier temps, il nous est demandé de tester empiriquement, pour six tailles différentes de tableaux, le temps d'exécution des algorithmes ayant dû être implémentés. Le tableau ci-après reprend les résultats (en secondes) obtenus en prenant pour chaque test la moyenne de dix mesures. Notons que tous les tableaux de dimension N ont été générés aléatoirement et que le résultat est un tableau trié dans l'ordre croissant. La figure 1 n'est autre qu'une représentation graphique des résultats extrapolés.

N	InsertionSort (s)	QuickSort (s)	RandomizedQuickSort (s)	HeapSort (s)
10	$\simeq 0$	$\simeq 0$	$\simeq 0$	$\simeq 0$
100	$\simeq 0$	$\simeq 0$	$\simeq 0$	$\simeq 0$
1.000	0,00181818	$\simeq 0$	$\simeq 0$	0,00909091
10.000	0,11016528	0,00272727	0,00272727	0,00289992
100.000	11,09637866	0,06388429	0,06842975	0,03389181
1.000.000	1238,03 ($\rightarrow \infty$)	5,02894815	6,38503906	0,47423810

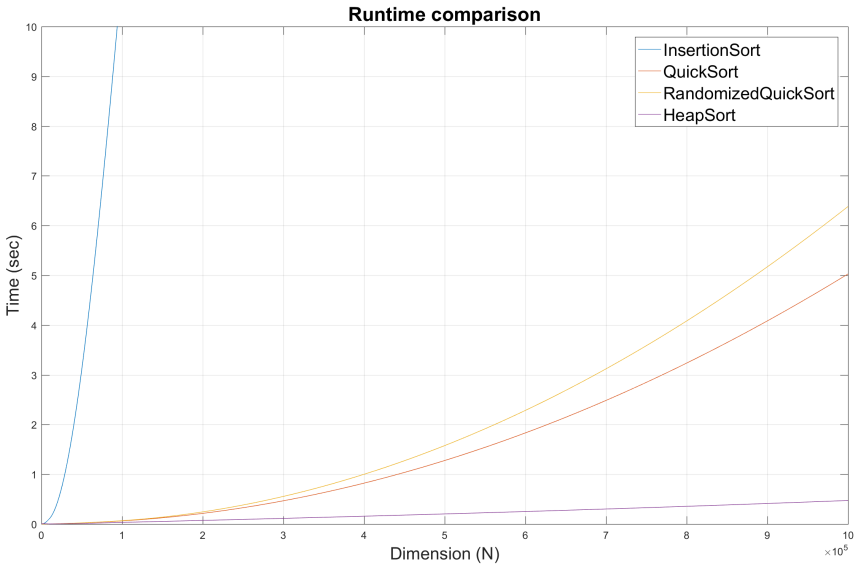


FIGURE 1 – Extrapolation des temps d'exécution

2.b. Analyse des résultats

Un rapide coup d'œil au tableau nous apprend que la rapidité d'exécution est liée forcément au choix de l'algorithme mais également à la taille du tableau. Ainsi, si l'on remarque aisément que HeapSort est

1. ©*Sort & Co.*

l'algorithme le plus rapide pour trier un tableau de taille conséquente, force est de constater qu'il se fait supplanter par QuickSort pour une dimension de tableau aux alentours de 10.000. Les deux autres restent en retrait avec un net écart pour InsertionSort qui n'est pas adapté pour le tri de tableau important. Cela peut s'expliquer par le calcul de la complexité théorique :

Commençons par QuickSort avec une affectation systématique du pivot au dernier élément du tableau. Tout comme InsertionSort, cet algorithme est dans le pire cas en $\theta(n^2)$. Ce cas correspond à un choix de pivot tel que la fonction partition doit diviser le tableau en deux sous-tableaux de taille 0 et n-1, c'est-à-dire quand le pivot est l'élément le plus grand du tableau traité. Toutefois, ce cas est relativement rare et l'algorithme peut être considéré comme en moyenne en $\theta(nlog(n))$. Il affiche de bonnes performances au niveau du cache et est caractérisé par une complexité en espace en O(n). Cela en fait un algorithme relativement performant pour le tri de tableau.

RandomizedQuickSort n'est autre qu'une adaptation de son semblable avec une légère nuance : le choix du pivot est désormais aléatoire. Cette technique est une parmi celles proposées pour réduire considérablement le risque de tomber sur un mauvais cas de partitionnement. Nous remarquons que cette technique n'améliore pas au final la vitesse d'exécution.

Quant à HeapSort, il possède une complexité constante en $\theta(nlog(n))$, ce qui en fait un algorithme plus adapté pour le tri de tableau conséquent. Par contre, il mobilise inutilement plus de ressources pour des tableaux de petite taille où des solutions plus simples seront préférées.

Enfin, InsertionSort est de loin l'algorithme le moins bien adapté pour des grandes tailles de tableau. Sa complexité quadratique le désavantage considérablement par rapport à ses concurrents. Il reste toutefois une solution correcte pour des petits tableaux (typiquement inférieurs à la centaine).

Terminons sur une remarque : seul InsertionSort n'est pas implémenté de manière récursive [2] et tous les algorithmes présentés sont en place, c'est-à-dire qu'ils modifient directement la structure en train d'être triée. Ils ne nécessitent ainsi qu'une quantité très limitée de mémoire supplémentaire.

Le tableau ci-dessous résume les notions de complexité évoquées. Les facteurs constants négligés peuvent avoir de l'importance pour des problèmes de petite taille.

Complexité	InsertionSort	QuickSort	RandomizedQuickSort	HeapSort
Pire cas	$\theta(n^2)$	$\theta(n^2)$	$\theta(n^2)$	$\theta(nlog(n))$
Moyenne	$\theta(n^2)$	$\theta(nlog(n))$	$\theta(nlog(n))$	$\theta(nlog(n))$
Meilleur cas	$\theta(n)$	$\theta(nlog(n))$	$\theta(nlog(n))$	$\theta(nlog(n))$

3 Analyse de GSort

GSort(A, p, r)
1 **if** $p >= r$
2 **then**
3 **return**
4 GSort$(A, p, r-1)$
5 **if** $A[r-1] > A[r]$
6 **then**
7 swap$(A[r-1], A[r])$
8 GSort$(A, p, r-1)$

3.a.

Le code fonctionne de la sorte :

Le premier appel, avec pour arguments $p \equiv$ début $= 0$ et $r \equiv$ fin $= dimension - 1$ va, dans un premier temps, appeler tous les $Gsort(0, r-1 \to 0)$. $Gsort(0, 0)$ va rentrer dans la première condition et s'arrêter. Ce

2. Chaque appel récursif nécessite en plus de mémoriser le contexte d'invocation.

n'est pas pour autant la fin de l'exécution du programme. En effet, $Gsort(0,1)$ rentre alors dans sa deuxième condition. Si $A[0] > A[1]$, un échange a lieu et l'on appelle à nouveau la fonction précédente ($Gsort(0,0)$) qui terminera sur return. Au final, dans les deux cas, la main revient à l'appelant de $Gsort(0,1)$, $Gsort(0,2)$ qui va enfin pouvoir comparer : $A[1] > A[2]$? Si ce n'est pas le cas, on passe son chemin et au tour de $G(0,3)$. Si, par contre, la condition est remplie, on échange A[1] et A[2] pour recommencer tout un processus avec $Gsort(0,1)$. Pourquoi ? Tout simplement parce que l'ancien A[2], devenu désormais A[1], pourrait très bien être encore supérieur aux éléments correspondant aux indices qui lui sont inférieurs. Et ainsi de suite.

On se rend bien compte que tous les appels aboutissent tôt ou tard à $G(0,0)$ qui se chargera, lors du dernier appel, de clôturer définitivement le tri. L'algorithme se termine donc bien avec un tableau trié, et ce, sans *overflow*.

Plus formellement, l'on peut prouver que l'algorithme est correct par preuve inductive. Après avoir posé deux cas de base, on considère que l'algorithme est correct pour le tri d'un tableau de taille n-1. Deux cas se présentent alors. Soit tout était bien trié et on termine, soit on vérifie la condition du second if et il faut permuter les deux éléments restants ainsi qu'appeler à nouveau un GSort sur n-1 qui, par hypothèse, est correct. Par induction, l'algorithme est donc correct.

3.b.

GSort est un algorithme qui est en place car il modifie directement la structure de données (ici le tableau) qu'il est en train de trier. Il est également considéré comme stable. Vu l'inégalité stricte, il conserve l'ordre relatif des éléments et ne peut permuter deux éléments égaux.

3.c.

Le meilleur cas correspond naturellement à celui où le tableau est déjà trié. Dans ce cas, le premier appel de GSort va en causer n-1 autres jusqu'à atteindre le *return* du G(0,0). Il y a ainsi n appels. Ensuite, pour chacun d'eux, le second if est évalué une seule fois étant donné que le tableau est trié. Il n'y a donc pas d'appel supplémentaire et l'on aura fait environ $n + n = 2n$ opérations.

Plus formellement, on a

$$T(n) = T(n-1) + c_1$$

étant donné que l'évaluation de la condition est en $O(1)$. Pour résoudre cette équation, réécrivons T(n-1) :

$$T(n-1) = T(n-2) + 2.c_2$$

En développant et en remplaçant successivement de la sorte les $T(n-i); i = 0, ..., n$ dans la formule de récurrence, on obtient

$$T(n) = n.c_1 + c_2$$

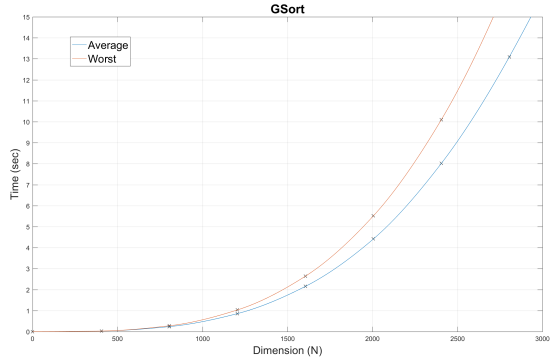
d'où une complexité linéaire dans le meilleur cas.

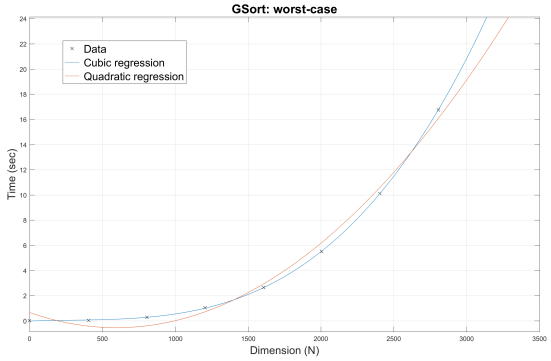
3.d.

Le pire cas correspond à celui où le tableau, ne contenant que des éléments différents, est trié dans l'ordre décroissant. Voici un tableau reprenant quelques échantillons du temps d'exécution nécessaire pour trier un tableau trié dans l'ordre décroissant stricte :

N	400	800	1200	1600	2000	2400	2800
Temps(s) :	0.03	0.28	1.035	2.6425	5.51625	10.103125	16.7565625

Après import des données dans Matlab©, différentes régressions nous apprennent que la mieux adaptée n'est autre que celle utilisant un polynôme du troisième degré.





Une façon intuitive de se convaincre que la complexité n'est pas de 2^N est de doubler la taille du tableau :

$$T(N_1) = 2^{N_1} = 2^{400} \Rightarrow t_1 = 0,03s$$
$$T(N_2) = 2^{(2*N_1)} = 2^{800} = 2^{400} * 2^{400} \Rightarrow t_2 = 0,28s \neq 0.03.2^{400}s$$

On a plutôt :

$$T(N_1) = N_1^3 = (400)^3 \Rightarrow t_1 = 0,03s$$
$$T(N_2) = T(2 * N_1) = (2 * 400)^3 = 8 * T(N_1) \Rightarrow t_2 \approx 8 * t_1 = 0,24s$$

3.e.

Le premier appel en engendre directement N-1 autres qui vont se charger de trier le tableau de 0 à N-2 avec N qui représente la taille du tableau. À chaque étape, il s'agit de prendre le minimum à droite et de le déplacer tout au bout à gauche dans la séquence déjà triée.

Ainsi, nous ne sommes plus dans le pire cas vu que cet algorithme repasse plusieurs fois par les mêmes étapes en recomparant des éléments qui l'avaient déjà été, mais contrairement à la première comparaison, ceux-ci sont déjà correctement ordonnés.

Or, rappelons-le, dans le meilleur cas (tableau trié), la complexité nous est donnée par $T_1(N) = c_1.N + c_2$. Désignons par T_2 la complexité de l'algorithme quand certains éléments sont déjà triés à gauche du tableau sauf le minimum qui suit à droite et par $T(N)$, la complexité générale de l'algorithme. Vu que les conditions et les affectations ont une complexité constante, on a

$$T_2(N) = T_1(N-1) + c_3 + T_2(N-1)$$

Soit en remplaçant,

$$T_2(N) = c_1.(N-1) + c_2 + c_3 + T_2(N-1)$$

Et vu que

$$\sum_{i=1}^{N}(N-i) = \frac{N.(N-1)}{2} \in \theta(N^2)$$

$T_2(N)$ est donc de la forme $T_2(N) = c_6.N^2$
Enfin,

$$T(N) = c_4 + T_3(N-1) + c_5 + T_2(N-1) = c_4 + T_3(N-1) + c_5 + c_6.N^2$$

Au final, puisque $\sum_{i=1}^{N}(i^2) = \frac{N.(N+1).(2N+1)}{6}$, la complexité dans le pire des cas est "seulement" de $\theta(N^3)$.

3.f.

Si cet algorithme est certes catastrophique à propos de son temps d'exécution pour des tableaux de taille importante, il peut toutefois être utile pour compléter un tableau quasi parfaitement ordonné en raison de sa complexité linéaire avantageuse dans le meilleur cas. Il sera aussi plus performant si la structure de données contient juste quelques éléments non triés qui sont contigus par deux.