INFO2050: Projet 1: Algorithme de tri

Julien Gustin, Mathias Carlisi

1 Analyse théorique

1.1 Invariant

- ☐ $\{P\}$ = "A est un tableau d'entiers de taille $A.length$"
  $\{Q\}$ = "Le tableau A est trié"
  $\{I\}$ = "Le sous tableau $A[1..i]$ contient les i premiers éléments de A triés, le sous tableau $A[j..A.length]$ est la partie du tableau de A qui n'est pas encore triée"

- ☐ $\{P\} i = 1$ && $j = 2 \{I\}$
  $\{I\}$ $\exists i, j,\ 1 \leq i < j \leq A.length,\ A[1..i],\ \forall k,\ 1 \leq k < i, A[k] \leq A[k+1]$
  Fonction de terminaison : $f = A.length - j + 1$
  $\{Q\} = \forall k,\ 1 \leq k < A.length,\ A[k] \leq A[k+1]$ && $j = A.length + 1$

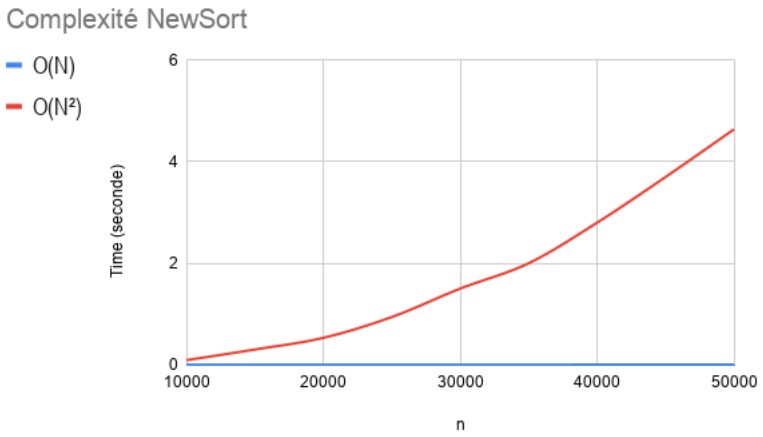
1.2 PseudoCode

NEW-SORT(A)

```
1  i = 1
2  for j = 2 to A.length
3        while A[j] ≤ A[j + 1] and j < A.length
4              j = j + 1
5        MERGE(A, 1, i, j)
6        i = j
```

1.3 Complexité

La complexité en temps, dans le meilleur des cas, est $\Theta(n)$ (si le tableau est pré-trié par ordre croissant) et dans le pire des cas $\Theta(n^2)$ (si le tableau est pré-trié dans un ordre décroissant) , où 'n' est la taille du tableau (array).



(Graphique testé avec des valeurs réelles)

1.4 Stabilité ?

NewSort est un tri stable en effet, grâce a "$A[j] \leq A[j+1]$" [1] , les valeurs égales n'interchangent pas de place et sont considérée comme déjà triée, de plus la fonction utilisée "*Merge*" étant aussi stable nous pouvons en conclure que **NewSort** est stable.

1.5 Complexité au pire des cas

1.

1 **while** $A[j] \leq A[j+1]$ and $j < A.length$ $\longrightarrow$ voir PseudoCode 1.2

2 Analyse expérimentale

2.1 Temps d'exécution sur des tableaux aléatoires

n	InsertionSort	QuickSort	HeapSort	MergeSort	NewSort
10^1	0,000015	0,000005	0,000006	0,000009	0,000005
10^2	0,000052	0,000035	0,000061	0,000054	0,000090
10^3	0,001055	0,000225	0,000959	0,000297	0,001841
10^4	0,060314	0,003429	0,003847	0,002229	0,112815
10^5	7,216459	0.036962	0,032527	0,017463	10,771157
10^6	768,522644	2.639571	0,353273	0,179450	1429,154037

L'**InsertionSort** et le **NewSort** sont les tris les plus lents des cinq, en effet avec leurs complexités moyennes de $\Theta(n^2)$, ces tris prennent un temps quadratique pour un tableau de taille 'n', cette complexité "élevée" découle du fait que, contrairement aux QuickSort, MergeSort ou HeapSort, ces tris requièrent (pour l'**InsertionSort**) de parcourir le tableau plusieurs fois pour insérer une valeur à sa bonne place et pour le **MergeSort**, de trié deux sous tableaux dont le premier commence de la première case jusqu'à la dernière précédemment triée, et le deuxième, de la case qui suit le premier sous tableau jusqu'à la dernière case où la suivante n'est pas triée par ordre croissant, et ainsi la fusion de ces deux sous-tableaux pré-triés crée un tableau trié. Parcourir plusieurs fois le tableau mène à une complexité de $\Theta(n^2)$.

L'algorithme qui suit est de type "diviser pour régner" (découper le problème initial en sous problèmes, résoudre ceux ci permettent de résoudre le problème initial), sa complexité moyenne est $\Theta(n\ log\ n)$, cependant, la lenteur de **QuickSort** comparé à MergeSort et HeapSort est dû, au pire des cas, à sa complexité quadratique, il procède ainsi par une méthode qui s'appelle "partition" et qui consiste à choisir un pivot que nous plaçons à la fin du sous tableau. Les éléments inférieurs à celui ci sont insérés au début de ce sous tableau, ensuite ce pivot est placé à la fin des éléments déplacés, cette méthode permet de trier le tableau rapidement, cependant, si le tableau est déjà trié en entrée, ce tri n'est pas vraiment efficace.

Les deux prochains algorithmes, **HeapSort** et **MergeSort**, sont tous deux extrêmement rapides avec une complexité, que ce soit dans le pire ou le meilleur des cas de $\Theta(n\ log\ n)$, qui est asymptotiquement optimale. Le **MergeSort** est de type "diviser pour régner", cet algorithme fonctionne par le principe que, à partir de deux tableaux triés, nous pouvons former un tableau triés, ainsi par récursivité, nous créons des sous-tableaux de plus en plus petits jusqu'à ce que le tableau ne contienne plus qu'un seul élément et par la remontée récursive, nous fusionnons ces tableaux qui deviennent de plus en plus grands à chaque remontée jusqu'à ce que le tableau soit complètement trié.
Le **HeapSort** fonctionne par arbre binaire.
Ce qui crée la légère différence en temps entre ces deux tris est la stabilité, en effet, le HeapSort ne préserve pas nécessairement l'ordre des éléments à valeurs identiques contrairement au MergeSort qui gagne ainsi en rapidité.

Cas	InsertionSort	QuickSort	HeapSort	MergeSort	NewSort
Meilleur	$\Theta(n)$	$\Theta(n\ log\ n)$	$\Theta(n\ log\ n)$	$\Theta(n\ log\ n)$	$\Theta(n)$
Pire	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n\ log\ n)$	$\Theta(n\ log\ n)$	$\Theta(n^2)$
Moyen	$\Theta(n^2)$	$\Theta(n\ log\ n)$	$\Theta(n\ log\ n)$	$\Theta(n\ log\ n)$	$\Theta(n^2)$
Stable	Oui	Non	Non	Oui	Oui

(si intervalle de rapport plus élévé = plus rapide)

3 Temps d'exécution sur des tableaux de blocs pré-triés

k	InsertionSort	QuickSort	HeapSort	MergeSort	NewSort
1	0,066221	0,066969	0,065319	0,001146	0,000132
20	0,005462	0,005605	0,004157	0,001470	0,000974
100	0,002183	0,002485	0,002158	0,002002	0,002533
500	0,001761	0,001638	0,001644	0,001810	0,006723
1000	0,001476	0,001626	0,000947	0,001876	0,013732
5000	0,048251	0,047346	0,047010	0,001124	0,058827