

INFO2050: Projet 1: Algorithme de tri

Julien GUSTIN, Mathias CARLISI

1 Analyse théorique

1.1 Invariant

- $\{P\} = "A \text{ est un tableau d'entiers de taille } A.length"$
 $\{Q\} = "Le \text{ tableau } A \text{ est trié}"$
 $\{I\} = "Le \text{ sous tableau } A[1..i] \text{ contient les } i \text{ premiers éléments de } A \text{ triés, le sous tableau } A[j..A.length] \text{ est la partie du tableau de } A \text{ qui n'est pas encore triée}"$
- $\{P\} i = 1 \ \&\& \ j = 2 \{I\}$
 $\{I\} \exists i, j, 1 \leq i < j \leq A.length, A[1..i], \forall k, 1 \leq k < i, A[k] \leq A[k+1]$
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

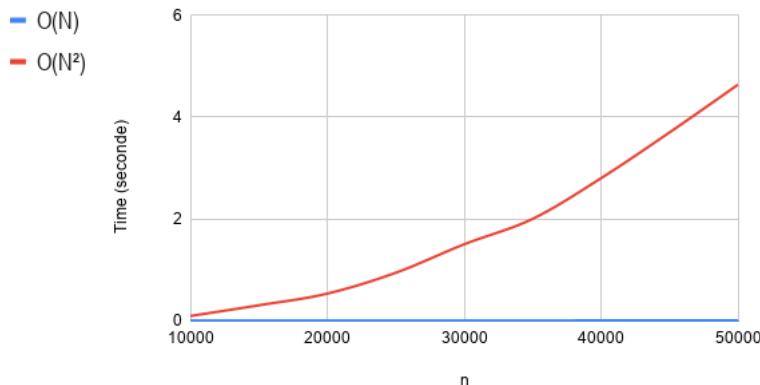
NEW-SORT(A)

```
1   $i = 1$ 
2  for  $j = 2$  to  $A.length$ 
3      while  $A[j] \leq A[j+1]$  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).

Complexité NewSort



(Graphique testé avec des valeurs réelles)

1.4 Stabilité ?

Le tri est stable parce que, grâce aux $A[j] \leq A[j+1]$, les valeurs égales n'interchangent pas de place, vu que *Merge* est lui aussi stable NewSort ne peut être que stable, ainsi le tri gagne en rapidité.

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

1.5 Complexité au pire des cas

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 trier 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, au pire des cas, est quadratique ce qui est la cause de sa "lenteur" comparé aux HeapSort et MergeSort, le **QuickSort** procède 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 placés 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 tout deux extrêmement rapides avec une complexité qui soit dans le pire ou meilleur des cas de $\Theta(n \log n)$ soit de complexité 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 trié ainsi par récursivité nous créons des sous-tableaux de plus en plus petits jusqu'à ce que le tableau ne contienne 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