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 //Parcours le tableau dans son entièreté
3       while A[j] ≤ A[j + 1] and j < A.length
4            j = j + 1 //Permet de trouver le prochain indice 'j' du tableau dont la valeur est triée
5       MERGE(A, 1, i, j) //Fusionne deux sous tableaux triés A[1..i] et A[i+1..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). En effet, si les valeurs sont triées par ordre croissant, nous itérons la boucle principale qu'une seule fois ainsi *Merge* (d'une complexité de $O(n)$) est appelé lui aussi une seule fois, au contraire si les valeurs sont triées par ordre décroissant *Merge* sera appelé 'n' fois parce que il n'existera aucun sous tableau trié, de taille plus grand que un, ainsi la boucle principale itéra 'n' fois.

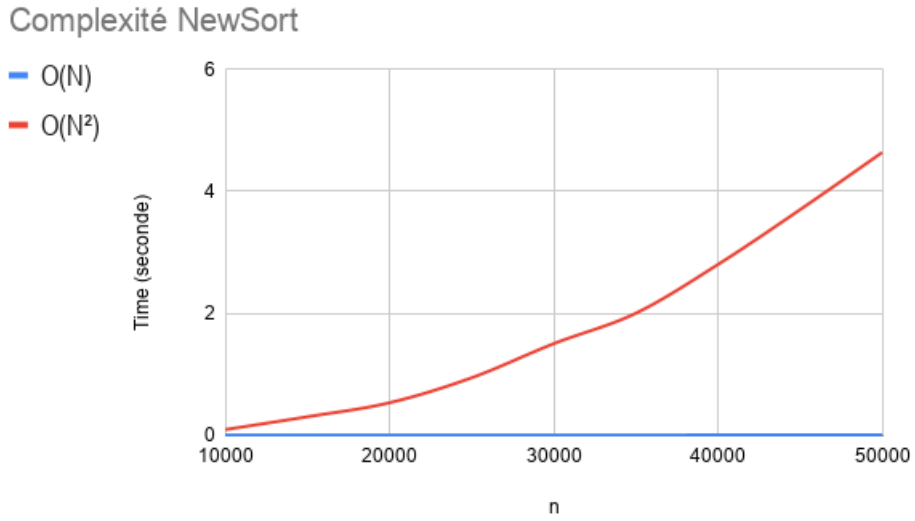
Preuve :

$T(n) \leq \sum_{i=1}^{n} i$

Puisque $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

On obtient $T(n) \leq \frac{n(n+1)}{2}$

ainsi nous obtenons $T(n) = O(n^2)$ puisque nous bornons supérieurement



(Graphique testé avec des valeurs réelles, où O(n) correspond à un tableau pré-trié et O(n^2), pré-trié dans un ordre décroissant)

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 considérées comme déjà triées, de plus vu que la fonction "*Merge*" est stable. Donc, nous pouvons en conclure que **NewSort** est lui aussi stable.

1.5 Complexité au pire des cas

Pour un tableau de taille 'n' constitué de $k(\leq n)$ blocs pré-triés de taille identique nous avons une complexité (au pire des cas) de $\Theta(k \cdot n)$, en effet, si k = n nous avons une complexité de $\Theta(n^2)$ et d'un autre coté si k = 1 nous avons $\Theta(n)$, respectivement nous avons bien le meilleur et le pire des cas décrit ci-dessus. Cela vient du fait que "*Merge*" a une complexité de $O(n)$ qui est appelée au maximum [2] 'k' fois vu que la boucle [3] secondaire permet de trouver l'indice du tableau auquel sa valeur à l'indice suivant est plus petit. Ainsi nous savons bien que dans le pire des cas *Merge* est appelé 'k' fois.

Preuve : $\sum_{i=1}^{k} i \cdot \frac{n}{k} <=> \frac{n}{k} \sum_{i=1}^{k} i$
Puisque $\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$
Nous avons : $\frac{n}{k} \cdot \frac{k(k+1)}{2} <=> \frac{n(k+1)}{2}$
Nous bornons $T(n)$ et ainsi $T(n) = O(n \cdot k)$

1.

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

2. Deux blocs pré-trié différents peuvent très bien être trié entre eux ainsi ne forme qu'un et même seul bloc trié

3.

```
1  while A[j] ≤ A[j + 1] and j < A.length
2        j = j + 1
```

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 de parcourir le tableau plusieurs fois afin d'y insérer une valeur à sa bonne place que ce soit par fusion ou insertion.

L'algorithme qui suit, **QuickSort** est de type "diviser pour régner" (découper le problème initial en sous problèmes, résoudre ceux ci permet de résoudre le problème initial), sa complexité moyenne est $\Theta(n\ log\ n)$, cependant, sa lenteur comparé à **MergeSort** et **HeapSort** est dû, aux pires de ses cas, avoir des valeurs triées par ordre croissant ou décroissant, le mène à avoir une 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 trié, 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. (L'absence de "pire cas" le rend "puissant" cependant dans des cas où la plage de nombres est plus élevées **QuickSort** est souvent préféré.)
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

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

k	InsertionSort	QuickSort	HeapSort	MergeSort	NewSort
1	0,000060	0,065099	0,002262	0,001146	0,000132
20	0,014462	0,005105	0,002415	0,001470	0,000974
100	0,015836	0,001485	0,001888	0,001702	0,002533
500	0,01761	0,001638	0,001644	0,001810	0,006723
1000	0,015769	0,001626	0,001752	0,001876	0,013732
5000	0,030251	0,047346	0,002043	0,001124	0,058827

Nous pouvons voir que pour un tableau pré-trié (k = 1) **InsertionSort** et le **NewSort** sont les plus rapides. En effet, pour ces deux tris, un tableau pré-trié est leur meilleur cas, cependant en pratique ce sont les plus lents : avoir un tableau déjà trié en entrée est quelque chose de très très rare et inutile, et en comparant les valeurs de ces deux tris avec les trois autres nous remarquons que pour des cas plus probables ils sont moins efficaces...

Le **QuickSort** est un algorithme très efficace si l'intervalle de rapport est plus élevé en effet vu que ce tri n'est pas stable avoir plusieurs valeurs les mêmes les unes à cotés des autres lui complique la tâche. De plus ses pires cas sont d'avoir comme pivot soit la valeur la plus petite soit la plus grande, autrement dit avoir un tableau trié par ordre croissant ou décroissant c'est pour cela que le quicksort atteint son efficacité maximale entre 'k' = 100 et 'k' = 1000 (plus rapide que les 4 autres), et au contraire est très peu efficace en 'k' = 1 ou 5000

Le **HeapSort** et **MergeSort** sont tout les deux très rapides et constants, c'est ce qui constitue leur force, ils n'ont pas de meilleur ou pire cas. Cependant vu que le HeapSort n'est pas stable, pour un tableau qui contient des répétitions il est préférable de choisir le **MergeSort**.

En bref, EST-CE QUE LE **NEWSORT** A UN RÉEL INTÉRÊT ?
Pour un tableau de taille assez petit (plus petit que 10^3) avec des blocs pré-triés en amont, ce tri peut avoir un intérêt. Cependant en situation réelle il sera préférable de se diriger vers le QuickSort ou le MergeSort.