PROJET 1

STRUCTURES DES DONNÉES ET ALGORITHMES

DAVINDER SINGH BAWEJA, MAHMOUD CHAIRI

Question 1

a) Voici les résultats en secondes obtenus empiriquement grâce à nos algorithmes :

Type de tableau	aléatoire			croissant			décroissant		
Taille	10^4	10^5	10^6	10^4	10^5	10^6	10^4	10^5	10^6
InsertionSort	0.057787	5.679562	612.952836	0.000056	0.000457	0.003982	0.117181	11.394522	1226.909264
QuickSort	0.000991	0.026486	37.46729	0.18539	17.897986	34378.10453	0.134191	13.38957	19484.95013
MergeSort	0.003574	0.033118	0.348247	0.004126	0.030260	0.302273	0.003547	0.027138	0.288907
HeapSort	0.001926	0.021542	0.259171	0.002041	0.018703	0.195636	0.001727	0.016254	0.193437

b) Pour InsertionSort, on constate que le temps de calcul pour trier un tableau aléatoire et décroissant augmente d'un facteur 100 lorsque la taille du tableau augmente d'un facteur 10, cela semble se comporter donc comme $O(n^2)$, ce qui correspond bien à la complexité théorique qui est $O(n^2)$ puisque nous sommes dans le pire cas.

Pour QuickSort, pour un tableau croissant et décroissant on constate que lorsque la taille du tableau augmente d'un facteur 10 le temps augmente d'un facteur 100, ce qui semble être $O(n^2)$. Cela correspond en effet au worst case théorique pour QuickSort.

Pour MergeSort on constate une croissance linéaire avec un certain facteur, d'où on peut déduire un comportement d'ordre $O(n.logn)$, ce qui correspond bien à la complexité théorique qui est $O(n.logn)$.

Pour HeapSort on a exactement les mêmes observations que pour MergeSort sachant que ces deux algorithmes ont la même complexité $O(n.logn)$.

Voici un tableau classant les différents algorithmes en fonction de leur rapidité avec joint leur complexité théorique :

Ordre	**Type de tableau**	**Complexité théorique**
1	HeapSort	$O(n.logn)$
2	MergeSort	$O(n.logn)$
3	InsertionSort	$O(n^2)$
4	QuickSort	$O(n^2)$

Question 2

a) Nous avons repéré essentiellement 3 erreurs dans le pseudocode. La première est que l'on a $\mathbf{i_l < q}$ et $\mathbf{i_r < r}$ alors qu'il faudrait avoir $\mathbf{i_l \leq q}$ et $\mathbf{i_r \leq r}$ sinon on ne parcourt pas l'entièreté du tableau. La seconde est que si l'élément à la position i_l est supérieur à l'élément à la position i_r, on incrémente i_l mais il faut aussi incrémenter i_r. La troisième est que dans ce même cas il faut aussi incrémenter q.

b) Voici le pseudocode de InPlaceMerge :

```
InPlaceMerge(A,p,q,r)
1 il = p
2 ir = q + 1
3 while il ≤ q  and ir  ≤ r
4       if A[il] ≤ A[ir]
5               il = il +1
6       else A[il] > A[ir]
7               tmp = A[ir]
8               for k = ir downto il +1
9                       A[k] = A[k-1]
10              A[il] = tmp
11              il = il + 1
12              ir = ir + 1
13              q = q + 1
```

c) Cet algorithme de tri est stable car il conserve l'ordre initial des clés identiques. On le voit ici dans la ligne 4 du pseudocode, si deux éléments comparés sont identiques le second ne prendra pas la place du premier, il attendra la comparaison suivante jusqu'à ce qu'il trouve un élément qui lui est supérieur pour prendre sa place.

d) Voici le tableau introduit à la question 1 avec une ligne en plus pour InPlaceSort :

Type de tableau	aléatoire			croissant			décroissant		
Taille	10^4	10^5	10^6	10^4	10^5	10^6	10^4	10^5	10^6
InsertionSort	0.057787	5.679562	612.952836	0.000056	0.000457	0.003982	0.117181	11.394522	1226.909264
QuickSort	0.000991	0.026486		0.182539	17.897986		0.134191	13.389957	
MergeSort	0.003574	0.033118	0.348247	0.004126	0.030260	0.302273	0.003547	0.027138	0.288907
HeapSort	0.001926	0.021542	0.259171	0.002041	0.018703	0.195636	0.001727	0.016254	0.193437
InPlaceSort	0.039035	3.606378	356.391858	0.000399	0.004051	0.032619	0.104960	9.556742	952.914528

Ici pour un tableau décroissant et un tableau aléatoire, lorsque la taille de ces tableaux augmente d'un facteur 10 le temps de calcul augmente d'un facteur 100. Cela semble donc se comporter comme $O(n^2)$. Pour un tableau croissant la croissance est linéaire avec un certain facteur et donc $O(n.logn)$.

e) Commençons par discuter la complexité en temps. Sachant qu'il s'agit d'un algorithme récursif on peut utiliser le master theorem. On a la forme générale :

$$T(n) = aT\left(\frac{n}{b}\right) + \boldsymbol{f(n)} \text{ avec a > 0, b > 1}$$

Commençons par établir les relations aux récurrences pour les 3 cas généraux qui sont le meilleur, le pire et le moyen cas. Dans ces cas on discutera de la complexité de InPlaceMerge (qui correspond à f(n) ici) sachant que le reste de l'équation n'est pas affectée par ces considérations, ensuite cela nous permettra grâce au master theorem de déduire la complexité de notre algorithme.

Pour le meilleur cas, le tableau est déjà trié et donc l'algorithme ne fait que parcourir le tableau sans jamais devoir le modifier, ce qui implique que la seconde boucle de l'algorithme n'est jamais sollicitée. Sachant qu'il ne parcourt que la moitié du tableau, il effectue $\frac{n}{2}$ opérations, on a alors une complexité d'ordre n. On a alors la relation aux récurrences :

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \boldsymbol{n}$$

Ce qu'on peut mettre sous la forme :

$$T(n) = 2T\left(\frac{n}{2}\right) + \boldsymbol{n}$$

Donc ici $log_2 2 = 1$, dès lors on a que $n = \Theta(n)$ et donc notre complexité en temps est ici $O(n.logn)$.

Pour le pire des cas et le cas moyen la seconde boucle va être sollicitée un certain nombre de fois, ce qui nous permet de dire en notation asymptotique que l'on aura une complexité pour InPlaceMerge $O(n^2)$. On obtient dès lors la relation aux récurrences :

$$T(n) = 2T\left(\frac{n}{2}\right) + \boldsymbol{n^2}$$

Donc ici $log_2 2 = 1$, dès lors on a que $n^2 = \Omega(n)$ et donc notre complexité en temps est ici $O(n^2)$.

Quant à la complexité en espace, sachant qu'on divise le tableau à chaque appel récursif en deux sous-tableau jusqu'à arriver à des éléments individuels, on en déduit qu'on a au maximum $log_2 n$ appels récursifs simultanément enregistrés sur le stack. A chaque appel récursif InPlaceMerge est appelé mais sa complexité en espace est $O(1)$, ce qui nous permet finalement de conclure que la complexité en espace d'InPlaceSort est $O(logn)$.

f) Sachant que la complexité en espace de cet algorithme de tri est $O(logn)$, on peut conclure que l'utilité pratique de ce dernier est de pouvoir économise de l'espace mémoire.