Structures de données et algorithmes : Projet 1

El Feniche Kamal (s171282) - Hernandez Capel Esteban (s174817)

Deuxième bachelier ingénieur civil 2018-2019

Université de Liège

21 mars 2019

1 Question 1

1.1 Analyse expérimentale du temps d'exécution

Les temps de calculs sont répertoriés dans le tableau suivant [1] :

Type de tableau	aléatoire			croissant			décroissant		
Taille du tableau	10^4	10^5	10^6	10^4	10^5	10^6	10^4	10^5	10^6
InsertionSort	0.1028	14.0345	∞	0.0000	0.015	0.0062	0.2066	25.1333	∞
QuickSort	0.0020	0.0519	3.4798	0.3167	∞	∞	0.2616	∞	∞
MergeSort	0.0047	0.0502	0.5560	0.0027	0.0439	0.4691	0.0047	0.0453	0.4020
HeapSort	0.0047	0.0387	0.4804	0.0031	0.0316	0.4182	0.0016	0.0328	0.3743

1.2 Analyse des données

Tout d'abord, on remarque pour des tableaux aléatoires que *InsertionSort* est le plus lent. En effet, il présente une complexité asymptotique $\Theta(n^2)$. De plus, *MergeSort* et *HeapSort* présentent presque les mêmes temps pour n'importe quelle taille de tableaux, car ils ont la même complexité asymptotique $\Theta(n \log n)$. Quant à *QuickSort*, on remarque que pour une taille de 10^4, il est le plus efficace. Néamoins, pour une taille de 10^6, il devient 10 fois plus lent, comparé aux *MergeSort* et *HeapSort*, bien qu'ils jouissent tous les trois d'une complexité $\Theta(n \log n)$, dans ce cas-ci.
Ensuite, on observe aisément pour des tableaux croissants que *InsertionSort* est l'algorithme le plus rapide. En effet, cela correspond à son meilleur cas. Quant à *QuickSort*, les tableaux croissants correspondent à son pire cas. En effet, dans ce cas il possède une complexité $\Theta(n^2)$, le rendant ainsi inefficace pour ce type de tableaux. Les observations pour *MergeSort* et *HeapSort* sont identiques que pour les tableaux aléatoires.
Enfin, pour des tableaux décroissants on observe que *QuickSort* et *InsertionSort* sont inefficaces puisqu'ils sont tous les deux, dans ce cas, de complexité $\Theta(n^2)$, et possèdent donc des temps conséquents. Une fois de plus, les observations pour *HeapSort* et *MergeSort* sont identiques, et ils sont dans ce cas-ci les plus performants avec une complexité $\Theta(n \log n)$. La lenteur de *QuickSort* pour des tableaux croissants et décroissants s'explique par le fait qu'il a été implémenté avec un pivot qui correspond à l'élément le plus à droite.

2 Question 2 : *InPlaceSort*

2.1 Correction du pseudo-code

Nous avons repéré 3 erreurs dans le pseudo-code.
Premièrement, vérifier que l'élément à la position $i_r - 1$ est plus grand que l'élément à la position i_r est incorrect. Cette condition est fausse et empêche le bon fonctionnement de l'algorithme.
Deuxièmement, i_l et i_r doivent être respectivement inférieurs ou égales à q et r, et non strictement inférieurs.
Troisièmement, dans le cas où l'élément à la position i_r est strictement plus petit que l'élément à la position i_l, il faut à la fois incrémenter i_l, i_r et q. En effet, si on rentre dans cette condition, on décale tout le tableau d'une position vers la droite depuis la position i_l. Dès lors, le tableau trié s'étend jusqu'à la position $q+1$. Et, pour passer à l'élément suivant on doit incrémenter i_r car on a décalé le sous-tableau $A[i_l...i_{r-1}]$ d'une position vers la droite.

1. Tests réalisés sur un processeur Intel Core i3-6006Li, 2.0 Ghz

2.2 Pseudo-code

```
InPlaceMerge(A,p,q,r)
i_r = q + 1
i_l = p
while i_l ≤ q and i_r ≤ r do
  if A[i_l] ≤ A[i_r] then
    i_l ++
  else
    value = A[i_r]
    for (k = i_r,  k ≠ i_l,  k − −) do
      A[k] = A[k − 1]
    end for
    A[i_l] = value
    i_l ++
    i_r ++
    q ++
  end if
end while
```

2.3 Stabilité de l'algorithme

On dit qu'un algorithme de tri est stable lorsqu'il ne modifie pas l'ordre initial des clés identiques. Ici, l'algorithme est bien stable. En effet, lorsqu'on effectue la comparaison $A[i_l] \leq A[i_r]$, si les deux clés sont égales, on incrémente i_l. Et si la clé à la position i_l (qui vient d'être incrémenté) est plus grand que $A[i_r]$, alors on insère $A[i_r]$ à la position i_l. Dès lors, l'ordre initial des clés est conservé puisque la deuxième clé sera toujours placée après la première clé.

2.4 Analyse expérmientale du temps d'exécution

Les temps de calculs sont répertoriés dans le tableau suivant :

Type de tableau	aléatoire			croissant			décroissant		
Taille du tableau	10^4	10^5	10^6	10^4	10^5	10^6	10^4	10^5	10^6
InsertionSort	0.1028	14.0345	∞	0.0000	0.015	0.0062	0.2066	25.1333	∞
QuickSort	0.0020	0.0519	3.4798	0.3167	∞	∞	0.2616	∞	∞
MergeSort	0.0047	0.0502	0.5560	0.0027	0.0439	0.4691	0.0047	0.0453	0.4020
HeapSort	0.0047	0.0387	0.4804	0.0031	0.0316	0.4182	0.0016	0.0328	0.3743
InPlaceSort	0.0073	0.9292	∞	0.0000	0.0007	0.0108	0.0141	2.0701	∞

Tout d'abord, on remarque que l'algorithme est bien plus rapide que tous les autres algorithmes pour un tableau trié. Néanmoins, on remarque que pour les tableaux aléatoires et décroissant, InPlaceSort est 10 voir 100 fois plus lent que le MergeSort et le HeapSort qui sont de complexités asymptotiques $\Theta(n \log n)$. Cela laisse suggérer que la complexité asymptotique de InPlaceSort est supérieur à celles de MergeSort et HeapSort. Néanmoins, il reste plus rapide que InsertionSort pour des tableaux de taille 10^4 et 10^5.
Pour des tableaux aléatoires et décroissant, lorsqu'on multiplie la taille du tableau par 10, on remarque que le temps de calcul est approximativement multiplié par 100. Cela suggère que InPlaceSort est $\Theta(n^2)$.

2.5 Etude de la complexité en temps et en espace

Soit $T(n)$ le temps nécessaire pour trier un tableau de taille n. L'équation récurrente est semblable à celle $MergeSort$, sauf que l'on a un terme en n^2. Cela s'explique par le fait que l'on parcourt les deux sous-tableaux et que si la condition est remplie, on doit décaler tout le sous-tableau $A[i_l...i_{r-1}]$ d'une position vers la droite. Dès lors, on a une boucle qui est $\Theta(n)$ qui contient une boucle qui est $\Theta(n)$,

donnant ainsi le terme en n^2.
Donc, le temps d'exécution pour trier un tableau de taille n est donné par :

$$T(n) = \begin{cases} c_0, & \text{si } n \leq 1 \\ 2T(\frac{n}{2}) + c_1 n^2, & \text{sinon} \end{cases} \tag{1}$$

On peut résoudre cette équation par téléscopage, c'est-à-dire que l'on applique l'équation à elle-même.

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + c_1 n^2 \\ &= 2(2T(\frac{n}{4}) + c_1 \frac{n^2}{2^2}) + c_1 n^2 \\ &= 2^2 T(\frac{n}{2^2}) + c_1 n^2 (1 + \frac{1}{2}) \\ &= 2^3 T(\frac{n}{2^3}) + c_1 n^2 (1 + \frac{1}{2} + \frac{1}{2^2}) \\ &= 2^i T(\frac{n}{2^i}) + c_1 n^2 (\sum_{k=0}^{i-1} \frac{1}{2^k}) \end{aligned} \tag{2}$$

En posant $i = \log_2 n$, on obtient :

$$\begin{aligned} T(n) &= 2^{\log_2 n} T(1) + c_1 n^2 (\sum_{k=0}^{(\log_2 n)-1} \frac{1}{2^k}) \\ &= nT(1) + 2c_2 n^2 (1 - (\frac{1}{2})^{\log_2 n}) \\ &= nT(1) + 2c_2 n^2 (1 - n^{\log_2 \frac{1}{2}}) \\ &= c_0 n + 2c_2 n^2 (1 - \frac{1}{n}) \\ &= c_0 n + 2c_2 n^2 - 2c_2 n \\ &\in \Theta(n^2) \end{aligned} \tag{3}$$

La complexité $\Theta(n^2)$ obtenue en résolvant l'équation corrobore nos observations. En effet, on peut constater sur le tableau que lorsque la taille du tableau est multiplié par 10, le temps de calcul est multiplié par 100, approximativement.
La complexité en espace de ce tri est $\mathcal{O}(\log n)$. En effet, à chaque appel récursif on divise la taille du tableau par deux. Par conséquent, on appelle la fonction un nombre de fois qui est nécessaire pour avoir une taille finale de 1 (cas de base). Dès lors, la fonction est appelée $\log_2 n$ fois. Donc, la complexité en espace de *InPlaceSort* est $\mathcal{O}(\log n)$.

2.6 Conclusion

On peut conclure que l'algorithme *InPlaceSort* est inefficace. En effet, bien qu'il soit en place et stable, il présente une complexité en temps $\Theta(n^2)$, ce qui est moins bien que *HeapSort* et *MergeSort* qui ont une complexité en temps $\Theta(n \log n)$. Malgré le fait que l'algorithme soit en place, il ne présente que peu d'intérêt puisqu'il existe d'autres algorithmes plus efficaces en terme de temps d'exécution.