Université de Liège

Projet 1: Algorithmes de tri

Structure de données et algorithmes

Maxime Meurisse
Valentin Vermeylen

2e année de Bachelier Ingénieur Civil
Année académique 2017-2018

1 Algorithmes vus au cours : analyse expérimentale

1.a Temps d'exécution moyen

Les algorithmes implémentés ont été testés sur des tableaux de différentes tailles, générés aléatoirement. Les temps reportés dans le tableau 1 sont des temps moyens établis sur base de 10 expériences.

N	`InsertionSort`	`QuickSort`	`MergeSort`	`HeapSort`
10	0,000 003	0,000 004	0,000 004	0,000 005
100	0,000 017	0,000 023	0,000 030	0,000 019
1000	0,001 576	0,000 226	0,000 292	0,000 395
10 000	0,117 356	0,002 145	0,002 234	0,003 090
100 000	11,675 235	0,058 665	0,023 543	0,037 855
1 000 000	1177,404 141	4,462 740	0,267 726	0,496 849

Tableau 1 – Temps d'exécution moyen, en secondes, des algorithmes de tri pour différentes tailles de tableaux.

1.b Analyse des résultats

Comparaison des algorithmes entre eux

Pour de petits tableaux, tous les algorithmes se valent. Il n'y a pas de différence significative entre leur temps d'exécution.

Pour des tableaux contenant 1000 éléments ou plus, l'algorithme InsertionSort devient beaucoup plus lent. Les autres algorithmes se valent toujours.

À partir de 100 000 éléments, des différences significatives apparaissent : QuickSort est devenu deux fois plus lent que MergeSort, et HeapSort, auparavant proche de MergeSort, est devenu un peu plus lent.

Pour des très grands tableaux (1 000 000 d'éléments), on constate que l'algorithme InsertionSort n'est plus du tout efficace. L'algorithme QuickSort, bien que son temps d'exécution soit encore raisonnable, perd également de sa rapidité. Le plus efficace est MergeSort, suivi par HeapSort, qui est deux fois plus lent, mais néanmoins dans le même ordre de grandeur.

Comparaison des algorithmes à leur complexité théorique

Tous les tableaux étant générés de manière aléatoire, on peut se baser sur la complexité moyenne pour une première comparaison.

Algorithme	Pire	Moyenne	Meilleure
`InsertionSort`	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
`QuickSort`	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$
`MergeSort`	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
`HeapSort`	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$

Tableau 2 – Complexité théorique des algorithmes vus au cours.

Tous les algorithmes ont une complexité moyenne de l'ordre de $n \log n$, excepté InsertionSort qui est de l'ordre de n^2. Les résultats obtenus confirment bien ces complexités : les temps d'exécution de QuickSort, MergeSort et HeapSort restent fort proches l'un de l'autre (sans oublier qu'on ne tient pas compte d'une possible constante multiplicative devant le $n \log n$), tandis que ceux de InsertionSort s'écartent de plus en plus de ces derniers.

2 Un nouvel algorithme de tri : `OtherSort`

2.a Pseudo-code de l'algorithme

Le pseudo-code du nouvel algorithme OtherSort est donné à la figure 1.

```
OtherSort (A, p, r)
 1  size = r − p + 1
 2  if size > 1
 3        if A[p] > A[r]
 4              tmp = A[q]
 5              A[q] = A[p]
 6              A[p] = tmp
 7        if size > 2
 8              t = size/3
 9              OtherSort (A, p, r − t)
10              OtherSort (A, p + t, r)
11              OtherSort (A, p, r − t)
```

Figure 1 – Pseudo-code de l'algorithme OtherSort.

2.b L'algorithme est-il correct ?

Une première solution consiste à tester concrètement l'algorithme. En affichant les tableaux avant et après son exécution, on constate que ceux-ci sont correctement triés. Cependant, il est impossible d'envisager tous les cas avec des tests concrets, c'est pourquoi une preuve formelle par récursion est fournie.

Preuve par récursion

Pour commencer, on montre que l'algorithme est correct pour les cas de base suivants : $n = 0, 1, 2$. Par « correct », on entend qu'il fournit en sortie la séquence triée qui lui a été fourni en entrée.

Si la taille du tableau est 0 ou 1, l'algorithme n'effectue aucune opération, ce dernier étant trivialement trié. Si le tableau est de taille 2, l'algorithme se contente de trier les deux éléments par une simple comparaison et un échange de position si nécessaire. Le tableau renvoyé est alors bien trié.

On suppose maintenant que l'algorithme est correct pour toute taille de tableau inférieure à n. On va montrer qu'il l'est également pour un tableau de taille n, et ce par logique.

L'algorithme va procéder à trois appels récursifs de lui-même pour des sous-tableaux ayant une taille $\lceil \frac{2n}{3} \rceil$. Ces tableaux ayant une taille strictement inférieure à n, l'algorithme va les trier correctement.

Il reste à montrer que ces trois tris successifs vont bien mener au tri du tableau initial.

Pour cela, on divise le tableau de départ en trois segments appelés A, B et C. Le premier appel va se charger de trier A et B, le second va trier B et C et le dernier va trier A et B une seconde fois (les tableaux étant mis à jour avant le tri suivant).

La définition de l'algorithme assure que la zone B est au moins aussi étendue que les deux zones extrêmes (voir sous-point ci-dessous). Dès lors, le premier appel va stocker dans B tous les éléments de A et B qui devront se trouver dans C à la fin de l'appel initial. Le second appel va placer ces éléments dans C. B étant au moins aussi étendu que C et A, on est certain que tous les éléments de C s'y trouveront, et correctement triés après cet appel récursif. B va alors emmagasiner tous les éléments de C qui doivent se trouver dans A, et les y placer lors du dernier appel. C étant trié lors du second appel, et A et B l'étant après le troisième, on est donc certain que l'algorithme a correctement trié le tableau de taille n.

L'algorithme étant correct pour des tableaux de taille 0, 1 et 2, et $mod(\frac{2n}{3}) \in \{0, 1, 2\}$, on est assuré que toutes les tailles de tableau supérieures se rapporteront bien à ces trois cas de base, et donc que l'algorithme est correct pour toute taille de tableau > 0.

Preuve que B est au moins autant étendu que A et C

On distingue 3 cas : la taille du tableau peut s'exprimer comme 3^k, $3^k + 1$ ou $3^k + 2$. La définition de l'algorithme assurant que, si le tableau initial n'est pas un multiple de 3, les sous-tableaux auront une taille arrondie à l'entier supérieur, permet d'écrire :

$$\begin{aligned} 3^k &\rightarrow |A| = k \quad ; \quad |B| = k \quad ; \quad |C| = k \\ 3^k + 1 &\rightarrow |A| = k \quad ; \quad |B| = k + 1 \quad ; \quad |C| = k \\ 3^k + 2 &\rightarrow |A| = k \quad ; \quad |B| = k + 2 \quad ; \quad |C| = k \end{aligned}$$

Avec |X| le nombre d'éléments dans la section X. On voit donc bien que B est au moins aussi étendu que A et C.

2.c L'algorithme est-il stable ? En place ?

Stabilité

La seule opération susceptible de modifier les positions d'éléments dans le tableau est l'échange du premier et dernier élément du sous-tableau traité, si nécessaire, à chaque appel de l'algorithme. L'inégalité présente dans la condition comparant ces deux éléments étant stricte, si deux éléments sont égaux, ils ne seront pas échangés.

L'algorithme est donc stable puisqu'il conserve l'ordre relatif des éléments égaux.

En place

Mise à part l'allocation mémoire du tableau de taille n, la seule opération demandant une allocation supplémentaire est l'échange entre le premier élément et le dernier élément, si nécessaire, du tableau. Au vu de la complexité en espace, discutée par après, on constate que, même dans le pire cas, l'algorithme n'utilisera pas plus de mémoire que celle utilisée pour stocker le tableau à trier.

L'algorithme est donc en place puisqu'il modifie directement le tableau qu'il est en train de trier, sans passer pas des structures de données intermédiaires.

2.d Complexité en temps : analyse expérimentale

Afin de comparer ce nouvel algorithme à ceux étudiés précédemment, une nouvelle colonne a été ajoutée au tableau 1.

Du à un temps de calcul trop long, les résultats pour un tableau de 100 000 et 1 000 000 d'éléments n'ont pas été fournis.

Ce nouvel algorithme est plus long que tous les autres, et cela peu importe la taille de tableau. Pour de petits tableaux ($N = 10$), l'écart n'est pas significatif. Dès une taille de

N	InsertionSort	QuickSort	MergeSort	HeapSort	OtherSort
10	0,000 003	0,000 004	0,000 004	0,000 005	0,000 004
100	0,000 017	0,000 023	0,000 030	0,000 019	0,001 731
1000	0,001 576	0,000 226	0,000 292	0,000 395	1,485 837
10 000	0,117 356	0,002 145	0,002 234	0,003 090	927,139 057
100 000	11,675 235	0,058 665	0,023 543	0,037 855	x
1 000 000	1177,404 141	4,462 740	0,267 726	0,496 849	x

Tableau 3 – Comparaison des temps d'exécution, en secondes, de l'algorithme OtherSort avec les autres algorithmes étudiés précédemment.

100 , l'écart devient important, et pour de grands tableaux (plus de 100 000), l'algorithme devient inutilisable.

2.e Complexité en temps et en espace : analyse théorique

Complexité en temps

Afin de prouver la complexité en temps, une preuve par induction est fournie.

Soit les 3 cas de base : le tableau est de taille 0 , le tableau est de taille 1 et le tableau est de taille 2 . Dans les trois cas, l'algorithme n'effectue que des comparaisons et échanges de valeurs ; des opérations dont le coût est une constante :

$$T(0) = C_0$$
$$T(1) = C_1$$
$$T(2) = C_2$$

Si le tableau est de taille supérieure ou égale à 3 , il sera partitionné et trié 3 fois, chaque tri triant deux tiers du tableau. En considérant toujours les opérations de comparaison comme ayant un coût constant, la relation de récurrence s'écrit :

$$T(n) = 3T\left(\frac{2n}{3}\right) + C_3$$

En remplaçant successivement le membre de droite par l'expression ci-dessus, on ob-

tient

$$\begin{aligned} T(n) &= 3T\left(\frac{2n}{3}\right) + C_3 \\ &= 3\left[3T\left(\frac{4n}{9}\right) + C_4\right] + C_3 \\ &= 3^2\left[3T\left(\frac{8n}{27}\right) + C_5\right] + C_4 + C_3 \\ &= ... \\ &= 3^k T\left[\left(\frac{2}{3}\right)^k n\right] + C \end{aligned}$$

Cette dernière expression étant générale, elle doit également convenir aux cas de base. En prenant l'argument de T égal à 1, on doit donc obtenir un coût constant. C'est le cas si

$$\left(\frac{2}{3}\right)^k n = 1 \Leftrightarrow k = \log_{\frac{2}{3}} \frac{1}{n}$$

En ayant la valeur de k, on trouve la complexité :

$$\begin{aligned} T(n) &= 3^{\log_{\frac{2}{3}} \frac{1}{n}} + C \\ &= 3^{-\log_{\frac{2}{3}} n} \\ &= 3^{\frac{\log_3 n}{\log_3 \frac{2}{3}}} \\ &= n^{\frac{\log 3}{\log \frac{2}{3}}} \\ &\approx n^{2.7} \end{aligned}$$

La complexité est donc $O(n^{2.7})$.

L'algorithme travaillant de la même façon, peu importe le tableau fourni, cette complexité est à la fois celle des meilleur, moyen et pire cas.

Complexité en espace

Pour un tableau de taille n, il faut allouer n emplacements mémoire. L'opération d'échange de deux éléments dans l'algorithme requiert l'allocation d'un emplacement mémoire supplémentaire (pour la variable `tmp`). Aucune autre structure de données tierce n'est utilisée.

La complexité en espace est donc $O(n+1) = O(n)$ (et est même $\Theta(n)$).

2.f Conclusion

Au vu des complexités obtenues et des comparaisons faites avec les autres algorithmes, OtherSort n'est pas un algorithme de tri efficace. Pour de petits tableaux, son inefficacité n'est pas significative, mais sa complexité étant exponentielle, son utilisation devient vite impossible.