Université de Liège

Structure de données et algorithmes

Projet 1 : Algorithmes de tri

BOUSTANI Mehdi
ALBASHITYALSHAIER Abdelkader

Table des matières

1. Qui est qui ?

1.1. Temps d'exécution et nombre de comparaisons

1.1.1. Tableau aléatoire

algo	10^3	10^4	10^5
0	0.0019354	0.1332347	14.205093
1	0.0002057	0.0022503	0.0225989
2	0.0001322	0.0015953	0.016435
3	0.002997	0.110497	10.7417
4	0.000472	0.004535	0.02308
5	0.000323	0.005663	0.03399
6	0.006209	0.263488	29.8067

Temps d'exécution pour un tableau aléatoire

algo	10^3	10^4	10^5
0	254884	25199759	2503234233
1	8718	120424	1536779
2	10466	142767	1813362
3	499500	49995000	4999950000
4	16870	235220	3019364
5	14024	236094	3724841
6	498870	49994180	4999945247

Nombre de comparaisons pour un tableau aléatoire

1.1.2. Tableau croissant

algo	10^3	10^4	10^5
0	0.0000054	0.0000437	0.0003952
1	0.0000897	0.0010835	0.0119552
2	0.0000509	0.0005143	0.0062049
3	0.003481	0.111178	10.78916
4	0.000439	0.004581	0.022246
5	0.000058	0.000728	0.005323
6	0.0000096	0.0000823	0.0007941

Temps d'exécution pour un tableau croissant

algo	10^3	10^4	10^5
0	999	9999	99999
1	5044	69008	853904
2	9451	125916	1572338
3	499500	49995000	4999950000
4	17583	244460	3112517
5	5457	75243	967146
6	999	9999	99999

Nombre de comparaisons pour un tableau croissant

1.1.3. Tableau décroissant

algo	10^3	10^4	10^5
0	0.0037964	0.2603298	28.6098232
1	0.0000995	0.001069	0.0124393
2	0.0000725	0.0007861	0.0100445
3	0.003856	0.116222	11.415536
4	0.000389	0.004152	0.0219931
5	0.000126	0.001728	0.010699
6	0.006053	0.266097	26.55633

Temps d'exécution pour un tableau décroissant

algo	10^3	10^4	10^5
0	499500	49995000	4999950000
1	4932	64608	815024
2	13135	193981	2620334
3	499500	49995000	4999950000
4	15965	226682	2926640
5	8550	120190	1533494
6	499500	49995000	4999950000

Nombre de comparaisons pour un tableau décroissant

1.2. Complexités des algorithmes

algo	Complexités			stable ?	en place ?	algorithme présumé
	meilleure cas	pire cas	cas moyen			
0	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓	Bubble sort
1	$\Theta(n\ log\ n)$	$\Theta(n\ log\ n)$	$\Theta(n\ log\ n)$	✓	×	Merge Sort
2	$\Theta(n\ log\ n)$	$\Theta(n^2)$	$\Theta(n\ log\ n)$	×	✓	Quick sort
3	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	×	✓	Selection sort
4	$\Theta(n\ log\ n)$	$\Theta(n\ log\ n)$	$\Theta(n\ log\ n)$	×	✓	Heap sort
5	$\Theta(n\ log\ n)$	$\mathcal{O}(n^2)$	$\Theta(n\ log\ n)$	×	?	Tri Inconnu
6	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓	Insertion sort

1.3. Analyse des algorithmes

Soit n la taille du tableau à trier.

Afin de déterminer si l'algorithme est stable, nous utilisons directement la fonction `isSortStable` développée pour la partie 2 du projet.

Pour les algorithmes connus, nous savons initialement que le Merge-Sort n'est pas en place et que tous les autres algorithmes de tri le sont (Insertion-Sort, Selection-Sort, Bubble-Sort, Quick-Sort et Heap-Sort). Cependant, pour l'algorithme inconnu, cela demande une analyse plus précise développée au point 1.4

1.3.1. Algorithme 0 et 6

Meilleur cas : Le meilleur cas correspond au cas où le tableau est déjà trié par ordre croissant. Dans ce cas, les temps de calculs (et le nombre de comparaisons) sont multipliés par 10 entre 10^3 et 10^4, 10^4 et 10^5. La complexité est donc linéaire : $\Theta(n)$. Ainsi, le nombre de comparaisons $= n - 1$, ce qui pourrait nous donner un indice sur le tri correspondant.

Pire cas : Le pire cas correspond au cas où le tableau est déjà trié par ordre décroissant. Dans ce cas, les temps de calculs (et le nombre de comparaisons) sont multipliés par 100 entre 10^3 et 10^4, 10^4 et 10^5. La complexité est donc quadratique : $\Theta(n^2)$. Ainsi, le nombre dcomparaisons $\approx \frac{n^2}{2}$

Cas moyen : La complexité du cas moyen est évidemment quadratique : $\Theta(n^2)$. En effet, les temps de calculs (et le nombre de comparaisons) sont multipliés par 100 entre 10^3 et 10^4, 10^4 et 10^5 lorsque le tableau est aléatoire.

Les algorithmes connus ayant ces complexités correspondent aux **Bubble sort** et **Insertion sort**

Bien que les complexités temporelles et la stabilité soient similaires entre le Bubble sort et l'Insertion sort, le nombre de comparaisons dans le cas moyen de l'algorithme 0 est significativement moins élevé que celui de l'algorithme 6, suggérant ainsi que l'algorithme 0 correspond au Bubble sort. Pour confirmer cette hypothèse, nous avons mené des tests sur des tableaux de taille 5 et 10, observant les indices des tableaux lors des comparaisons et des échanges, simulant ainsi chaque algorithme et l'état des tableaux après chaque swap.

Ces tests ont permis de distinguer plus clairement les deux algorithmes, confirmant que l'algorithme 0 correspond au **Bubble sort** et que l'algorithme 6 correspond à l'**Insertion sort**.

1.3.2. Algorithme 1

Meilleur cas : Le meilleur cas correspond au cas où le tableau est déjà trié par ordre croissant. Dans ce cas, les temps de calculs sont multiplié par 11 entre 10^3 et 10^4, et par environ 12 entre 10^4 et 10^5. La complexité est donc quasi-linéaire : $\Theta(n\ log\ n)$.

Pire cas : Pour cet algorithme, le pire cas ne correspond pas nécessairement au tableau décroissant. En effet, on le remarque facilement en comparant le nombre de comparaison entre un tableau croissant et décroissant. De plus, le **Merge Sort** atteint son pire cas lorsque les sous-tableaux gauche et droit impliqués dans l'étape de fusion contiennent des éléments alternés du tableau initial déjà trié. Pour le vérifier, il faut créer un algorithme créant un tableau donnant ce cas (assez compliqué). Ainsi, les temps de calculs étant même très proche dans les 3 types de tableaux, la complexité est quasi-linéaire : $\Theta(n\ log\ n)$.

Cas moyen : La complexité du cas moyen est donc aussi quasi-linéaire vu le comportement du nombre de comparaisons et du temps d'exécution dans le cas d'un tableau aléatoire : $\Theta(n\ log\ n)$.

L'algorithme stable connu ayant ces complexités correspond au **Merge sort**.

Nous avons pu éliminer les autres algorithmes ayant le même ordre de complexité, car le seul algorithme stable connu qui nous reste parmi les choix ayant une complexité similaire est le Merge sort.

1.3.3. Algorithme 2

Dans cet algorithme, les temps d'exécution sont assez similaires dans les 3 types de tableaux, se multipliant en moyenne de 12 lorsque la taille du tableau est multipliée par 10. Nous en avons déduit une complexité en temps quasi-linéaire : $\Theta(n\ log\ n)$. Parmi les choix possibles, Heap sort et le Tri inconnu partagent approximativement le même ordre de complexité. Cependant, étant donné que le pire cas du Quick sort se produit rarement, nous l'avons également inclus parmi les choix. Ainsi, nous avons trois options : **Heap sort**, **Quick sort**, et **Inconnu**.

Pour mieux identifier le tri correspondant à cet algorithme, nous avons analysé les indices des tableaux lors des comparaisons. Nous avons remarqué que l'algorithme commence par comparer trois éléments, puis le premier, le dernier, et l'élément au milieu du tableau, répétant ce processus plusieurs fois (voir image ci-dessous). Cette observation nous a dirigés vers le Quick sort, car nous savons que cet algorithme utilise la méthode `median-of-3` pour le choix du pivot, en prenant la médiane des trois éléments mentionnés. De plus, le meilleur cas se produit lorsque le tableau est déjà trié, où le pivot choisi est toujours l'élément du milieu, en utilisant la méthode `median-of-3`.

En conclusion, l'algorithme 2 correspond au **Quick sort**.

```
stsorting 100 2
Sorting a random array of size 100 with algorithm 2
i = 0 j = 49
i = 0 j = 99
i = 49 j = 99
i = 1 j = 98
i = 2 j = 98
```

1.3.4. Algorithme 3

Pour cet algorithme, que le tableau soit trié par ordre croissant ou décroissant ou même aléatoirement, la complexité reste quadratique : $\Theta(n^2)$. En effet, lorsque la taille du tableau est multipliée par 10, les temps de calculs et le nombre de comparaisons évoluent quadratiquement en se multipliant par 100. Ainsi, la complexité est identique dans le meilleur, pire et moyen cas. Dans le meilleur cas, même si le tableau est déjà presque trié, l'algorithme doit quand même parcourir tous les éléments pour s'assurer que chaque élément est à sa position correcte (mais moins de swap à faire).

L'algorithme connu ayant ces complexités correspond au **Selection-Sort**

1.3.5. Algorithme 4

Dans cet algorithme, les temps d'exécution sont comparables pour les 3 types de tableaux, avec un facteur multiplicatif qui varie de manière quasi-linéaire lorsque la taille du tableau est multipliée par 10. Jusqu'ici, nous avons identifié deux choix possibles : **Heap-Sort** et **Tri inconnu**. Pour identifier cet algorithme, nous l'avons comparé à l'algorithme 5, qui produit des résultats similaires.

Nous avons ensuite réalisé un test sur le nombre de swaps dans le cas où le tableau est croissant. Nous avons observé que l'algorithme 4 effectuait malgré tout quelques opérations de swap, tandis que l'algorithme 5 n'en effectuait aucune. Cette observation nous a orientés vers le Heap sort, qui, même lorsque le tableau est déjà trié, effectue quelques swaps pour maintenir la propriété d'ordre du tas dans `MAX_HEAPIFY`.

Tout cela pourrait nous affirmer que l'algorithme 4 correspond au **Heap-Sort**.

1.3.6. Algorithme 5

Dans cet algorithme, le nombre de comparaisons augmente de façon similaire dans les 3 types de tableaux, avec un facteur de multiplication variant de manière quasi-linéaire (13-16) lorsque la taille du tableau est multiplié par 10. Cette observation nous rapproche d'une complexité $\Theta(n\ log\ n)$ dans le meilleur cas, le pire cas et le cas moyen en adoptant l'hypothèse que le meilleur cas se produit lors d'un tableau croissant, et que le pire cas correspond au tableau trié par ordre décroissant.

En éliminant les autres algorithmes, il ne nous reste qu'un seul choix, ce qui nous permet de conclure que l'algorithme 5 correspond au **Tri Inconnu** et celui-ci est non-stable.

1.4. Efficacité de l'algorithme de tri

Pour les trois types de tableaux, cet algorithme de tri se révèle très efficace. Dans le meilleur cas, correspondant à un tableau croissant, sa complexité temporelle est $\Theta(n \log n)$. Le cas moyen, représenté par un tableau aléatoire, maintient également cette complexité. Cependant, le pire cas, bien que potentiellement être un tableau décroissant, pourrait conduire à une complexité temporelle quadratique, similaire au **Quick-Sort**. Nous ne sommes pas certains que le pire cas s'est produit lorsque nous avons effectué les tests de cet algorithme et nous ne pouvons pas confirmer que cet algorithme aurait une complexité de $\Theta(n\ log\ n)$ dans le pire cas. Ainsi, dans le pire cas, la complexité du **Tri inconnu** est $\mathcal{O}(n^2)$,

En outre, à mesure que la taille du tableau augmente, le facteur multiplicatif du temps d'exécution diminue. Par exemple, dans le cas moyen, le temps d'exécution est multiplié par

17 entre 10^3 et 10^4, tandis qu'il est multiplié par 6 entre 10^4 et 10^5. Cet algorithme se révèle donc particulièrement utile lorsque combiné à d'autres algorithmes de tri pour des ensembles de données volumineux.

2. Stabilisation d'un algorithme de tri

2.1. Implémentation de stableSort

Pour l'implémentation stableSort, nous avons utilisé la structure suivante :

```
1 typedef struct {
2    int key;
3    void *value;
4 }Element;
```

Cette structure est conçue pour stocker les indices originaux des éléments du tableau en tant que clés, afin de conserver l'ordre relatif des éléments du tableau.

Nous utilisons un tableau `global_array` qui donne accès aux éléments du tableau principal `array`, ainsi qu'un tableau de structure **Element** appelé `element_array`. Ce dernier sert à stocker les éléments du tableau principal sous forme d'entités clé-valeur grâce à la fonction `create_element_array` pour suivre l'ordre de leurs positions initiales. Ces deux tableaux sont stockés dans des variables globales.
Nous stockons aussi les fonctions `compare` et `swap` fournies en argument à la fonction stableSort dans des variables globales `compare_stablesort` et `swap_stablesort`.

Afin de réaliser notre tri de manière stable, nous utilisons la fonction `compare_new`. Cette fonction retourne un entier inférieur à 0 si l'élément à la position i est inférieur à l'élément à la position j, un entier supérieur à 0 si l'élément à la position i est supérieur à l'élément à la position j. Dans le cas où les deux éléments sont égaux, la fonction compare également les clés de ces 2 éléments à l'aide du tableau `element_array`. (et renvoie 0 si les 2 éléments sont égaux, ainsi que leurs clés).

Dans la fonction `swap_new`, après avoir échanger les éléments aux position i et j à l'aide de la fonction `swap_stablesort`, nous mettons également à jour le tableau `element_array`, en échangeant les éléments i et j pour suivre les positions initiales des éléments.

En appelant la fonction `sort`, le tri s'effectue de manière stable grâce aux différents outils que nous avons utilisés. Une fois le tri terminé, nous détruisons notre tableau de structure `element_array`.

2.2. Analyse de la complexité temporelle et en espace

La fonction **stableSort** ne fait qu'un seul parcours linéaire du tableau pour remplir le tableau des éléments `element_array`, mais maintient globalement la complexité en temps malgré une légère augmentation du nombre de comparaisons dans le cas d'un tableau aléatoire (où les comparaisons des clés des éléments égaux sont prises en compte).

Contrairement à la complexité en temps, **stableSort** augmente la complexité en espace des algorithmes en raison de l'allocation en mémoire du tableau de structures `element_array`, qui

stocke les positions initiales des éléments. Cette allocation ajoute une complexité en espace supplémentaire $\mathcal{O}(n)$ à l'algorithme.

2.3. Temps d'exécution et nombre de comparaisons pour les algorithmes non stables

2.3.1. Tableau aléatoire

algo	10^3	10^4	10^5
2	0.0002514	0.0028848	0.030578
3	0.0030649	0.2018764	19.16121175
4	0.0003535	0.0045218	0.0474303
5	0.0003031	0.0047065	0.0512689

Temps d'exécution pour un tableau aléatoire

algo	10^3	10^4	10^5
2	10504	143661	1846729
3	499618	49996052	4999959725
4	16909	235810	3024428
5	14134	237124	3734418

Nombre de comparaisons pour un tableau aléatoire

2.3.2. Tableau croissant

algo	10^3	10^4	10^5
2	0.000137	0.001371	0.0129142
3	0.003070	0.2043288	19.55681567
4	0.0003304	0.004115	0.0383998
5	0.0000918	0.0010544	0.0112224

Temps d'exécution pour un tableau croissant

algo	10^3	10^4	10^5
2	9451	125916	1572338
3	499500	49995000	4999950000
4	17583	244460	3112517
5	5457	75243	967146

Nombre de comparaisons pour un tableau croissant

2.3.3. Tableau décroissant

algo	10^3	10^4	10^5
2	0.0001744	0.0019374	0.01933009
3	0.0031007	0.200203	19.589387
4	0.000306272	0.003836	0.03671772
5	0.000152909	0.001821	0.0182809

Temps d'exécution pour un tableau décroissant

algo	10^3	10^4	10^5
2	13135	193981	2620334
3	499500	49995000	4999950000
4	15965	226682	2926640
5	8550	120190	1533494

Nombre de comparaisons pour un tableau décroissant

2.3.4. Comparaison des nouveaux résultats obtenus avec les précédents

Nous remarquons que les résultats sont assez proches, et que la version stable des algorithmes maintient bien la même complexité temporelle avec une légère différence au niveau du nombre de comparaisons [1] dans le cas d'un tableau aléatoire.

1. Nous avons remarqué que l'algorithme 2 effectue des comparaisons entre l'élément et lui même, et nous avons adapté notre implémentation pour ne pas comparer les clés lorsque ce cas tombe