Université de Liège

Structures des données et algorithmes

Rapport

Projet 1

Auteurs :
Louan Robert
Luca Heudt

Professeur :
Pr. Pierre Geurts

16 octobre 2023

Table des matières

1 Analyse théorique

1.1 Fonction findRun

Nous avons fait le choix de subdiviser la tâche de findRun en deux. Nous avons créé la fonction findSubArray qui cherche le sous-tableau trié de taille minimale $minSize$. Si un tableau est trié de manière décroissante, il est trouvé puis inversé.

```
findRun((A, start, minSize)
1  // Search for sorted subarray (ascending or descending)
2  sub = findSubArray(A, start, minSize)
3  if sub == −1
4        // If no subarray is found, sort from start to start + minSize
5        insertionSort(A[start], A[start + minSize])
6        return start + minSize
7  else
8        return sub
```

1.1.1 Fonction findSubArray

```
findSubArray(A, start, minSize)
 1  // Looking for ascending sorted array
 2  i = start
 3  while i < A.length and A[i] < A[i + 1]
 4        i = i + 1
 5        if i − start >= minSize
 6              return i
 7
 8  // Looking for descending sorted array
 9  j = start
10  while j < A.length and A[j] > A[j + 1]
11        j = j + 1
12        if j − start >= minSize
13              reverse(A, start, j)
14              return i
15  return -1 // No subarray found
```

1.1.2 Pseudo-code

Dans le pseudo-code, nous utilisons les même notations que celles vues en cours, mais nous considérons que les tableaux commencent à l'index 0.

1.2 Implémentation

1.2.1 Stabilité

Cet algorithme n'est pas stable à cause de la fonction reverse qui va inverser les nombres égaux qui sont un à la suite de l'autre.

Exemple de cette situation :

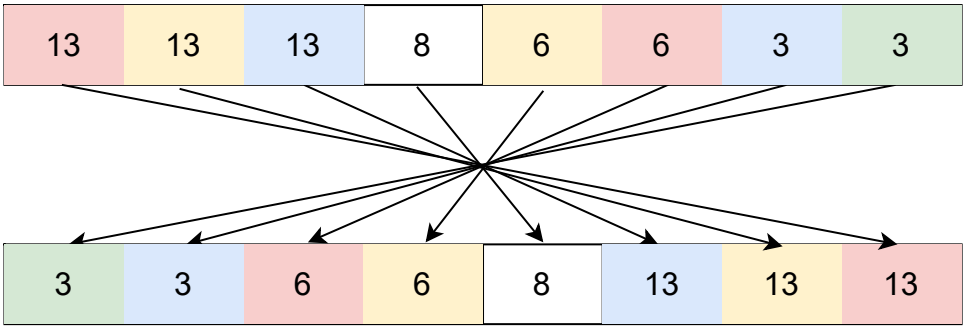

FIGURE 1 – Exemple d'action de l'algorithme REVERSE

1.2.2 En place ?

Cet algorithme n'est pas en place, étant donné qu'il est nécessaire de créer un nouveau tableau qui va contenir les valeurs triées.

1.3 Complexité

1.3.1 Meilleur cas

Le meilleur cas serait, bien entendu, le cas où le tableau serait déjà trié.

La complexité pour le meilleur cas est relativement simple à déterminer puisque la fonction FINDRUN va simplement parcourir tout le tableau, une seule fois. Ce qui nous donne une complexité $\theta(N)$.

1.3.2 Pire cas

Le pire cas serait le cas où le tableau ne serait pas du tout trié. C'est-à-dire qu'il ne contiendrait pas de sous-tableaux (de taille minimum `minSize`).

Dans ce cas, on ferait uniquement des appels à la fonction INSERTIONSORT, et donc on obtient la même complexité que celle-ci, à savoir $\theta(N^2)$

1.3.3 En espace

La complexité en espace de l'algorithme est $\theta(N)$, car on alloue 2 tableaux dont la taille dépend de N. La fonction f est $N + N/minSize + 2$, en admettant $c1 = 2$ et $c2 = 3$,
$c1.N \leq f(N) \leq c2.N$

2 Analyse empirique

2.1 Résultats

Toutes les valeurs sont le résultat d'une moyenne de 10 expériences. Un tableau presque trié est obtenu en faisant $0.01 \times N$ permutations de valeurs prises au hasard dans un tableau croissant de taille N.

2.1.1 Taille 10^4

Type de tableau	aléatoire		croissant	
	temps	nombre de comparaisons	temps	nombre de comparaisons
InsertionSort	0.059511	25113883.4	0.000054	9999.0
QuickSort	0.001113	152677.6	0.001737	153310.0
MergeSort	0.001164	120461.8	0.001749	69008.0
HeapSort	0.001699	255323.0	0.002290	264514
AdaptMergeSort	0.019917	1613827.0	0.000141	9999.0

Type de tableau	décroissant		presque trié	
InsertionSort	0.118395	49995000	0.001668	665614.8
QuickSort	0.000802	159558	0.000797	154550.1
MergeSort	0.000760	64608	0.000712	100801.1
HeapSort	0.001511	246706	0.001571	264348.9
AdaptMergeSort	0.000098	10000.0	0.008430	881061.0

2.1.2 Taille 10^5

Type de tableau	aléatoire		croissant	
	temps	nombre de comparaisons	temps	nombre de comparaisons
InsertionSort	6.581165	2501352863.1	0.000335	99999.0
QuickSort	0.013434	2027022.1	0.009018	1960666.0
MergeSort	0.013542	1536380.1	0.007231	853904.0
HeapSort	0.020514	3219572.1	0.018071	3313399.0
AdaptMergeSort	0.719246	152404434.0	0.000611	99999.0

Type de tableau	décroissant		presque trié	
InsertionSort	13.255042	4999950000.0	0.172990	66093620.2
QuickSort	0.009183	2074043.0	0.009103	2099880.4
MergeSort	0.007515	815024.0	0.007867	1339546.3
HeapSort	0.016724	3131749.0	0.017692	3311775.4
AdaptMergeSort	0.001283	100000.0	0.374585	84536170.0

2.1.3 Taille 10^6

Type de tableau	aléatoire		croissant	
	temps	nombre de comparaisons	temps	nombre de comparaisons
InsertionSort	340.764400	893667873.5	0.002000	999999.0
QuickSort	0.152064	24894509.8	0.097067	25614726.1
MergeSort	0.160667	18674281.5	0.080358	10066432.0
HeapSort	0.305650	38793973.0	0.227844	39736113.0
AdaptMergeSort	68.520348	15163510323	0.007009	999999.0

Type de tableau	décroissant		presque trié	
InsertionSort	681.529500	1783293664.0	0.199600	146121722.8
QuickSort	0.097420	24398325.1	0.099153	24259104.6
MergeSort	0.082384	9884992.0	0.095268	16622481.5
HeapSort	0.222576	37977135.0	0.238080	39726603.1
AdaptMergeSort	0.004100	1000000.0	37.579870	8471395834.0

2.2 Discussion des résultats

2.2.1 Temps de calcul, taille, et complexité

Aléatoire On remarque de façon directe que l'algorithme le moins bon est InsertionSort, son temps d'exécution augmente de façon quadratique par rapport à la taille du tableau. C'est tout à fait en lien avec sa complexité qui est $\theta(N^2)$.

Les algorithmes QuickSort, MergeSort, et HeapSort ont un temps d'exécution relativement similaire à l'exception de HeapSort pour un tableau de taille 10^6 où le temps d'exécution est presque double par rapport aux autres. Pour QuickSort et MergeSort, c'est un résultat tout à fait normal étant donné qu'ils ont une complexité moyenne $\theta(n \log n)$. C'est par contre relativement étonnant pour HeapSort qui a une complexité semblable aux deux précédents.

Croissant Dans ce cas InsertionSort, a contrario du paragraphe précédant, et AdaptiveMergeSort sont les algorithmes - se chevauchant - avec le meilleur temps d'exécution. Ceci s'explique par leur complexité dans le meilleur cas, qui est $\theta(n)$

Comme pour le paragraphe précédant, QuickSort et MergeSort sont environ 30 fois plus long en exécution, c'est logique étant donné leur complexité dans le meilleur cas moins bonne que InsertionSort. HeapSort, encore une fois, a un temps d'exécution doublé par rapport à QuickSort et MergeSort.

Décroissant De façon flagrante, InsertionSort est très mauvais, ce qui fait sens car un tableau trié à l'envers est le pire cas pour ce dernier. Ensuite, le MergeSort et le QuickSort sont relativement proche l'un de l'autre avec un léger avantage pour MergeSort, tant au nombre de comparaisons, qu'au temps d'exécution. On remarque également que le HeapSort est globalement plus long et plus coûteux (en nombre de comparaisons).

Le meilleur résultat pour les tableaux décroissant est sans surprise l'AdaptiveMergeSort. De part le fait qu'il ai un "système" de gestion des (sous-)tableaux décroissants, son nombre de comparaisons ainsi que le temps d'exécution sont énormément plus petit que ceux des autres algorithmes présentés ici.

Presque trié Ici encore, le MergeSort est le plus performant, suivi de près par le QuickSort. Le HeapSort et le InsertionSort sont, eux, relativement moins bon, mais le pire reste l'AdaptiveMergeSort.

2.3 Intérêt de l'AdaptiveMergeSort

Au final, l'AdaptiveMergeSort n'est pas très efficace, sauf dans le cas d'un tableau décroissant (voir justification dans la section 2.2.1). Par rapport aux autre algorithmes que l'InsertionSort il se débrouille très bien pour les tableaux déjà triés ce qui en ferai un bon algorithme dans une situation ou l'on sait que les données en input sont triées, sans savoir si c'est dans l'ordre croissant ou décroissant.