INFO0902 Projet 1 - Algorithmes de tri (Rapport)

DUCHEMIN Catherine, GERARD Manon

TOTAL POINTS

19 / 20

QUESTION 1

Analyse théorique 10 pts

1.1 Pseudo-code FindRun 4 / 4

✓ **- 0 pts** *Correct*

1.2 Stabilité et en place 1.5 / 2

✓ **- 0.5 pts** *Stabilité: justification incomplète ou incorrecte.*

1 Ok. Mais il pouvait être implémenté de manière stable.

1.3 Complexités 4 / 4

complexités en temps

✓ **- 0 pts** *Correct*

complexités en espace

✓ **- 0 pts** *Correct*

2 Attention, les meilleurs cas en temps et en espace ne sont pas toujours les mêmes. Il faut plutôt séparer l'analyse en temps et en espace.

QUESTION 2

Analyse empirique 10 pts

2.1 Temps de calcul 5 / 5

✓ **- 0 pts** *Correct*

2.2 Commentaires 3.5 / 4

Comparaison aux complexités théoriques.

✓ **- 0.5 pts** *Petites erreurs ou manques.*

Ordre relatif

✓ **- 0 pts** *Discussion correcte*

3 Pas à l'envers. C'est multiplié par 100.

4 Non, c'est multiplié par 100.

5 On ne voit pas de différence entre $$N$$ et $$N\log(N)$$ en général. L'algorithme n'est pas $$\Theta(N)$$.

2.3 Intérêt du tri par fusion adaptatif 1 / 1

✓ **- 0 pts** *Correct*

Analyse théorique

1.1 Donnez le pseudo-code de la fonction FINDRUN$(A, start, minSize)$.

FINDRUN$(A, start, minSize)$

```
1  i = start
2  if A[start] > A[start + 1]
3       while i < A.length and A[i] ≥ A[i + 1]
4            i = i + 1
5       for j = 0 to ⌊(i − start + 1)/2⌋
6            SWAP(A[start + j], A[i − j])
7       return ELONGATION(array, i, start, minSize)
8  else
9       while i < A.length and A[i] ≤ A[i + 1]
10           i = i + 1
11      return ELONGATION(array, i, start, minSize)
```

ELONGATION$(array, i, start, minSize)$

```
1  if i − start + 1 < minSize
2       ending = MIN(start + minSize − 1, A.length)
3       // Insert A[i + 1 .. ending] into the sorted sequence A[start .. i].
4       INSERTION-SORT(A, start, i, ending)
5       return ending
6  return i
```

INSERT-SORT$(A, start, i, ending)$

```
1  for k = i + 1 to ending
2       key = A[k]
3       j = k − 1
4       while j > 0 and A[j] > key
5            A[j + 1] = A[j]
6            j = j − 1
7       A[j + 1] = key
```

1.2 Est-ce que cet algorithme de tri peut être implémenté de manière stable ? Est-il en place ? Justifiez votre réponse.

Stable Non, cet algorithme de tri n'est pas stable. Cela est dû à la présence du swap quand on a un sous-tableau trié en ordre inverse.

Par exemple : FINDRUN$(A, 1, 4)$

A	3	2^1	2^2	1	devient A	1	2^2	2^1	3

En place Non, cet algorithme de tri n'est pas en place. L'utilisation de MERGESORT requiert un tableau supplémentaire de taille N. De plus, nous avons utilisé une liste liée afin de représenter une pile et ainsi qu'un tableau ce qui augmente la mémoire nécessaire. L'analyse de complexité en espace dans la partie suivante nous donne $\Theta(N)$, donc l'algorithme n'est pas en place.

1.3 Étudiez la complexité en temps et en espace de cet algorithme dans le pire et dans le meilleur cas en fonction de la taille du tableau N.

$$N = 2^k minSize \quad \Leftrightarrow \quad k = \log_2\left(\frac{N}{minSize}\right) \quad \rightarrow \quad \Theta(2^k) = \theta(N) \text{ et } \Theta(k) = \theta(\log N)$$

1.1 Pseudo-code FindRun **4 / 4**

✓ **- 0 pts** *Correct*

Analyse théorique

1.1 Donnez le pseudo-code de la fonction FINDRUN$(A, start, minSize)$.

FINDRUN$(A, start, minSize)$

```
1  i = start
2  if A[start] > A[start + 1]
3       while i < A.length and A[i] ≥ A[i + 1]
4            i = i + 1
5       for j = 0 to ⌊(i − start + 1)/2⌋
6            SWAP(A[start + j], A[i − j])
7       return ELONGATION(array, i, start, minSize)
8  else
9       while i < A.length and A[i] ≤ A[i + 1]
10           i = i + 1
11      return ELONGATION(array, i, start, minSize)
```

ELONGATION$(array, i, start, minSize)$

```
1  if i − start + 1 < minSize
2       ending = MIN(start + minSize − 1, A.length)
3       // Insert A[i + 1 .. ending] into the sorted sequence A[start .. i].
4       INSERTION-SORT(A, start, i, ending)
5       return ending
6  return i
```

INSERT-SORT$(A, start, i, ending)$

```
1  for k = i + 1 to ending
2       key = A[k]
3       j = k − 1
4       while j > 0 and A[j] > key
5            A[j + 1] = A[j]
6            j = j − 1
7       A[j + 1] = key
```

1.2 Est-ce que cet algorithme de tri peut être implémenté de manière stable ? Est-il en place ? Justifiez votre réponse.

Stable Non, cet algorithme de tri n'est pas stable. Cela est dû à la présence du swap quand on a un sous-tableau trié en ordre inverse.

Par exemple : FINDRUN$(A, 1, 4)$

A	3	2^1	2^2	1	devient A	1	2^2	2^1	3

En place Non, cet algorithme de tri n'est pas en place. L'utilisation de MERGESORT requiert un tableau supplémentaire de taille N. De plus, nous avons utilisé une liste liée afin de représenter une pile et ainsi qu'un tableau ce qui augmente la mémoire nécessaire. L'analyse de complexité en espace dans la partie suivante nous donne $\Theta(N)$, donc l'algorithme n'est pas en place.

1.3 Étudiez la complexité en temps et en espace de cet algorithme dans le pire et dans le meilleur cas en fonction de la taille du tableau N.

$$N = 2^k minSize \quad \Leftrightarrow \quad k = \log_2\left(\frac{N}{minSize}\right) \quad \rightarrow \quad \Theta(2^k) = \theta(N) \text{ et } \Theta(k) = \theta(\log N)$$

1.2 Stabilité et en place **1.5 / 2**

✓ **- 0.5 pts** *Stabilité: justification incomplète ou incorrecte.*

1 Ok. Mais il pouvait être implémenté de manière stable.

Analyse théorique

1.1 Donnez le pseudo-code de la fonction FINDRUN$(A, start, minSize)$.

FINDRUN$(A, start, minSize)$

```
1  i = start
2  if A[start] > A[start + 1]
3       while i < A.length and A[i] ≥ A[i + 1]
4            i = i + 1
5       for j = 0 to ⌊(i − start + 1)/2⌋
6            SWAP(A[start + j], A[i − j])
7       return ELONGATION(array, i, start, minSize)
8  else
9       while i < A.length and A[i] ≤ A[i + 1]
10           i = i + 1
11      return ELONGATION(array, i, start, minSize)
```

ELONGATION$(array, i, start, minSize)$

```
1  if i − start + 1 < minSize
2       ending = MIN(start + minSize − 1, A.length)
3       // Insert A[i + 1 .. ending] into the sorted sequence A[start .. i].
4       INSERTION-SORT(A, start, i, ending)
5       return ending
6  return i
```

INSERT-SORT$(A, start, i, ending)$

```
1  for k = i + 1 to ending
2       key = A[k]
3       j = k − 1
4       while j > 0 and A[j] > key
5            A[j + 1] = A[j]
6            j = j − 1
7       A[j + 1] = key
```

1.2 Est-ce que cet algorithme de tri peut être implémenté de manière stable ? Est-il en place ? Justifiez votre réponse.

Stable Non, cet algorithme de tri n'est pas stable. Cela est dû à la présence du swap quand on a un sous-tableau trié en ordre inverse.

Par exemple : FINDRUN$(A, 1, 4)$

A	3	2^1	2^2	1	devient A	1	2^2	2^1	3

En place Non, cet algorithme de tri n'est pas en place. L'utilisation de MERGESORT requiert un tableau supplémentaire de taille N. De plus, nous avons utilisé une liste liée afin de représenter une pile et ainsi qu'un tableau ce qui augmente la mémoire nécessaire. L'analyse de complexité en espace dans la partie suivante nous donne $\Theta(N)$, donc l'algorithme n'est pas en place.

1.3 Étudiez la complexité en temps et en espace de cet algorithme dans le pire et dans le meilleur cas en fonction de la taille du tableau N.

$$N = 2^k minSize \quad \Leftrightarrow \quad k = \log_2\left(\frac{N}{minSize}\right) \quad \rightarrow \quad \Theta(2^k) = \theta(N) \text{ et } \Theta(k) = \theta(\log N)$$

Meilleur cas le tableau est trié à l'endroit ou à l'envers.

- Complexité en temps : Dans ce cas, ce qui est avant la boucle `while` est négligeable par rapport à celle-ci. La première boucle `while` de la fonction SORT ne sera exécutée qu'une fois, mais il aura fallu tester la condition deux fois. Cette boucle `while` fait un appel à FINDRUN, STACKPUSH et MERGESTACK. La complexité de cette boucle `while` dépendra donc uniquement de la complexité des trois fonctions.

  - L'appel à FINDRUN rentre soit dans le `if` soit dans le `else`. Dans les deux cas la complexité sera la même. Le code vérifie la condition du `while` N fois. À l'intérieur du `while` il y a seulement une assignation de valeur. Si le tableau est trié de façon décroissante, il y aura $N/2$ SWAP, de complexité $\Theta(1)$. Ainsi, ils sont négligeables par rapport au `while` d'où le fait qu'on a deux meilleurs cas. Ensuite, la fonction renvoie un appel à ELONGATION.

    Calculons donc la complexité de ELONGATION dans ce cas. Le `if` est testé et ensuite la fonction renvoie i. Ainsi, dans ce cas, ELONGATION est $\Theta(1)$.

    Au final, dans le meilleur cas, FINRUN $\in \Theta(N)$
  - L'appel a STACKPUSH $\in O(1)$, comme nous l'avons vu en complément d'informatique.
  - L'appel MERGESTACK rentre uniquement dans le premier `if` et arrête la fonction. Ainsi, MERGESTACK $\in \Theta(1)$

    $\rightarrow$ première boucle `while` $\in \Theta(N)$

  Le code ne rentrera pas dans la deuxième boucle `while`. Et ensuite, fera un appel à STACKFREE qui est $O(1)$.

  Au final, dans le meilleur cas, notre algorithme est $\Theta(N)$.

- Complexité en espace : Il faut à présent s'intéresser à la mémoire allouée. Nous avons :
  - un tableau `aux` de taille N
  - une pile implémentée par liste liée. Nous avions vu en cours de complément d'informatique que sa complexité était $O(m)$ si la pile contient m éléments. Ici, la pile ne contiendra que 1 élément, elle est donc $O(1)$.
  - un tableau `tabIndex` de taille $maxLength = 2 \times \dfrac{N + minSize - 1}{minSize} = 2 \times \left\lceil \dfrac{N}{minSize} \right\rceil$

  Il faut aussi considérer les appels aux fonctions car cela prend de la place sur le "Stack".

  - Dans la première boucle `while`, on y rentrera qu'une fois. Dans celle-ci on fait un appel a FINDRUN, qui appel ELONGATION, qui a son tour appel INSERTSORT. Ensuite, il y a un appel à STACKPUSH et enfin un appel à MERGESTACK. La pile étant de taille 1, il n'y a pas d'appel à d'autres fonctions.
  - On ne rentre pas dans la deuxième boucle `while` donc il n'y a pas d'appels à considérer.

  Ainsi, on a une complexité en espace de $\Theta(N)$.

Pire cas aucune partie de tableau n'est triée et donc les $\frac{2^k minSize}{minSize} = 2^k$ appels à FINDRUN renvoie un sous-tableau de taille exactement $minSize$.

- Complexité en temps : Dans ce cas, ce qui est avant la boucle `while` est négligeable par rapport à celle-ci. La première boucle `while` de la fonction SORT sera exécutée 2^k fois, car ça correspond au nombre d'appels à FINDRUN. Cette boucle `while` fait un appel à FINDRUN, STACKPUSH et MERGESTACK. La complexité de cette boucle `while` sera donc 2^k fois la complexité des trois fonctions. Nous devons donc calculer ces complexités :

- Chaque appel à FINDRUN aura la même complexité. L'opération de SWAP dans le cas où les premières cases sont décroissantes pourront être négligés par rapport au reste. Le tableau étant considéré comme non trié, le code vérifie la condition du `while` 1 seule fois. À l'intérieur du `while` il y a seulement une assignation de valeur. Ensuite, la fonction renvoie un appel à ELONGATION. Dans ce dernier, la condition du `if` est vérifiée. Ensuite le `if` et le `else` ont la même complexité car il y a juste une assignation de valeur puis un appel à INSERTSORT et que nous considérons avoir un tableau de taille multiple de $minSize$.

  Nous devons donc nous intéresser à la complexité de INSERTSORT. Nous avons adapté l'algorithme vu en cours de INSERTION-SORT afin de ne pas répéter les i premières cases triées. On a considéré que la fonction était appelée avec $i = start+1$ et $ending = start+minSize-1$. La boucle `for` est donc exécutée $ending - (i+1) + 1$ fois ce qui correspond à $minSize - 2$. Dans le pire cas, il faudrait à chaque fois devoir ramener l'élément en première position, donc que cette suite de $minSize - 2$ éléments soit en ordre décroissant. Avec cela, pour le dernier élément, on devra être rentré $minSize - 1$ fois dans la boucle `while`. Ainsi, dans ce cas, INSERTSORT est $\Theta(minSize^2)$, donc ELONGATION et FINDRUN sont $\Theta(minSize^2)$.
- Chaque appel a STACKPUSH $\in O(1)$.
- Les 2^k appels à MERGESTACK eux dépendent de la taille de la pile. Étant donné que nos sous-tableaux sont tous de taille $minSize$, la condition $|A| \leq |B| + |C|$ ne sera jamais respectée alors que $|B| \leq |C|$ pourra l'être. MERGESTACK fait 3 pop puis appel MERGE quand c'est nécessaire et enfin fait 2 push. Sa complexité sera donc celle de MERGE. Nous avons vu en cours que sa complexité est $\Theta(n)$ où n représente la taille des 2 tableaux qu'il faut fusionner.

  Voici une représentation de ce qu'il se passe dans les différents appels a MERGESTACK combiné pour savoir en calculer la complexité :

A_1	B_2	C_3	D_4	...	...	$G_{2^{k-1}}$	H_{2^k}

  On fait $2^k/2$ fusions de 2 tableaux de taille combinée $2 \times minSize \rightarrow \Theta(2^k \times minSize)$

AB	CD	EF	GH

  On fait 2^{k-2} fusions de 2 tableaux de taille combinée $2 \times 2minSize \rightarrow \Theta(2^k \times minSize)$

  ...

ABCDEFGH

  À la $k$ième étape, on fait 2 fusions de 2 tableaux de taille combinée $2^k minSize \rightarrow \Theta(2^k \times minSize)$

  En sommant tout cela, on obtient $\Theta(k \times 2^k \times minSize) = \Theta(N \log N)$

  $\rightarrow$ première boucle `while` $\in \Theta(N \log N)$

Le code ne rentrera pas dans la deuxième boucle `while`. Et ensuite, fera un appel à STACKFREE qui est $O(1)$.

Au final, dans le pire cas, notre algorithme est $\Theta(NlogN)$

- Complexité en espace : Il faut à présent s'intéresser à la mémoire allouée. Nous avons :
  - un tableau `aux` de taille N
  - une pile implémentée par liste liée. Nous pouvons la négliger par rapport au tableau de taille N
  - un tableau `tabIndex` de taille $maxLength = 2 \times \dfrac{N + minSize - 1}{minSize} = 2 \times \left\lceil \dfrac{N}{minSize} \right\rceil$

  Il faut aussi considérer les appels aux fonctions car cela prend de la place sur le "Stack". On peut se simplifier la tâche en remarquant qu'on n'aura pas d'appel récursif donc cela sera négligeable.

Ainsi, on a une complexité en espace de $\Theta(N)$, comme dans le meilleur cas.

1.3 Complexités **4 / 4**

complexités en temps

✓ **- 0 pts** *Correct*

complexités en espace

✓ **- 0 pts** *Correct*

2 Attention, les meilleurs cas en temps et en espace ne sont pas toujours les mêmes. Il faut plutôt séparer l'analyse en temps et en espace.

Analyse empirique

2.1 Calculez empiriquement le temps d'exécution moyen et le nombre de comparaison de ces cinq algorithmes

Tableau	croissant					
Taille	10^4		10^5		10^6	
	Temps (s)	Comparaisons	Temps (s)	Comparaisons	Temps (s)	Comparaisons
InsertionSort	0,000029	9.999	0,000391	99.999	0,002139	999.999
QuickSort	0,210654	49.995.000	13,294472	4.999.950.000	SF	?
MergeSort	0,000638	69.008	0,005151	853.904	0,062548	10.066.432
HeapSort	0,002120	244.460	0,015380	3.112.517	0,158954	37.692.069
AMS	0,000030	10.000	0,000369	100.000	0,001917	1.000.000

Tableau	décroissant					
Taille	10^4		10^5		10^6	
	Temps (s)	Comparaisons	Temps (s)	Comparaisons	Temps (s)	Comparaisons
InsertionSort	0,141538	49.995.000	9,110239	4.999.950.000	>600	?
QuickSort	0,164464	49.995.000	10,748557	4.999.950.000	SF	?
MergeSort	0,000726	64.608	0,005095	815.024	0,062200	9.884.992
HeapSort	0,001880	226.682	0,015215	2.926.640	0,157121	36.001.436
AMS	0,000046	10.000	0,000252	100.000	0,002662	1.000.000

Tableau	aléatoire					
Taille	10^4		10^5		10^6	
	Temps (s)	Comparaisons	Temps (s)	Comparaisons	Temps (s)	Comparaisons
InsertionSort	0,073206	25.074.935	4,564025	2.499.583.754	>600	?
QuickSort	0,001312	156.349	0,010428	1.998.446	0,112885	24.779.170
MergeSort	0,001680	120.460	0,012965	1.536.470	0,149735	18.674.271
HeapSort	0,002437	235.330	0,019349	3.019.503	0,215957	36.794.254
AMS	0,001507	171.882	0,011849	2.054.355	0,133693	23.593.907

Tableau	Presque trié					
Taille	10^4		10^5		10^6	
	Temps (s)	Comparaisons	Temps (s)	Comparaisons	Temps (s)	Comparaisons
InsertionSort	0,001946	680.358	0,121140	65.098.205	11,940329	6.574.670.633
QuickSort	0,006357	1.761.072	0,053460	21.394.378	0,607992	282.595.776
MergeSort	0,000741	100.720	0,006095	1.342.266	0,070150	16.623.959
HeapSort	0,001916	244.300	0,015422	3.110.530	0,174196	37.696.460
AMS	0,000406	82.018	0,003952	1.174.823	0,046679	15.276.590

NB : Dans notre tableau, AMS signifie AdaptiveMergeSort, et SF signifie "Segmentation Fault".

2.2 Commentez ces résultats. Pour chaque type de tableau :

• comparer l'évolution des temps de calcul en fonction de la taille du tableau aux complexités théoriques

A l'exception des cas suivants, on observe une relation quasi-constante entre taille du tableau N et temps de calcul (car on est dans le cas $\frac{\Theta(N)}{N} = 1$) :

- InsertionSort : dans les cas des tableaux triés à l'endroit, à l'envers, et presque trié, le temps de calcul par case du tableau est multiplié par 10 lorsque la taille du tableau est multipliée par 10.

2.1 Temps de calcul **5 / 5**

✓ **- 0 pts** *Correct*

Analyse empirique

2.1 Calculez empiriquement le temps d'exécution moyen et le nombre de comparaison de ces cinq algorithmes

Tableau	croissant					
Taille	10^4		10^5		10^6	
	Temps (s)	Comparaisons	Temps (s)	Comparaisons	Temps (s)	Comparaisons
InsertionSort	0,000029	9.999	0,000391	99.999	0,002139	999.999
QuickSort	0,210654	49.995.000	13,294472	4.999.950.000	SF	?
MergeSort	0,000638	69.008	0,005151	853.904	0,062548	10.066.432
HeapSort	0,002120	244.460	0,015380	3.112.517	0,158954	37.692.069
AMS	0,000030	10.000	0,000369	100.000	0,001917	1.000.000

Tableau	décroissant					
Taille	10^4		10^5		10^6	
	Temps (s)	Comparaisons	Temps (s)	Comparaisons	Temps (s)	Comparaisons
InsertionSort	0,141538	49.995.000	9,110239	4.999.950.000	>600	?
QuickSort	0,164464	49.995.000	10,748557	4.999.950.000	SF	?
MergeSort	0,000726	64.608	0,005095	815.024	0,062200	9.884.992
HeapSort	0,001880	226.682	0,015215	2.926.640	0,157121	36.001.436
AMS	0,000046	10.000	0,000252	100.000	0,002662	1.000.000

Tableau	aléatoire					
Taille	10^4		10^5		10^6	
	Temps (s)	Comparaisons	Temps (s)	Comparaisons	Temps (s)	Comparaisons
InsertionSort	0,073206	25.074.935	4,564025	2.499.583.754	>600	?
QuickSort	0,001312	156.349	0,010428	1.998.446	0,112885	24.779.170
MergeSort	0,001680	120.460	0,012965	1.536.470	0,149735	18.674.271
HeapSort	0,002437	235.330	0,019349	3.019.503	0,215957	36.794.254
AMS	0,001507	171.882	0,011849	2.054.355	0,133693	23.593.907

Tableau	Presque trié					
Taille	10^4		10^5		10^6	
	Temps (s)	Comparaisons	Temps (s)	Comparaisons	Temps (s)	Comparaisons
InsertionSort	0,001946	680.358	0,121140	65.098.205	11,940329	6.574.670.633
QuickSort	0,006357	1.761.072	0,053460	21.394.378	0,607992	282.595.776
MergeSort	0,000741	100.720	0,006095	1.342.266	0,070150	16.623.959
HeapSort	0,001916	244.300	0,015422	3.110.530	0,174196	37.696.460
AMS	0,000406	82.018	0,003952	1.174.823	0,046679	15.276.590

NB : Dans notre tableau, AMS signifie AdaptiveMergeSort, et SF signifie "Segmentation Fault".

2.2 Commentez ces résultats. Pour chaque type de tableau :

• comparer l'évolution des temps de calcul en fonction de la taille du tableau aux complexités théoriques

A l'exception des cas suivants, on observe une relation quasi-constante entre taille du tableau N et temps de calcul (car on est dans le cas $\frac{\Theta(N)}{N} = 1$) :

- InsertionSort : dans les cas des tableaux triés à l'endroit, à l'envers, et presque trié, le temps de calcul par case du tableau est multiplié par 10 lorsque la taille du tableau est multipliée par 10.

- QuickSort : dans les cas des tableaux triés à l'endroit comme à l'envers, le temps de calcul par case du tableau est multiplié par 10 lorsque la taille du tableau est multipliée par 10.

On peut en tirer les conclusions suivantes :

- InsertionSort : les valeurs des temps de calcul correspondent bien aux complexités théoriques : $\Theta(N)$ dans le meilleur cas (tableau trié), $\Theta(N^2)$ dans les 3 autres cas.
- Quicksort : On a bien une complexité $\Theta(N^2)$ dans le pire cas, à savoir quand le tableau est trié à l'endroit comme à l'envers. En revanche, le temps de calcul est $\Theta(N)$ plutôt que $\Theta(NlogN)$ dans les autres cas.
- Mergesort et Heapsort : La complexité théorique est $\Theta(NlogN)$ dans tous les cas, mais les temps de calculs sont plutôt de l'ordre de $\Theta(N)$ donc meilleurs que prévus par la théorie.
- Adaptivemergesort : les temps de calcul sont tous de l'ordre de $\Theta(N)$, ce qui est logique pour les tableaux triés à l'endroit, à l'envers et presque triés.

• commentez l'ordre relatif des différents algorithmes en reliant vos observations aux complexités théoriques.

On peut classer les algorithmes dans l'ordre suivant, du plus performant en moyenne dans les 4 cas de tableaux rencontrés, jusqu'au moins performant :

- AdaptiveMergeSort : sa complexité en temps ($\Theta(N)$) est meilleure que celle du MergeSort ($\Theta(NlogN)$) dans les cas où le tableau est trié (à l'envers ou à l'endroit) ou presque trié. Dans le cas du tableau aléatoire, sa complexité est similaire au MergeSort ($\Theta(NlogN)$)
- MergeSort Sa complexité en temps est $\Theta(NlogN)$ dans tous les cas, et les temps de calculs sont meilleurs que pour les autres algorithmes à la seule exception du QuickSort lorsque le tableau est aléatoire.
- HeapSort : Avec sa complexité en temps qui est $\Theta(NlogN)$ pour tous les cas, et la similarité de ses temps de calculs pour les 4 types de tableaux lorsque ceux-ci sont de même taille, le HeapSort est un algorithme performant en moyenne.
- QuickSort : C'est le meilleur algorithme en matière temps de calculs dans le cas où le tableau est aléatoire (complexité en temps $\Theta(NlogN)$), par contre sa complexité de $\Theta(N^2)$ dans le pire cas en fait un algorithme peu efficace quand le tableau est (presque) trié.
- InsertionSort : C'est le meilleur algorithme en matière temps de calculs dans le cas où le tableau est déjà trié (complexité $\Theta(N)$), mais sa complexité de $\Theta(N^2)$ dans le cas moyen en fait un algorithme inefficace dans les autres cas.

2.3 Discutez de l'intérêt du tri par fusion adaptatif par rapport aux autres algorithmes.

Le tri par fusion adaptatif :

- est plus efficace que le MergeSort dans les cas où les tableaux sont triés ou presque, car on est dans le meilleur cas et sa complexité est $\Theta(N)$;
- est quasiment aussi efficace que le QuickSort lorsque le tableau est aléatoire, et plus efficace que le HeapSort d'après les calculs empiriques ;
- avec sa complexité en $\Theta(NlogN)$ dans le pire cas, il est beaucoup plus efficace que le InsertionSort dans les cas où le tableau est trié à l'envers, presque trié ou aléatoire, et aussi efficace lorsque le tableau est trié puisqu'il s'agit du meilleur cas pour ces 2 algorithmes.

Le tri par fusion adaptatif, sans être théoriquement le plus efficace dans tous les cas, a concrètement une efficacité qui le rend plus intéressant dans les cas pratiques que les 4 autres algorithmes testés.

2.2 Commentaires **3.5 / 4**

Comparaison aux complexités théoriques.

✓ **- 0.5 pts** *Petites erreurs ou manques.*

Ordre relatif

✓ **- 0 pts** *Discussion correcte*

3 Pas à l'envers. C'est multiplié par 100.

4 Non, c'est multiplié par 100.

5 On ne voit pas de différence entre $$N$$ et $$N\log(N)$$ en général. L'algorithme n'est pas $$\Theta(N)$$.

- QuickSort : dans les cas des tableaux triés à l'endroit comme à l'envers, le temps de calcul par case du tableau est multiplié par 10 lorsque la taille du tableau est multipliée par 10.

On peut en tirer les conclusions suivantes :

- InsertionSort : les valeurs des temps de calcul correspondent bien aux complexités théoriques : $\Theta(N)$ dans le meilleur cas (tableau trié), $\Theta(N^2)$ dans les 3 autres cas.
- Quicksort : On a bien une complexité $\Theta(N^2)$ dans le pire cas, à savoir quand le tableau est trié à l'endroit comme à l'envers. En revanche, le temps de calcul est $\Theta(N)$ plutôt que $\Theta(NlogN)$ dans les autres cas.
- Mergesort et Heapsort : La complexité théorique est $\Theta(NlogN)$ dans tous les cas, mais les temps de calculs sont plutôt de l'ordre de $\Theta(N)$ donc meilleurs que prévus par la théorie.
- Adaptivemergesort : les temps de calcul sont tous de l'ordre de $\Theta(N)$, ce qui est logique pour les tableaux triés à l'endroit, à l'envers et presque triés.

• commentez l'ordre relatif des différents algorithmes en reliant vos observations aux complexités théoriques.

On peut classer les algorithmes dans l'ordre suivant, du plus performant en moyenne dans les 4 cas de tableaux rencontrés, jusqu'au moins performant :

- AdaptiveMergeSort : sa complexité en temps ($\Theta(N)$) est meilleure que celle du MergeSort ($\Theta(NlogN)$) dans les cas où le tableau est trié (à l'envers ou à l'endroit) ou presque trié. Dans le cas du tableau aléatoire, sa complexité est similaire au MergeSort ($\Theta(NlogN)$)
- MergeSort Sa complexité en temps est $\Theta(NlogN)$ dans tous les cas, et les temps de calculs sont meilleurs que pour les autres algorithmes à la seule exception du QuickSort lorsque le tableau est aléatoire.
- HeapSort : Avec sa complexité en temps qui est $\Theta(NlogN)$ pour tous les cas, et la similarité de ses temps de calculs pour les 4 types de tableaux lorsque ceux-ci sont de même taille, le HeapSort est un algorithme performant en moyenne.
- QuickSort : C'est le meilleur algorithme en matière temps de calculs dans le cas où le tableau est aléatoire (complexité en temps $\Theta(NlogN)$), par contre sa complexité de $\Theta(N^2)$ dans le pire cas en fait un algorithme peu efficace quand le tableau est (presque) trié.
- InsertionSort : C'est le meilleur algorithme en matière temps de calculs dans le cas où le tableau est déjà trié (complexité $\Theta(N)$), mais sa complexité de $\Theta(N^2)$ dans le cas moyen en fait un algorithme inefficace dans les autres cas.

2.3 Discutez de l'intérêt du tri par fusion adaptatif par rapport aux autres algorithmes.

Le tri par fusion adaptatif :

- est plus efficace que le MergeSort dans les cas où les tableaux sont triés ou presque, car on est dans le meilleur cas et sa complexité est $\Theta(N)$;
- est quasiment aussi efficace que le QuickSort lorsque le tableau est aléatoire, et plus efficace que le HeapSort d'après les calculs empiriques ;
- avec sa complexité en $\Theta(NlogN)$ dans le pire cas, il est beaucoup plus efficace que le InsertionSort dans les cas où le tableau est trié à l'envers, presque trié ou aléatoire, et aussi efficace lorsque le tableau est trié puisqu'il s'agit du meilleur cas pour ces 2 algorithmes.

Le tri par fusion adaptatif, sans être théoriquement le plus efficace dans tous les cas, a concrètement une efficacité qui le rend plus intéressant dans les cas pratiques que les 4 autres algorithmes testés.

2.3 Intérêt du tri par fusion adaptatif **1 / 1**

✓ **- 0 pts** *Correct*

Université de Liège
Faculté des Sciences appliquées

Projet 1 : Algorithmes de tri

INFO0902-1 : Structure de données et algorithmes

Auteurs :
Catherine Duchemin s202046
Manon Gerard s201354

Professeur :
P. Geurts

3e année de Bachelier Ingénieur Civil

Année académique 2022 - 2023