Übungen zu Algorithmen
WSI für Informatik
Kaufmann/Kuckuk/Förster

Wintersemester 2021/22
Universität Tübingen
10. Januar 2022

Übungsblatt 10

Abgabe bis 12.01.2022, 8:00 Uhr

Besprechung: 17.01.2021, 16:15 Uhr

Übungspartner: Rune Schwarz

Aufgabe 1: Vergleiche verschiedener Sortieralgorithmen (je 1,5 Punkte) ★★☆☆☆

In der Vorlesung haben Sie mehrere Sortieralgorithmen kennengelernt. Im Folgenden sollen Sie näher auf die Unterschiede zwischen den Algorithmen eingehen.
Beim Bubblesort Algorithmus können Sie davon ausgehen, dass sich dieser bei jedem äußeren Schleifendurchlauf merkt, ob bereits zwei Elemente vertauscht wurden. Wurde über die ganze Liste iteriert ohne eine Vertauschung, so ist die Liste sortiert und der Algorithmus kann abgebrochen werden (siehe dazu auch das Präsenzübungsblatt).

a) Geben Sie die Konstruktion einer Liste, welche genau einmal jede Zahl zwischen 1 und n enthält, an, welche mit Bubblesort schneller sortiert werden kann als mit Heapsort.
Geben Sie die asymptotische Laufzeit beider Algorithmen für die Sortierung der konstruierten Liste an und begründen Sie ihre Antwort.

b) Geben Sie die Konstruktion einer Liste, welche genau einmal jede Zahl zwischen 1 und n enthält, an, welche mit Heapsort schneller sortiert werden kann als mit Bubblesort.
Geben Sie die asymptotische Laufzeit beider Algorithmen für die Sortierung der konstruierten Liste an und begründen Sie ihre Antwort.

Tipp: Beachten Sie, dass sowohl bei a) als auch b) eine Konstruktion gefragt ist, die für allgemeine n gilt!

c) Betrachten Sie erneut die Konstruktion einer Liste aus Aufgabenteil b). Sie möchten nun Quicksort nutzen um die Liste zu sortieren.
In der Vorlesung wurde die Wahl des Pivot-Elements unterschlagen, geben Sie eine Strategie zur Wahl des Pivot-Elements an, sodass ihr Algorithmus effizient die Liste sortiert.
Welche asymptotische Laufzeit ergibt sich daraus für die Sortierung der konstruierten Liste mit Quicksort?
Welche Laufzeit ergibt sich, wenn Sie dieselbe Strategie auf eine nach Aufgabenteil a) konstruierte Liste anwenden?
Begründen Sie ihre Antworten.

d) Wenden Sie auf die wie in Aufgabenteil a) sowie b) konstruierten Listen Bucketsort an. Gehen Sie davon aus, dass die Zahlen nach Binärdarstellung lexikographisch sortiert werden, in jedem Schritt wird also eine Binärstelle betrachtet und es gibt zwei Buckets.
Geben Sie an, welche asymptotische Laufzeit sich für die Sortierung nach Aufgabenteil a) bzw. b) konstruierten Listen ergeben.

1 Lösung Aufgabe 1

a)

Unsere Listenkonstruktion: eine sortierte Liste z.B. $L = [1, 2, 3, 4, ..., n]$.

Wie in der Aufgabenstellung schon beschrieben, kann man beim Bubble-Algorithmus davon ausgehen, dass nach dem ersten äußeren Schleifendurchlauf abgebrochen werden kann, sofern in der inneren Schleife nicht einmal der if-Fall betreten wurde. Dies wäre bei einer sortierten Liste der Fall. Dadurch dass wir nur die innere Schleife 1x durchlaufen würden, hätten wir eine Laufzeit von $O(n-1) = O(n)$.

Beim Heapsort benötigen wir zum Einfügen und Hochsteigen lassen $O(n \cdot log(n))$ und zur iterativen Minimumslöschung ebenfalls $O(n \cdot log(n))$. Demnach ist die Laufzeit von Heapsort bei unserer Listenkonstruktion weiterhin $O(n \cdot log(n))$. Und das sowohl im worst case als auch im best case.

Demnach ist für die oben genannte Liste der Bubblesort-Algorithmus effizienter.

b)

Unsere Listenkonstruktion: $L = [n, n-1, ..., 3, 2, 1]$.

Hier muss der Bubblesort-Algorithmus jeden einzelnen Schleifendurchlauf sich gönnen und somit nimmt er seine worst case-Laufzeit von $O(n^2)$ an.
Der Heapsort-Algorithmus benötigt seine gewohnte Laufzeit von $O(n \cdot log(n))$ und ist somit schneller als der Bubblesort-Algorithmus (mit dieser Listenkonstruktion).

c)
Wir würden als Pivot-Element immer das mittlere Element der derzeitigen/jeweiligen Liste nehmen. Dadurch bekommen wir als asymptotisches eine Laufzeit $O(n \cdot log(n))$.
Da in jeder Iteration des Quicksort-Algorithmuses jede Liste in jeweils 2 (nahezu) gleichgroße Listen unterteilt wird (da die ausgangsliste bereits absteigend (vllt auch aufsteigend je nach definietion) sortiert war, ist das Element an der mittleren Stelle jeder Teilliste auch das mittlere Element aller Listenelemente, und somit das optimale Pivot-Element) da die innere Itteration somit minimal ($log(n)$) ausgeführt werden, womit insgesamt ein asymtotisches Laufzeitverhalten von $O(n \cdot log(n))$.

Bei einer bereits in richtiger Reihnfolge sortierten Liste ist die asymptotische Laufzeit des Quick-Sort Algorhitmus bei selber Strategie identisch, analog den bereits oben genannten Gründen.

d)

Da die Listenkonstruktionen von a) und b) die Zahlen 1 bis n enthalten, gibt es unterschiedlich lange Binärdarstellungen der Listeneinträge (für $n > 1$) wodurch die Einträge anhand ihrer Länge in Binärschreibweise vorsortiert werden (weshalb beide Listenkonstruktionen vollständig den Algorhytmus durchlaufen und ein identisch asymtotisches Verhalten haben).
Laut Vorlesung für beide Listenkonstruktionen die asymptotische Laufzeit des Bucketsort-Algorithmuses $O(L + m)$ ist, mit L = benötigte Binärstellen zur Darstellung aller in der Listen vorhandenden Zahlen ($L = n \cdot log(n)$) und m die Anzahl der zum sortieren genutzten Buckets (=2). Damit ergibt sich eine Laufzeit von $O(L) = O(n \cdot log(n))$.

Aufgabe 2: Countingsort (2 + 2 Punkte) ★★★☆☆

Wie Sie aus der Vorlesung wissen, haben vergleichsbasierte Sortieralgorithmen eine worst-case

Laufzeit von mindestens $\mathcal{O}(n \log n)$. Es gibt allerdings auch Algorithmen, die ohne Vergleiche sortieren können, wie zum Beispiel Bucketsort. Ein anderes Beispiel ist *Countingsort*, das Sie im Folgenden betrachten sollen. Nehmen Sie dabei an, dass A ein Array der Länge n ist, das nur natürliche Zahlen zwischen 1 und k enthält.

Input: Ein Array A natürlicher Zahlen, dessen größte enthaltene Zahl k ist.
Output: Ein sortiertes Array R, das die Elemente aus A enthält.

```
1  Initialisiere ein Array C der Größe k mit 0 an jeder Stelle;
2  for i = 1, ..., n do
3  |  C[A[i]] ⟵ C[A[i]] + 1;
4  end
5  for i = 2, ..., k do
6  |  C[i] ⟵ C[i] + C[i − 1];
7  end
8  Initialisiere ein Array R der Größe n;
9  for i = n, ..., 1 do
10 |  R[C[A[i]]] ⟵ A[i];
11 |  C[A[i]] ⟵ C[A[i]] − 1;
12 end
13 return R;
```

a) Sei $A = [8, 3, 1, 3, 5, 4, 8, 7, 7, 2, 3, 1, 7, 5, 6, 2]$. Stellen Sie die Funktionsweise von Countingsort an diesem Beispiel dar. Geben Sie dazu

- das Array C nach Durchlauf der ersten for-Schleife
- das Array C nach Durchlauf der zweiten for-Schleife
- die Arrays C und R nach jeder Iteration der dritten for-Schleife

in Tabellenform an.

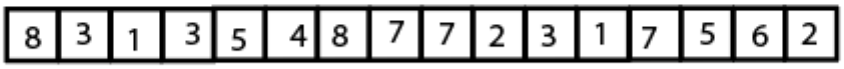
b) Argumentieren Sie, dass `Countingsort`(A) das Array A korrekt sortiert. Geben Sie die Laufzeit von Countingsort in $\mathcal{O}$-Notation in Abhängigkeit von n und k an.

2 Lösung Aufgabe 2

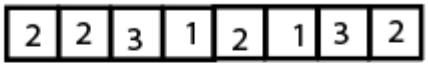
a)

Input :

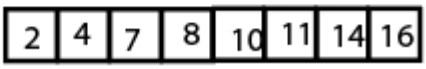
k = 8

A = 

C nach erstem Schleifendurchlauf :

C = 

C nach zweitem Schleifendurchlauf

C = 

Array-index

Anzahl Iterationen der for-Schleife

C	1	2	3	4	5	6	7	8
0	2	4	7	8	10	11	14	16
1	2	3	7	8	10	11	14	16
2	2	3	7	8	10	10	14	16
3	2	3	7	8	9	10	14	16
4	2	3	7	8	9	10	13	16
5	1	3	7	8	9	10	13	16
6	1	3	6	8	9	10	13	16
7	1	2	6	8	9	10	13	16
8	1	2	6	8	9	10	12	16
9	1	2	6	8	9	10	11	16
10	1	2	6	8	9	10	11	15
11	1	2	6	7	9	10	11	15
12	1	2	6	7	8	10	11	15
13	1	2	5	7	8	10	11	15
14	0	2	5	7	8	10	11	15
15	0	2	4	7	8	10	11	15
16	0	2	4	7	8	10	11	14

R	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0																
1				2												
2				2							6					
3				2						5	6					
4				2						5	6			7		
5		1		2						5	6			7		
6		1		2			3			5	6			7		
7		1	2	2			3			5	6			7		
8		1	2	2			3			5	6		7	7		
9		1	2	2			3			5	6	7	7	7		
10		1	2	2			3			5	6	7	7	7		8
11		1	2	2			3	4		5	6	7	7	7		8
12		1	2	2			3	4	5	5	6	7	7	7		8
13		1	2	2		3	3	4	5	5	6	7	7	7		8
14	1	1	2	2		3	3	4	5	5	6	7	7	7		8
15	1	1	2	2	3	3	3	4	5	5	6	7	7	7		8
16	1	1	2	2	3	3	3	4	5	5	6	7	7	7	8	8

b)

Laufzeit Initialisierung von C: $O(k)$
Laufzeit 1. for-Schleife: $O(n)$
Laufzeit 2. for-Schleife: $O(k)$
Laufzeit Initialisierung von R: $O(n)$
Laufzeit 3. for-Schleife: $O(n)$
Gesamtlaufzeit: $O(k) + O(n) + O(k) + O(n) + O(n) = O(2k) + O(3n) = O(k) + O(n)$

Korrektheit:

Nach dem ersten Schleifendurchlauf beinhaltet das Array C an der i-ten Stelle die Anzahl der Zahl i im zu sortierendem Array.
Nach dem zweiten Schleifendurchlauf beinhaltet das Array C die Anzahl aller Zahlen <= i des Arrays A.
Demnach bedeutet die i-te Stelle in C, dass die Zahl i sortiert an der Stelle C[i] erscheinen muss in A.

In der 3. for-Schleife wird nun in jeder Iteration zuerst das erste Element in A an korrekter Stelle eingefügt und anschließend wird der in C gespeicherte Counter um 1 verringert, womit die Behauptung im drüber genannten Satz erneut erfüllt wird.

Demanch ist die Korrektheit für den Algorithmus gegeben.

Aufgabe 3: Timsort (5 Punkte) ★★☆☆☆

In praktischen Anwendungen werden oft so genannte *hybride* Suchalgorithmen verwendet, die die Strategien mehrerer einfacher Suchalgorithmen kombinieren. *Timsort* ist ein solcher Algorithmus, der Insertion und Merge Sort kombiniert. Um genauer zu sein, werden zunächst kürzere Teilarrays (Länge höchstens k) jeweils mit Insertion Sort sortiert. Danach wird eine Lösung für das Gesamtarray A aus den Teillösungen mit Mergesort konstruiert. Der folgende Pseudocode beschreibt dieses Verfahren:

Input: Ein zu sortierendes Array A der Länge n, Länge der Teilarrays $k < n$
1 $\mathcal{B} \leftarrow \{A[i \cdot k \dots \min\{(i+1) \cdot k - 1, n-1\}] | 0 \le i < n/k\}$;
2 **for** $B \in \mathcal{B}$ **do** `InsertionSort(`B`)`;
3 **while** $|\mathcal{B}| > 1$ **do**
4 Seien $B_1, B_2 \in \mathcal{B}$ mit kleinsten Kardinalitäten in $\mathcal{B}$;
5 $B \leftarrow$ `Mische(`B_1`,`B_2`)`;
6 $\mathcal{B} \leftarrow (\mathcal{B} \cup \{B\}) \setminus \{B_1, B_2\}$;
7 **end**
8 Sei $B \in \mathcal{B}$;
9 $A \leftarrow B$;

Dabei ist `Mische(`B_1`,`B_2`)` die Subroutine von Merge Sort, welche eine sortierte Teilfolge aus den sortierten Teilfolgen B_1 und B_2 erstellt. $A[i \dots j]$ bezeichnet dabei das Teilarray von A vom Index i bis einschließlich Index j.
Wenn k nicht zu groß gewählt ist (z.B. $k = \mathcal{O}(\log n)$, in praktischen Anwendungen ist $k = 64$ ein gebräuchlicher Wert), wird eine Worst-Case Laufzeit von $\mathcal{O}(n \log n)$ erreicht.

Aufgabe: Visualisieren Sie anschaulich den Ablauf von Timsort auf einem selbstgewählten Beispiel mit $n \ge 8$ und $3 \le k \le \log n$. Kreative Lösungen sind gerne gesehen, zur Inspiration hier einige Darstellungen von Insertion Sort:

- `https://youtu.be/OGzPmgsI-pQ`
- `https://youtu.be/8oJS1BMKE64`
- `https://youtu.be/ROalU379l3U`
- `https://youtu.be/TZRWRjq2CAg`

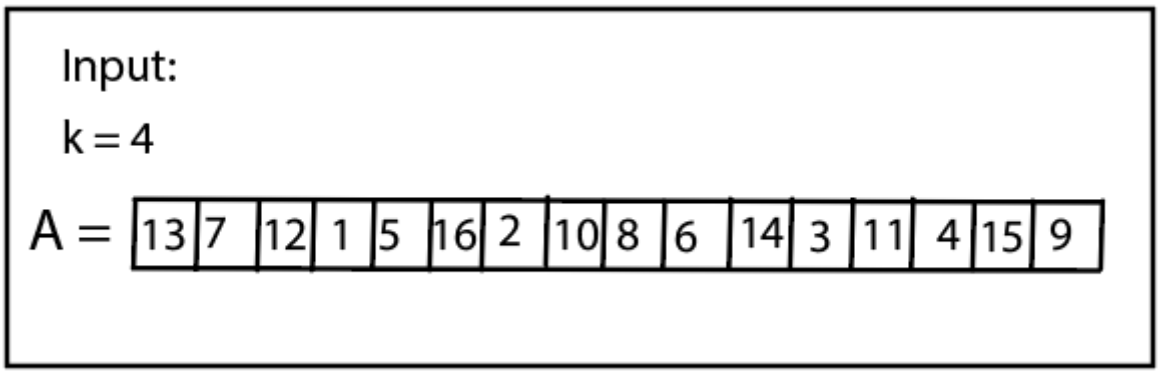
Sie dürfen diese Aufgabe in Gruppen von bis zu 8 Studierenden bearbeiten, in diesem Fall geben Sie bitte bei Ihrer Abgabe an, mit wem zusammen gearbeitet wurde. Vergessen Sie auch nicht eventuelle Inspirationsquellen zu zitieren. Die drei besten Abgaben können neben den Punkten auch Sachpreise gewinnen.

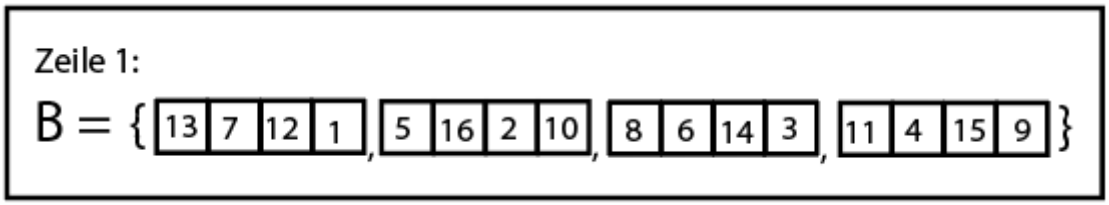
Noch ein paar Hintergrundinformationen: Timsort ist erfahrungsgemäß auf real vorkommenden Daten sehr performant. Dies liegt u.a. daran, dass Insertion Sort *fast* sortierte Daten sehr schnell sortieren kann und natürlich vorkommende Daten oftmals kurze aber fast sortierte Teilsequenzen (so genannte *runs*) enthalten. Beispielsweise ist es denkbar, dass das zu sortierende Array aus der Ausgabe verschiedener Algorithmen konstruiert wurde, die jeweils relevante IDs über eine iterative Suche innerhalb einer Datenbank gesucht haben. Aus diesem Grund wird Timsort standardmäßig

in vielen Plattformen wie Python, Java SE 7 oder Android verwendet. Zudem eignet sich Timsort auch sehr gut zum manuellen alphabetischen Sortieren von 400 Klausuren o.ä., da die anfängliche Insertion Sort-Phase die Zahl kleiner Stapel im Vergleich zu reinem Merge Sort drastisch reduziert.

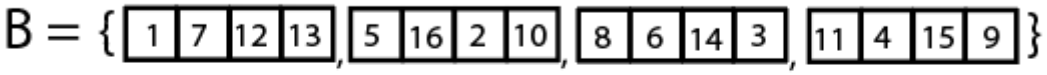
Wir wünschen ein frohes Weihnachtsfest und einen guten Rutsch ins neue Jahr 2022!

3 Lösung Aufgabe 3

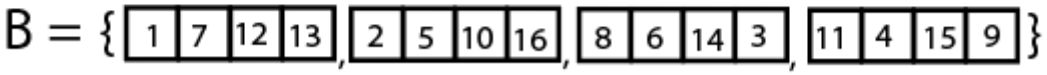




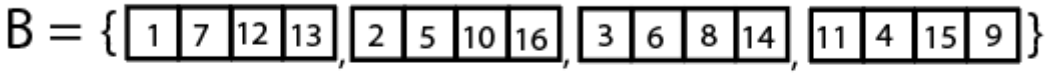
Zeile 2 (Insertionsort auf erste Teilfolge):



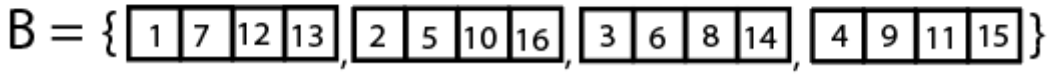
Zeile 2 (Insertionsort auf Teilfolge):



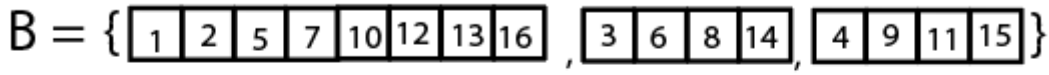
Zeile 2 (Insertionsort auf Teilfolge):



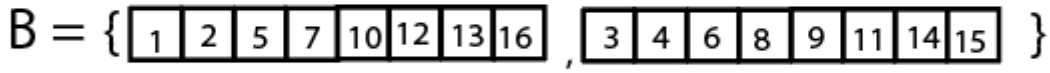
Zeile 2 (Insertionsort auf Teilfolge):



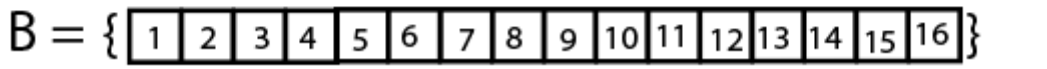
Zeile 3-7 (Mergesort auf Teilfolgen niedrigster Kardinalität):



Zeile 3-7 (Mergesort auf Teilfolgen niedrigster Kardinalität):



Zeile 3-7 (Mergesort auf Teilfolgen niedrigster Kardinalität):



Zeile 8 (setze **B** auf das einzige Element in B):

B =

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Zeile 9 (setze Input A auf **B**):

A =

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16