Übung 3: Algorithmen und Datenstrukturen

Theodor Bajusz	7159556	x	x	x	x	
Valerij Dobler	7068135	x	x	x	x	
Matz Radloff	6946325	x	x	x	x	
Robin Wannags	6948409	x	x	x	x	

14. Dezember 2020

Aufgabe 1

In Algorithmus 1 ist der Pseudocode des Algorithmus StoogeSort zu sehen.

Algorithmus 1: StoogeSort(A, i, j)

```
1  if A[i] > A[j]
2      A[i] ↔ A[j]
3  if i + 1 ≥ j
4      return
5  k ← ⌊(j − i + 1)/3⌋
6  StoogeSort(A, i, j − k) # sortiere ersten beiden Drittel
7  StoogeSort(A, i + k, j) # sortiere letzten beiden Drittel
8  StoogeSort(A, i, j − k) # sortiere ersten beiden Drittel
```

a)

Beweisen Sie die Korrektheit von *STOOGESORT*. Das heißt, beweisen Sie dass der Aufruf StoogeSort(A, 1, length(A)) das Array A korrekt sortiert. Führen Sie dazu einen Induktionsbeweis über die Länge von A. (5 Punkte)

Induktionsbehauptung: StoogeSort(A, 1, length(A)) sortiert das Array A korrekt.

Induktionsbeginn:

Zeile 3 und 4 stellen sicher, dass der Algorithmus bei Arrays mit der Länge 1 und 2 abbricht.

Arrayintervalle der Länge 1 sind trivialerweise sortiert. Für Intervalle der Länge 2 werden, falls sie falsch sortiert sind durch die Abfrage in Zeile 1 immer in die richtige Reihenfolge gebracht. //

Induktionsschritt:

Für Intervalle größer als 2 wird in k ein Index bestimmt, welcher annähernd ein Drittel des Intervalls markiert. Mit Hilfe von k wird StoogeSort auf den ersten beiden Dritteln aufgerufen. Ziel dabei ist, dass Elemente, welche im dritten Drittel sein müssen, ins zweite Drittel "geschoben"werden. Danach

geschieht ein Aufruf an StoogeSort mit den letzten beiden Dritteln. Dabei werden Elemente, welche im ersten Drittel sein müssen, ins zweite Drittel "geschoben". Das dritte Drittel hat dann nur Elemente, welche auch dort sein müssen. Beim letzten Aufruf an StoogeSort werden wieder die ersten beiden Drittel erneut sortiert. Dabei werden endgültig die Elemente, die im ersten Drittel sein müssen, auch ins erste Drittel "geschoben".

Daraus folgt, dass Arrays mit der Länge A immer korrekt sortiert sind.

b)

Analysieren Sie die worst-case Laufzeit von StoogeSort im O-Kalkül. Nutzen Sie dazu eine geeignete Rekursionsgleichung. (4 Punkte)

$n = length(A)$

$$T(n) := \begin{cases} O(1), & \text{für } n < 3 \\ 3 \cdot T(\lfloor n \cdot 2/3 \rfloor), & \text{sonst} \end{cases}$$

Rechnung:

$$\begin{aligned} &\geq 3 \cdot T(n \cdot 2/3) \\ &\geq 3 \cdot (3 \cdot T(n \cdot (2/3)^2)) \\ &\geq 3 \cdot (3 \cdot (3 \cdot T(n \cdot (2/3)^3))) \\ &\quad \vdots \end{aligned}$$

Wir berechnen wie oft Rekursionsaufrufe stattfinden, bis die Abbruchbedingung $n < 3$ erreicht ist:

$$3 < n \cdot (2/3)^k$$

$$3 \cdot (3/2)^k < n$$

$$log_{1,5}3 + k < log1,5n$$

$$k < log_{1,5}n - log_{1,5}3$$

$$k < log_{1,5}(\frac{n}{3})$$

$$k < \frac{log(\frac{n}{3})}{log(1,5)}$$

Wenn wir nun Ganzzahlige k suchen, welche diese Ungleichung genügen, müssen wir das Ergebnis aufrunden:

$$k < \lceil \frac{log(\frac{n}{3})}{log(1,5)} \rceil$$

.

$T(n)$ wird $n^{log_{1,5}(\frac{n}{3})}$-mal ausgeführt, bevor der Rekursionsabbruch stattfindet. Daraus folgt, dass $T(n) = O(n^{log_{1,5}(\frac{n}{3})})$

Aufgabe 2

a)

Sortieren Sie das Array $A = \langle 20, 13, 8, 5, 2, 12, 9 \rangle$ mithilfe des *QuickSort*-Algorithmus aus der Vorlesung. Stellen Sie den Inhalt des Arrays nach jedem Aufruf von *Partition* dar. (3 Punkte)

Durchlauf n	Array A
0	$\langle 20, 13, 8, 5, 2, 12, 9 \rangle$
1	$\langle 8, 5, 2, \mathbf{9}, 20, 12, 13 \rangle$
2	$\langle \mathbf{2}, 5, 8, \mathbf{9}, 20, 12, 13 \rangle$
3	$\langle \mathbf{2}, \mathbf{5}, \mathbf{8}, \mathbf{9}, 20, 12, 13 \rangle$
4	$\langle \mathbf{2}, \mathbf{5}, \mathbf{8}, \mathbf{9}, \mathbf{12}, \mathbf{13}, \mathbf{20} \rangle$

b)

Welchen Einfluss auf die Laufzeit hat allgemein die Auswahl des Pivotelements bei *QuickSort*? Skizzieren Sie worst-case- und best-case-Eingaben für eine konkrete Auswahl des Pivotelements (z.B. immer am linken oder rechten Rand). Begründen Sie Ihre Antwort. (2 Punkte)

Die Wahl des Pivotelements entscheidet beim nächsten Rekursionsaufruf über die Größe der beiden Intervalle. Falls das Pivotelement das kleinste oder größte Element im Intervall der Länge n des Arrays ist, dann wird ein Quicksort-Aufruf mit einem Intervall Länge n-1 aufgerufen und das andere mit der Länge 1.

Betrachten wir eine Folge $a = a_1, a_2, a_3, \ldots, a_n$, wobei $\exists i \exists j \in \mathrm{N}$ und $i \neq j$

c)

Ein Sortieralgorithmus ist stabil, wenn er die Reihenfolge der Arrayeinträge mit gleichem Wert bewahrt. Ist der *QuickSort*-Algorithmus aus der Vorlesung stabil? Begründen Sie Ihre Antwort. (2 Punkte)

Der Sortier-Algorithmus ist nicht stabil was mit folgendem Beispiel einfach zu widerlegen ist:

1 ist das Pivotelement
$\{2_1, 2_2, 1\} \xrightarrow{QuickSort} \{1, 2_2, 2_1\}$

Anfangs wird der das l auf den Index 2_1 gesetzt. Unser r ist dabei auf den Index 1 gesetzt. Das i wird auf den Index $l - 1$ gesetzt, also einen Platz links vom Index l. Außerdem wird nun j auch auf den Index 2_1 gesetzt. Das j wandert nun einen Index nach rechts und zeigt auf 2_2. Nun werden die Positionen von den Indizes 2_1 und 2_2 im Array vertauscht, da die Bedingung $if A[j] \leq x$ gilt. Nun rückt das i einen Index weiter nach rechts. Da das j im nächsten Schritt auf den Index 1 zeigt und die Bedingung $if A[j] \leq x$ wieder gilt, wird Index 1 die Stelle mit dem Index 2_1 tauschen. Nun erhalten wir die sortierte, aber nicht stabile Reihenfolge $1, 2_2, 2_1$. Würde die Bedingung einzeln die Relation kleiner und einzeln die Relation gleich prüfen, dann könnte man eine stabile Sortierung mit *QuickSort* erhalten.

Aufgabe 3

a)

Beschreiben Sie einen Algorithmus in Pseudocode, der zwei vollständige Max- Heaps gleicher Größe n vereinigt. Gehen Sie dazu davon aus, dass die Heaps keine gemeinsamen Elemente enthalten. (2 Punkte)

Algorithm 1: TwoMaxHeap(A1, A2)

```
1  A ← Array[2n]                                   /* Neues Array der Größe 2n */
2  k ← 1
3  for i ← 1; i ≤ length(A1);i ← i + 1 do
4      A[k] ← A1[i]
5      k ← k + 1
6  for i ← 1;i ≤ length(A2);i ← i + 1 do
7      A[k] ← A2[i]
8      k ← k + 1
     /* Sortiere das Array aufsteigend mithilfe von Countingsort in linearer Zeit */
9  countingsort(A)
                                                   /* Sortierungsreihenfolge umkehren */
10 for i ← 1;i ≤ ⌊length(A)/2⌋;i ← i + 1 do
11     A[i] ↔ A[length(A) − i]
12
```

Durch unseren Algorithmus entsteht ein sortiertes Arrays was widerrum ein ein Heap ist.

b)

Analysieren Sie die Laufzeit ihres Algorithmus. (2 Bonuspunkte, wenn sie beweisen können, dass ihr Algorithmus in $O(n)$ läuft.) (2 Punkte)

Zeile	Laufzeit
1	$O(1)$
2	$O(1)$
3	$O(n)$
4	$O(1)$
5	$O(2)$
6	$O(n)$
7	$O(1)$
8	$O(2)$
9	$O(2n)$
10	$O(n)$
11	$O(2)$

$$O(1) + O(1) + O(n) \cdot (O(1) + O(2)) + O(n) \cdot (O(1) + O(2)) + O(2n) + O(n)(2) = O(5n) = O(n)$$

Für die Bonuspunkte: Wir können hier *CountingSort* anwenden, weil wir davon ausgehen konnten, dass beide Heaps keine gemeinsamen Elemente beinhalten, und dass wir wissen, dass der Wertebe-

reich durch die größere Zahl der beiden Maxima ($max\{A1[1], A2[1]\}$ beschränkt ist. Wobei max die Mathematische Maximum-funktion ist:

$$max\{a, b\} = \begin{cases} a & a \geq b \\ b & \text{sonst} \end{cases}$$

Und $A1\,[1] := max(A1)$ und $A2\,[1] := max(A2)$ jeweils die größten Elemente beider Heaps $A1$ und $A2$ sind.