Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement Informatik
Markus Püschel
Peter Widmayer
Thomas Tschager
Tobias Pröger

3. November 2016

Datenstrukturen & Algorithmen Lösungen zu Blatt 6 HS 16

Lösung 6.1 *Vergleich von Sortieralgorithmen.*

	`bubbleSort`		`insertionSort`	
	min	max	min	max
Vergleiche	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$
Eingabefolge	jede	jede	$1, 2, \ldots, n$	$n, n-1, \ldots, 1$
Vertauschungen	0	$\Theta(n^2)$	0	$\Theta(n^2)$
Eingabefolge	$1, 2, \ldots, n$	$n, n-1, \ldots, 1$	$1, 2, \ldots, n$	$n, n-1, \ldots, 1$
Zusatzspeicher	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Eingabefolge	jede	jede	jede	jede

	`selectionSort`		`quicksort`	
	min	max	min	max
Vergleiche	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n^2)$
Eingabefolge	jede	jede	$(\star)$	$1, 2, \ldots, n$
Vertauschungen	0	$n-1$	$\Theta(n)$	$\Theta(n \log n)$
Eingabefolge	$1, 2, \ldots, n$	$n, 1, 2, \ldots, n-1$	$1, 2, \ldots, n$	$(\star)$
Zusatzspeicher	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$
Eingabefolge	jede	jede	$(\star)$	$1, 2, \ldots, n$

$(\star)$: Eine entsprechende Folge ist nicht leicht hinzuschreiben. Die Folge muss so konstruiert sein, dass jedes Pivotelement den zu sortierenden Bereich halbiert. Für $n = 7$ etwa ist eine solche Folge durch $4, 5, 7, 6, 2, 1, 3$ gegeben.

Lösung 6.2 *Verschiedene Fragen zu Sortieralgorithmen.*

a) Wegen der Bedingung $A[\lfloor i/2 \rfloor] > A[i]$ für alle $i \geq 2$ kann der kleinste Schlüssel nur an Positionen i mit $i > n/2$ stehen, also genau in der hinteren Hälfte des Arrays (genauer: auf den letzten $\lceil n/2 \rceil$ Positionen).

b) Sowohl Bubblesort als auch Sortieren durch Einfügen sind bereits in naiven Implementierungen stabil. Mergesort ist stabil, wenn man beim Verschmelzen darauf achtet, bei zwei gleichen Schlüsseln stets das weiter vorn stehende zu bevorzugen. Bei Sortieren durch Auswahl, Heapsort und Quicksort gibt es keinen einfachen Weg, Stabilität sicher zu stellen.

c) Bubblesort, Sortieren durch Auswahl, Sortieren durch Einfügen und Heapsort arbeiten direkt auf dem zu sortierenden Array und sind daher in-situ. Quicksort kann so angepasst werden, dass nur konstanter Extraplatz benötigt wird und ist daher auch in-situ. Mergesort muss wiederholt Teile des Arrays verschmelzen. Es gibt (komplizierte) Verfah-

ren, das Verschmelzen in-situ auszuführen, aber keine, die durch simple Modifikation des Standardalgorithmus erreichbar sind.

d) Zwar hat Quicksort im schlechtesten Fall eine Laufzeit von $\Theta(n^2)$, allerdings ist bei zufälliger Wahl des Pivotelements die Wahrscheinlichkeit für eine quadratische Laufzeit extrem klein. Im Erwartungswert beträgt die Laufzeit $\mathcal{O}(n \log n)$. Ausserdem wissen wir aus dem vorherigen Aufgabenteil, dass Quicksort im Gegensatz zu Mergesort in-situ arbeitet. Dies ist in der Praxis ein sehr grosser Vorteil von Quicksort. Weiterhin sind die Konstanten, die in der erwarteten Laufzeit von Quicksort "versteckt" sind, kleiner als die Konstanten in der Laufzeit von Mergesort. Trotzdem hat auch Mergesort seine Daseinsberechtigung, denn im Gegensatz zu Quicksort sortiert es stabil. Deswegen benutzt die `sort`-Methode der Java-Standardbibliothek eine Variante von Quicksort zum Sortieren von Arrays aus primitiven Typen (z.B. Integer, Float und Double), während zur Sortierung von Arrays aus allgemeinen Objekten eine Kombination aus Mergesort und Sortieren durch Einfügen benutzt wird.

Lösung 6.3 *3-Median-Quicksort.*

Die Grundidee bei der Konstruktion einer solchen Instanz besteht darin, dass in jedem Rekursionsaufruf das grösste und der zweitgrösste Schlüssel an die korrekte Position im Array verschoben werden, sodass der nächste rekursive Aufruf auf dem Rest der Schlüssel erfolgt. Sei $n \in \mathbb{N}$ beliebig. Wir konstruieren ein Array $A^{(n)}$ der Länge n, auf dem 3-Median-Quicksort eine quadratische Laufzeit besitzt.

Sei $A^{(n)} = (a_1, \ldots, a_n)$ so, dass $a_n < a_1 < a_2 < \cdots < a_{n-1}$ gilt. Ein mögliches Array, das diese Bedingung erfüllt, ist $(2, 3, 4, \ldots, n, 1)$. Das kleinste Element steht also an der letzten und das zweitkleinste Element an der vorletzten Stelle. Wir zeigen nun folgendes: Der Aufteilungsschritt von Quicksort zerlegt $A^{(n)}$ in ein Array der Länge 1 und ein Array $A' = (a'_1, \ldots, a'_{n-2})$ der Länge $n-2$, das genau die gleiche Struktur wie A hat, d.h., das $a'_{n-2} < a'_1 < a'_2 < \cdots < a'_{n-1}$ erfüllt. Insbesondere sind in A' die ersten $n-3$ Elemente aufsteigend sortiert, das kleinste Element befindet sich an der ersten und das zweitkleinste Element an der letzten Position.

Dazu betrachten wir einen Aufteilungsschritt auf $A^{(n)}$. Im ersten Schritt berechnen wir den Median aus dem ersten Element a_1, dem mittleren Element $a_{\lceil n/2 \rceil}$ und dem letzten Element a_n. Da a_n das kleinste und a_1 das zweitkleinste Element von $A^{(n)}$ sind, wird a_1 das Pivotelement, mit a_n vertauscht und der Aufteilungsschritt wie bisher durchgeführt. Der Durchlauf von links stoppt bei der Position $i = 2$ (die den Schlüssel a_2 enthält, der grösser als das Pivotelement a_1 ist). Der Durchlauf von rechts stoppt erst bei der Position $j = 1$ (die den Schlüssel a_n enthält, der kleiner als das Pivotelement a_1 ist). Da $j < i$ gilt, werden a_i und a_j nicht mehr vertauscht. Stattdessen wird das Pivotelement mit a_2 vertauscht und wir erhalten das umsortierte Array

$$(a_n, a_1, a_3, a_4, \ldots, a_{n-1}, a_2),$$

welches in das Array (a_n) der Länge 1 und das Array $A' = (a_3, a_4, \ldots, a_{n-1}, a_2)$ aufgeteilt wird. Wir beobachten jetzt, dass A' genau die gleiche Struktur wie $A^{(n)}$ hat: Da $a_2 < a_3 < \cdots < a_{n-1}$ gilt, sind die ersten $n-3$ Elemente strikt aufsteigend sortiert, das kleinste Element steht am Ende und das zweitkleinste am Anfang des Arrays.

Lösung 6.4 *Algorithmenentwurf und untere Schranke.*

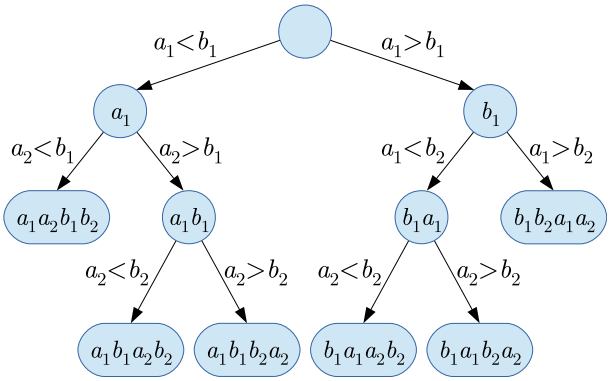
a) Die Verschmelz-Routine von Mergesort verwaltet zwei Zahlen i und j (initial, $i = j = 1$), die auf den jeweils nächsten Schlüssel aus A bzw. B zeigen. Solange sowohl $i \leq n$ als auch $j \leq n$ gelten, wird $A[i]$ mit $B[j]$ verglichen, und ist $A[i] \leq B[j]$, dann wird $A[i]$

als nächstes in die sortierte Gesamtsequenz aufgenommen (und i um 1 erhöht), und $B[j]$ ansonsten (und j um 1 erhöht). Mit jedem solchen Vergleich wird also entweder i oder j erhöht. Ist irgendwann entweder $i > n$ oder $j > n$, dann werden die bisher noch nicht betrachteten Schlüssel aus B bzw. aus A ohne weitere Vergleiche direkt in die sortierte Sequenz übernommen.

Sowohl i als auch j können jeweils maximal $n-1$ Mal erhöht werden, bis $i = n$ oder $j = n$ (oder beides) gilt. Ist also nach $2(n-1) = 2n-2$ vielen Vergleichen $i = j = n$, dann reicht ein weiterer Vergleich zwischen $A[n]$ und $B[n]$, um den vorletzten und den letzte Schlüssel der sortierten Gesamtsequenz festzulegen. Ist nach höchstens $2n-2$ vielen Vergleichen dagegen entweder $i > n$ oder $j > n$, dann terminiert das Verfahren sogar noch früher. Damit vergleicht die Verschmelz-Routine von Mergesort höchstens $2n-2+1 = 2n-1$ viele Schlüssel.

Die Arrays $A = (1, 3, 5, \ldots, 2n-1)$ und $B = (2, 4, 6, \ldots, 2n)$ zeigen, dass im schlechtesten Fall tatsächlich $2n-1$ viele Vergleiche ausgeführt werden. Die soeben hergeleitete obere Schranke von $2n-1$ vielen Vergleichen im schlechtesten Fall ist also exakt.

b) Seien $A = (a_1, a_2)$ und $B = (b_1, b_2)$ die gegebenen Arrays. In jedem inneren Knoten des Entscheidungsbaums werden zwei Schlüssel a_i und b_j miteinander verglichen. Der Einfachheit halber nehmen wir an, dass in der sortierten Gesamtfolge kein Schlüssel doppelt vorkommt, ein Vergleich ergibt daher immer $a_i < b_j$ oder $a_i > b_j$. Die Beschriftung jedes Knotens zeigt die bereits festgelegten Schlüssel in der sortierten Gesamtsequenz. Die Blätter des Entscheidungsbaums enthalten die sortierten Gesamtfolgen. Damit erhalten wir den folgenden Entscheidungsbaum:



c) Das sortierte Gesamtarray hat $2n$ Schlüssel, von denen genau n aus A und genau n aus B stammen. Jedes solche Gesamtarray können wir also durch eine Folge von Nullen und Einsen codieren, wobei eine Null bedeutet, den nächsten Schlüssel aus A zu nehmen, und eine Eins entsprechend bedeutet, den nächsten Schlüssel aus B zu nehmen. Die Folge hat $2n$ Positionen und exakt n Nullen, und es gibt exakt $\binom{2n}{n}$ viele Möglichkeiten, n Nullen auf $2n$ Positionen zu verteilen. Damit kann die Verschmelzung zweier Arrays zu $\binom{2n}{n}$ vielen verschiedenen Gesamtarrays führen.

Ein korrekter Algorithmus zum Verschmelzen zweier Arrays muss jede solche Folge korrekt zurückliefern. Repräsentieren wir einen solchen Algorithmus wie in b) durch einen Entscheidungsbaum, dann muss dieser für jede mögliche Ausgabe mindestens einen Knoten enthalten. Da es $\binom{2n}{n}$ viele mögliche Gesamtfolgen hat, muss ein Entscheidungsbaum eines korrekten Algorithmus also mindestens so viele Knoten haben.

Die Höhe dieses Baums (also die um Eins erhöhte Anzahl der im schlechtesten Fall vergli-

chenen Schlüssel) beträgt mindestens

$$\log \binom{2n}{n} = \log \left(\frac{(2n)!}{(n!)((2n-n)!)} \right) = \log \Big((2n)! \Big) - 2 \log(n!), \tag{1}$$

und mit der in Aufgabe 2.2 bewiesenen Abschätzung

$$n \ln n - n \leq \ln(n!) \leq n \ln n - n + \mathcal{O}(\ln n)$$

erhalten wir

$$\log \binom{2n}{n} \geq \frac{1}{\ln 2} \Big(2n \ln(2n) - 2n - 2(n \ln n - n + \mathcal{O}(\ln n)) \Big) \tag{2}$$

$$= \frac{1}{\ln 2} \Big(2n \ln 2 + 2n \ln n - 2n - 2n \ln n + 2n - \mathcal{O}(\ln n) \Big) = 2n - \mathcal{O}(\ln n). \tag{3}$$

d) Wir betrachten die Eingabe $A = (a_1, \ldots, a_n) = (1, 3, \ldots, 2n-1)$ und $B = (b_1, \ldots, b_n) = (2, 4, \ldots, 2n)$, und zeigen nun, dass *jedes* korrekte allgemeine deterministische Verfahren mindestens $2n - 1$ viele Schlüssel vergleichen muss.

Dazu überlegen wir zunächst, dass jedes korrekte Verfahren die Schlüssel a_1 und b_1 miteinander vergleichen muss. Würde dieser Vergleich ausgelassen, dann könnten wir in A und B die Schlüssel a_1 und b_1 miteinander vertauschen und erhielten die Eingabe $A'_1 = (b_1, a_2, a_3, \ldots, a_n)$ und $B'_1 = (a_1, b_2, b_3, \ldots, b_n)$. Man beachte, dass sowohl A'_1 als auch B'_1 noch immer aufsteigend sortiert sind. Da a_1 und b_1 nicht miteinander verglichen werden, kann der Algorithmus die Eingaben (A, B) und (A', B') nicht voneinander unterscheiden und würde für mindestens einer der beiden ein falsches Ergebnis ausgeben.

Genauso muss für jedes $k \in \{2, \ldots, n\}$ der Schlüssel a_k sowohl mit b_{k-1} als auch mit b_k verglichen werden. Angenommen, a_k würde nie mit b_k verglichen. Wie zuvor können wir a_k und b_k in A und B vertauschen und erhalten die (noch immer sortierten) Folgen $A'_k = (a_1, \ldots, a_{k-1}, b_k, a_{k+1}, \ldots, a_n)$ sowie $B'_k = (b_1, \ldots, b_{k-1}, a_k, b_{k+1}, \ldots, b_n)$, die der Algorithmus nicht von A und B unterscheiden kann. Angenommen, a_k würde nie mit b_{k-1} verglichen. Dann können erneut a_k und b_{k-1} in A und B vertauscht werden, und wieder könnte der Algorithmus die entstehenden (noch immer sortierten) Folgen nicht von A und B unterscheiden.