Karlsruher Institut für Technologie — Prof. Dr. Jörn Müller-Quade
Institut für Theoretische Informatik — Björn Kaidel, Sebastian Schlag, Sascha Witt

6. Übungsblatt zu Algorithmen I im SoSe 2017

http://crypto.iti.kit.edu/index.php?id=799
{bjoern.kaidel,sascha.witt}@kit.edu

Musterlösungen

Aufgabe 1 (*Quicksort,* 3 *Punkte*)

Sortieren Sie die Ziffern Ihrer Matrikelnummer mittels Quicksort. (Falls Sie zu zweit abgeben, genügt eine von beiden.) Wählen Sie als Pivot immer jeweils das letzte Element des Arrays. Verwenden Sie Quicksort mit dem Partitionierungs-Algorithmus *partition* aus der Vorlesung (Vorlesung 08, 24.05.2017, Folien 11). Verwenden Sie zur Darstellung des Ablaufs das Schema aus der Vorlesung (Vorlesung 08, 24.05.2017, Folie 12) und markieren Sie insbesondere immer den Wert von i und j.

Musterlösung:
Beispiel: Matrikelnummer 1542603 mit $\overline{i}$ und $\underline{j}$

1	5	4	2	6	0	3
$\overline{\underline{1}}$	5	4	2	6	0	3
1	$\overline{\underline{5}}$	4	2	6	0	3
1	$\overline{5}$	$\underline{4}$	2	6	0	3
1	$\overline{5}$	4	$\underline{2}$	6	0	3
1	2	$\overline{4}$	5	$\underline{6}$	0	3
1	2	$\overline{4}$	5	6	$\underline{0}$	3
1	2	0	$\overline{5}$	6	$\underline{4}$	3
1	2	0	$\overline{3}$	6	$\underline{4}$	5
1	2	0				
$\overline{\underline{1}}$	2	0				
$\overline{1}$	$\underline{2}$	0				
$\overline{0}$	$\underline{2}$	1				
	2	1				
	$\overline{\underline{2}}$	1				
	$\overline{1}$	2				
		2				
			6	4	5	
			$\overline{\underline{6}}$	4	5	
			$\overline{6}$	$\underline{4}$	5	
			4	$\overline{\underline{6}}$	5	
			4	$\overline{5}$	6	
			4			
					6	

Aufgabe 2 (k-*Wege Merging,* $3+3=6$ *Punkte*)

Gegeben seien k doppelt-verkettete aufsteigend sortierte Listen $L_1, ..., L_k$. Jede Liste L_i habe $\frac{n}{k}$ viele Elemente (die natürliche Zahlen enthalten), wobei $n, k \in \mathbb{N}$ und n durch k teilbar sei. Sie müssen zur Lösung der Aufgabe keinen Pseudocode angeben, es genügt, wenn Sie Ihre Algorithmen eindeutig beschreiben. Die Algorithmen müssen die Strukturen der Listen L_i nicht erhalten.

a) Geben Sie zwei Algorithmen an, die jeweils in Zeit $\mathcal{O}(n \cdot k)$ laufen und das Folgende tun: Der erste Algorithmus soll eine *aufsteigend* sortierte Liste erzeugen, die genau die Elemente der k sortierten Listen $L_1, ..., L_k$ enthält. Der zweite Algorithmus soll eine *absteigend* sortierte Liste erzeugen, die genau die Elemente der k sortierten Listen $L_1, ..., L_k$ enthält. Begründen Sie die Laufzeit ihrer Algorithmen.

b) Angenommen, die Listen sind nur einfach verkettet, es existieren keine Zeiger auf die letzten Elemente der Listen (d.h. man kann nur in $\mathcal{O}(n)$ auf die letzten Elemente zugreifen, aber nicht in $\mathcal{O}(1)$) und es ist Ihnen nicht erlaubt, andere Datenstrukturen außer einfach verketteten Listen zu verwenden. Geben Sie auch für diesen Fall zwei Algorithmen an, die aus den k Teillisten eine aufsteigend und eine absteigend sortierte Liste in Laufzeit $\mathcal{O}(n \cdot k)$ erzeugen. Begründen Sie die Laufzeit Ihrer Algorithmen.

Musterlösung:

a) Folgender Algorithmus erzeugt eine aufsteigend sortierte Liste:
Suche in jedem Schritt das kleinste Element der k Listen. Entferne es aus der entsprechenden Liste und hänge es an das Ende der neuen Liste an.
Das Finden des kleinsten Elements aus den k Listen kann in $\mathcal{O}(k)$ erfolgen, indem man die Listen Stück für Stück durchgeht. Die jeweils kleinsten Elemente der Listen $L_1, ..., L_k$ stehen immer am Listenanfang (da sie aufsteigend sortiert sind), so dass das kleinste Element der k Listen mit höchstens k Vergleichen gefunden werden kann. Das Entfernen der Elemente aus den Listen ist dabei essenziell, da wir ansonsten immer wieder die gleichen Elemente vergleichen würden. Die Suche kann man beispielsweise so implementieren, dass man einen Zeiger auf das kleinste Element von Liste L_1 speichert. Danach vergleicht man dieses Element nach und nach mit den kleinsten Elementen der anderen Listen. Wird ein kleineres Element gefunden, so aktualisiert man den Zeiger.

Insgesamt gibt es n Elemente, so dass man das Finden des kleinsten Elements der k Listen höchstens n mal durchführen muss. Ausschneiden eines Elements und Einfügen eines Elements aus einer doppelt verketteten Liste geht in konstanter Zeit. Damit ergibt sich die Gesamtlaufzeit von $\mathcal{O}(n \cdot k)$.

Der Algorithmus zum Erzeugen einer absteigend sortierten Liste funktioniert analog, nur suchen wir nun jeweils das maximale Element. Dieses steht, da die Listen L_i aufsteigend sortiert sind, immer am Ende der Liste. Da eine doppelt verkette Liste auch einen Zeiger auf das letzte Element bereit stellt, können wir auch auf dieses in konstanter Zeit zugreifen. Die Laufzeitargumentation ist analog zum obigen Algorithmus.

b) Um eine absteigend sortierte Liste zu erzeugen, verwenden wir eine Variante zu der Lösung aus Aufgabe a): Wir entfernen jeweils das kleinste Element der k Listen und fügen es dann am *Anfang* der neuen Liste ein. Dadurch ist die Liste am Ende absteigend sortiert.

Um eine aufsteigend sortierte Liste zu erhalten, erstellen wir zuerst wie eben eine absteigend sortierte Liste und invertieren diese einfach (d.h. wir drehen die Reihenfolge der Elemente um). Dies geht in Laufzeit $\Theta(|L|) = \Theta(n)$ (vergleiche mit Aufgabe 4 des dritten Übungsblatts). Insgesamt ergibt sich also auch hier eine Laufzeit in $\mathcal{O}(n \cdot k)$.

Aufgabe 3 (*Algorithmenentwurf,* 4 *Punkte*)

Gegeben sei ein Array von insgesamt n natürlichen Zahlen im Intervall von 1 bis k. Geben Sie einen Algorithmus an, der für beliebige natürliche Zahlen a, b mit $1 \leq a < b \leq k$ in konstanter Zeit (also in $\mathcal{O}(1)$) berechnen kann, wieviele der n Zahlen im Intervall $[a, b]$ liegen. Der Algorithmus darf $\mathcal{O}(n + k)$ Vorberechnungszeit benötigen. D.h. ihr Algorithmus darf vor der ersten a, b-Anfrage Vorberechnungen durchführen, die in Laufzeit $\mathcal{O}(n + k)$ möglich sind. Ihr Algorithmus darf bei den Anfragen nun auf

die Ergebnisse der Vorberechnung zugreifen, um die Laufzeit in $\mathcal{O}(1)$ zu erreichen. Begründen Sie die Laufzeit Ihrer Vorberechnungsphase und Ihres Algorithmus.

Sie müssen zur Lösung der Aufgabe keinen Pseudocode angeben, es genügt, wenn Sie Ihre Algorithmen eindeutig beschreiben. Für schlechtere Laufzeiten kann es noch Teilpunkte geben.
Hinweis: Zählen Sie in der Vorberechnungsphase, wie häufig jede Zahl von 1 bis k im Array vorkommt und nutzen Sie diese Information geschickt.

Musterlösung:
Es werden zwei zusätzliche Arrays $B[1 \dots k]$ und $C[1 \dots k]$ benötigt. Der Algorithmus initialisiert zunächst alle Elemente in B zu 0, dieser Schritt benötigt Zeit $\mathcal{O}(k)$. Dann erhöhen wir für jede der gegebenen Zahlen i den Eintrag an der i-ten Stelle $B[i]$ um 1. Dieser Schritt benötigt Zeit $\mathcal{O}(n)$. Der Eintrag $B[i]$ gibt nun die Häufigkeit der Zahl i unter den gegebenen n Zahlen an. Nun setzen wir $C[1] = B[1]$ und berechnen für alle $l = 2, 3, \dots, l$ den Eintrag $C[l] = B[l] + C[l-1]$. Dieser Schritt benötigt Zeit $\mathcal{O}(k)$. Der Eintrag $C[i]$ gibt nun die Anzahl der Zahlen an, die kleiner oder gleich i sind. Wir können nun jede Anfrage der Form a, b in konstanter Zeit mit $C[b] - C[a] + B[a]$ beantworten (alle Zahlen, die kleiner oder gleich b sind - alle Zahlen, die kleiner oder gleich a sind + alle Zahlen, die gleich a sind). Dies ist in $\mathcal{O}(1)$, da es sich nur um konstant viele Arrayzugriffe handelt.

Aufgabe 4 (*Sortiertes Einfügen,* $2 + 3 = 5$ *Punkte*)

Gegeben sei eine aufsteigend sortierte doppelt verkette Liste mit n Elementen (die jeweils natürliche Zahlen enthalten). Nun haben Sie $k \leq n$ weitere natürliche Zahlen in einem unsortierten Array, die in die sortierte Liste eingefügt werden sollen, sodass die Liste nach dem Einfügen weiterhin sortiert ist. Für diese Aufgabe ist kein Pseudocode notwendig, es genügt, wenn Sie Ihre Algorithmen eindeutig beschreiben.

a) Geben Sie einen Algorithmus an, der Laufzeit $\mathcal{O}(k \cdot n)$ hat und das Problem löst, ohne den Inhalt des Arrays und insbesondere die Reihenfolge der Elemente im Array zu ändern. Begründen Sie die Laufzeit des Algorithmus. Sie dürfen keine weitere Datenstruktur (wie Arrays, Heaps, weitere Listen...) anlegen, d.h. Sie dürfen nur das gegebene Array und die doppelt verkettete Liste verwenden.

b) Geben Sie nun einen Algorithmus an, der Laufzeit $\mathcal{O}(k \cdot \log(k) + n)$ hat und das Problem löst. Begründen Sie die Laufzeit des Algorithmus.

Musterlösung:

a) Man fügt die Elemente nacheinander folgendermaßen in die sortierte Liste ein: Um die richtige Position zu finden, führt man jeweils eine lineare Suche durch. D.h. man geht die Liste Stück für Stück durch und vergleicht den Wert des aktuellen Elements mit dem Wert des einzufügenden Elements, bis man die richtige Stelle gefunden hat. Die lineare Suche hat pro einzufügendem Element Aufwand $\mathcal{O}(n + k)$, da im schlimmsten Fall die ganze Liste mit schon hinzugefügten Array-Elementen durchlaufen werden muss. Da $k \leq n$ ist, liegt dies auch in $\mathcal{O}(n)$. Es werden insgesamt k Elemente einfügt, somit ergibt sich ein Gesamtaufwand von $\mathcal{O}(k \cdot n)$.

b) Hier geht man etwas anders vor: Zuerst sortieren wir das Array mit den neuen Elementen in Zeit $\mathcal{O}(k \log k)$, beispielsweise mit dem Algorithmus Mergesort aus der Vorlesung. Das sortierte Array wandeln wir dann in Zeit $\mathcal{O}(k)$ in eine doppelt verkette Liste um. Danach haben wir zwei sortierte Listen: in der einen sortierten Liste befinden sich die k neuen Elemente und in der anderen befinden sich die n alten sortierten Elemente. Nun verwenden wir die Funktion `merge` (aus Mergesort, angepasst auf Listen) und erhalten so in Zeit $\mathcal{O}(n)$ (da $k < n$) eine sortierte Liste, die die Elemente beider Listen aufsteigend sortiert enthält. Insgesamt ergibt sich also ein Aufwand von $\mathcal{O}(k \log(k) + k + n) = \mathcal{O}(k \cdot \log(k) + n)$.

Hinweis: Als Optimierung kann man sich die Umwandlung des Arrays in eine Liste auch sparen, dann muss man aber `merge` so anpassen, dass es gleichzeitig mit Arrays und Listen umgehen kann.