RWTH Aachen **SS 2016**
Lehrgebiet Theoretische Informatik Lösungsvorschlag 6
Rossmanith–Hensel–Kuinke–Sánchez 30. Mai 2016

Übung zur Vorlesung Datenstrukturen und Algorithmen

Aufgabe T17

a) Gesucht ist die Anzahl der Index-Paare (i, j) mit $i < j$, so daß der Wert an Stelle i größer ist als der Wert an Stelle j. Für die gegebene Zahlenfolge sind das genau die Paare

$$\begin{aligned} &\{(1,2),(1,3),(1,4),(1,5),(1,6),(1,8),(2,3),(2,5),(2,6),(2,8)\} \\ \cup\ &\{(3,6),(3,8),(4,5),(4,6),(4,8),(5,6),(5,8),(6,8),(7,8)\}, \end{aligned}$$

also hat sie 19 Inversionen.

b) Die Folge hat 6 Läufe:

- 8
- 6
- $3, 7$
- 4
- $2, 20$
- -45

c)
- 8 wird als Pivotelement gewählt
- Vertauschen von 20 und -45: $8, 6, 3, 7, 4, 2, -45, 20$
- Vertauschen von -45 und 20: $8, 6, 3, 7, 4, 2, 20, -45$
- Abbruch der Schleife: $l > r$ (l zeigt auf -45, r auf 20)
- Wert an Stelle l rückt nach vorne, Wert an Stelle r rückt an Stelle l und das Pivotelement rückt an Stelle r: $-45, 6, 3, 7, 4, 2, 8, 20$

d) In der letzten Mischphase sind die linke und die rechte Teilfolge bereits sortiert: $3, 6, 7, 8, -45, 2, 4, 20$. Gemischt werden sollen $3, 6, 7, 8$ und $-45, 2, 4, 20$.

- $3 > -45 \Rightarrow b[0] = -45$
- $3 > 2 \Rightarrow b[1] = 2$
- $3 \leq 4 \Rightarrow b[2] = 3$
- $6 > 4 \Rightarrow b[3] = 4$
- $6 \leq 20 \Rightarrow b[4] = 6$
- $7 \leq 20 \Rightarrow b[5] = 7$
- $8 \leq 20 \Rightarrow b[6] = 8$
- linker Counter hat linke Teilfolge verlassen $\Rightarrow b[7] = 20$

Aufgabe T18

Das oberste Element des Stacks wird sofort am Anfang der Schleife entfernt, interessant ist also das erste der beiden eingefügten Intervalle. Bringen wir den Algorithmus dazu, als erstes Element ein sehr kleines Intervall (etwa der Größe eins) auf den Stack zu legen und dann gezwungenermaßen ein sehr großes Intervall als zweites, so benötigt Quicksort $\Omega(n)$ Iterationen der äußeren Schleife—eine Beispielinstanz dafür wäre schlicht ein bereits aufsteigend sortiertes Array. Da nun in jeder Iteration ein Element auf dem Stack verbleibt, erreicht der Stack eine Größe von $\Omega(n)$.

Dieses Verhalten kann verhindert werden, indem man immer das kürzere Intervall oben auf den Stack legt: so kann der Stack maximal eine Größe von $O(\log n)$ erreichen.

Nehmen wir o.b.d.A. an, die Eingabe bestehe aus einer beliebigen Permutation der Elemente $1 \dots n$ und alle diese Permutationen haben die gleiche Wahrscheinlichkeit. Ein einfaches Zählargument zeigt, daß die anfängliche Vertauschung Permutationen auf den Elementen $2 \dots n$ erzeugt, die wieder alle gleich wahrscheinlich sind: für jede der $(n-1)!$ vielen Permutationen können n verschiedene Permutation der Elemente $1 \dots n$ erzeugt werden, indem ein Element durch die 1 ersetzt und anschließend den anderen vorangestellt wird. Da dies niemals zwei gleiche Permutationen erzeugt, erhalten wir $n(n-1)! = n!$ viele Permutationen auf n Elementen—umgekehrt ist damit die Wahrscheinlichkeit, eine konkrete Permutation der Elemente $2 \dots n$ zu erhalten, $n\frac{1}{n!} = \frac{1}{(n-1)!}$

Aufgabe T19

Die **while**-Schleife wird genau dann betreten, falls $a[i]$ nicht das größte Element in dem Unterarray $a[0], ..., a[i]$ ist.

Die Wahrscheinlichkeit, daß sich an der Stelle i die größte Zahl befindet ist $1/(i+1)$ für ein zufällig initialisiertes Array. Demnach ist die Wahrscheinlichkeit, daß dies nicht der Fall ist $1 - 1/(i+1)$. Somit ergibt sich für den Erwartungswert:

$$E = \sum_{i=1}^{n-1} \left(1 - \frac{1}{(i+1)}\right) = n - 1 - H_n + 1 = n - \ln(n) + O(1)$$

Wobei H_n die n-te harmonische Zahl ist.

Aufgabe H16

a) Die äußere **for**-Schleife durchläuft das gesamte Array von links nach rechts. In der Variable m wird das letzte Element gespeichert. Die zweite **for**-Schleife durchläuft das Teilarray, welches beim Element $i+1$ (Index der äußeren Schleife) beginnt. Diese innere Schleife sucht das kleinste Element in dem Unterarray $a[i+1], ..., a[n-1]$ und speichert sich den Index in m. Am Ende innerhalb der äußeren **for**-Schleife wird das i-te mit dem m-ten Element vertauscht. Das heißt, die Elemente in dem Intervall $[0, i]$ sind zu jeder Zeit sortiert.

b) Es wird nicht überprüft, ob $a[m]$ überhaupt kleiner als $a[i]$ ist. In dem Fall, daß sich an dem 0. Index das kleinste Element befindet, hat das z.B. die Auswirkung, daß das kleinste Element nach rechts durchgereicht wird. Sei die Eingabe $[1, 2, 3, 4]$, dann sieht das Array am Ende der äußeren **for**-Schleife nach jedem Durchlauf wie folgt aus:

0. $[2, 1, 3, 4]$ 1. $[2, 3, 1, 4]$ 2. $[2, 3, 4, 1]$

c) Anstatt m initial auf $n - 1$ zu setzen, könnte man es auf i setzen, so daß in dem oben beschriebenen Fall i mit sich selbst getauscht würde.

Aufgabe H17

a) Wir nutzen eine Inorder-Traversierung des Baumes:

Falls der aktuelle Knoten ein linkes Kind hat, steigen wir zu diesem hinab und gehen für diesen nach gleichem Prinzip vor. Anschließend geben wir den Schlüssel des aktuellen Knotens aus. Schließlich überprüfen wir, ob es ein rechtes Kind gibt, und falls ja, gehen wir auch für dieses nach gleichem Prinzip vor.

Dabei wird jeder Knoten genau einmal besucht, wobei ein Besuch konstante Laufzeit hat. Die Laufzeit des Algorithmus is somit $O(n)$.

b) Man kann einen binären Suchbaum zum Sortieren verwenden, indem man alle Schlüssel der Reihe nach einfügt und anschließend mithilfe der Inorder-Traversierung ausgibt. Damit die Einfügeoperationen möglichst effizient durchgeführt werden können und der Baum nicht degeneriert, können wir dafür zum Beispiel AVL-Bäume verwenden.

c)
- Gewöhnliche binäre Suchbäume: Im schlimmsten Fall haben wir eine bereits sortierte (oder umgekehrt sortierte) Folge von Zahlen, sodass die Höhe des Baumes linear in der Anzahl der Schlüssel ist. Damit braucht das Einfügen lineare Zeit und die gesamte Laufzeit ist $O(n^2)$.
- AVL-Bäume: Hier ist die Laufzeit für Einfügeoperationen $O(\log(n))$, da sowohl die Höhe des Baumes als auch das Rebalanzieren im worst-case logarithmisch in der Anzahl der Schlüssel sind. Deshalb ergibt sich beim Einfügen von n Schlüsseln eine Laufzeit von $O(n \log(n))$.
- Splay-Bäume: Amortisiert ist auch hier die Laufzeit für Einfügeoperationen $O(\log(n))$. Deshalb ergibt sich insgesamt wieder eine Laufzeit von $O(n \log(n))$.

d) Falls die Folge (umgekehrt) sortiert ist, ergeben sich folgende Fälle:

- Gewöhnliche binäre Suchbäume: Der nächste Schlüssel wird im sortierten Fall immer rechts unten (bzw. im umgekehrt sortierten Fall immer links unten) hinzugefügt, sodass sich ein degenerierter Baum mit linearer Höhe bildet. Damit braucht eine Einfügeoperationen lineare Zeit und $\Omega(n)$ viele Operationen brauchen $\Omega(n)$ Zeit. Damit ist die gesamte Laufzeit in $\Theta(n^2)$.
- AVL-Bäume: Da es sich hier um balancierte Bäume handelt, ist die Länge eines Pfades zwischen Wurzel und einem beliebigen Blatt immer $\Omega(\log(n))$. Da man beim Einfügen eines neuen Schlüssels bis zu einem Blatt hinabsteigt, ergibt sich also in jedem Fall eine Laufzeit von $\Theta(n \log(n))$.
- Splay-Bäume: Der nächste Schlüssel wird im sortierten Fall immer rechts von der Wurzel eingefügt und anschließend mit einem einfachen *zag* hochrotiert. Das Einfügen eines Elementes geschieht also in konstanter Zeit. Im umgekehrt sortierten Fall werden die Schlüssel analog links von der Wurzel eingefügt und mit einem *zig* rotiert. Insgesamt ergibt sich also eine Laufzeit von nur $\Theta(n)$.