Datenstrukturen und effiziente Algorithmen
Blatt 2

Markus Vieth, David Klopp, Christian Stricker

5. November 2015

Nr.1

a)

Wir gehen von einem sortierten Array aus. Dieser wird in zwei gleich große Hälften zerteilt. Ist der gesuchte Werte kleiner als der mittlere Wert, wird im linken Teilarray weiter gesucht, andernfalls im rechten. Wenn das gesuchte Element nicht gefunden wird, gibt der Algorithmus -1 zurück.

b)

$$T(n) = 1 \cdot T\left(\frac{n}{2}\right) + 2 \cdot \left(n^0\right) \quad \Rightarrow$$

a = 1 // in genau einem Array wird pro Schritt gesucht
b = 2 // zwei Teilarrays pro Schritt
$\alpha = 0$

Fall 3 des Master-Theoreme: $a = b^\alpha$

$\Theta(n^0 \cdot log(n)) = \Theta(log(n))$

Nr.2

Wir verwenden im folgenden Mergesort zum sortieren des Array. Nach dem Hinweis gilt für die Konstanten: c = 1:

Mergesort:

$$T_M(n) = n^{\log_2(2)} + n\log_2(n) = n + n\log_2(n)$$

Binäresuche:

$$T_B(n) = n^{\log_2(1)} + n^0 \log_2 n = 1 + \log_2 n$$

$\Rightarrow$ sortierte Suche:

$$T_1(n) = T_M(n) + T_B(n) \cdot k \quad //k\ Iterationen\ der\ Schleife$$

lineare Suche:

$$T_2(n) = n \cdot k$$

Wir suchen also jenes k, so dass gilt:

$$T_1(n) - T_2(n) = 0$$
$$\Leftrightarrow n(1 + \log_2(n)) + k_{grenz}(1 + \log_2(n)) - n \cdot k_{grenz} = 0$$
$$\Leftrightarrow k_{grenz} = \frac{-n(1 + \log_2(n))}{1 + \log_2(n) - n} \quad //bei\ n > 2$$

Für alle $k > k_{grenz}$ macht es Sinn den Array zuerst zu sortieren und anschließend zu suchen. Für $n < 2$ oder $k < k_{grenz}$ ist die lineare Suche effizienter.

Nr.3

a)

$Beh: 2^{n+k} \in O(n^{2)}$
$Bew:$

$$\lim_{n\to\infty}\frac{2^{n+k}}{2^n} = \lim_{n\to\infty}\frac{2^n \cdot 2^k}{2^n} = \lim_{n\to\infty}2^k = 2^k < \infty$$

$q.e.d$

b)

$Beh: 2^{2n} \notin O(n^{2)}$
$Bew:$

$$\lim_{n\to\infty}\frac{2^{2n}}{2^n} = \lim_{n\to\infty}2^{2n} \cdot 2^{-n} = \lim_{n\to\infty}2^{2n-n} = \lim_{n\to\infty}2^n = \infty \not< \infty$$

$q.e.d$

c)

$Beh: \log(n!) \in O(n \cdot \log(n))$
$Bew:$

$$Beachte: \forall m < n \quad m, n \in \mathbb{N} | \log(n) > \log(m)$$

$$Beachte: \log(n!) = \sum_{k=1}^{n} log(k) \leq \sum_{k=1}^{n} log(n) = n \cdot \log(n)$$

$$\Rightarrow \lim_{n\to\infty}\frac{\log(n!)}{n \cdot \log(n)} \leq \lim_{n\to\infty}\frac{n\log(n)}{n\log(n)} = 1 < \infty$$

$q.e.d$

d)

$Beh: 2^{kn} \notin O(2^n)$
$Bew:$

$$\lim_{n\to\infty}\frac{2^{kn}}{2^n} = \lim_{n\to\infty}2^{(k-1)\cdot n} = \infty \not< \infty$$

$q.e.d$

e)

$Beh: 2^{\frac{n}{2}} \in O(\frac{2^n}{log(n)})$
$Bew:$

$$\lim_{n\to\infty}\frac{2^{\frac{n}{2}}}{\frac{2^n}{log(n)}} = \lim_{n\to\infty}\frac{2^{\frac{n}{2}}\log(n)}{2^n} = \lim_{n\to\infty}\frac{\log(n)}{2^{\frac{n}{2}}} = 0 < \infty$$

$q.e.d$

Nr. 4

a)

Zeile 1-5: Kopiere die erste sortierte Hälfte des Arrays A in ein Hilfsarray B.
Zeile 6-13: Vergleiche das Hilfsarray mit der zweiten Hälfte des Arrays A und sortiere die Zahlen in der richtigen Reihenfolge in das Array A.
Zeile 14-16: Falls nach dem Vergleichen Elemente in B verblieben sind, kopiere diese in der gleichen Reihenfolge ans Ende von A.
Alle Elemente im Array A, welche zwischen k und j stehen, können überschrieben werden, weil diese Elemente entweder im Array B stehen oder bereits in die erste Hälfte des Arrays A kopiert wurden. Alle Elemente zwischen left und k sind bereits sortiert. (Für Sortierung: siehe b))

b)

Wie in a) geklärt, gehen bei dem Algorithmus keine Daten verloren. Nach jedem Durchlauf der zweiten Schleife sind die Elemente zwischen left und k sortiert, da nach dem sort Schritt die linke Hälfte und die rechte Hälfte vorsoriert sind. Solleten alle Elemente der linken Hälfte kleiner als das größte Element der rechten Hälfte sein, wird nur die zweite Schleife durchlaufen, sonst werden die restlichen Elemente im Array A mit den noch nicht kopierten Elementen des Array B überschrieben.

c)

Speicherbedarf:

Im Gegensatz zur merge Funktion aus der Vorlesung, welche ein Hilfsarray der Größe $|A|$ benötigt, genügt hier ein Hilfsarray B der Größe $\frac{|A|}{2}$, da das Array A nur zur Hälfte und nicht vollständig kopiert werden muss.

Kopieroperationen:

Vorlesung Das komplette Array A der Größe $|A|$ wird sortiert nach B kopiert, anschließend wird das Array B nach A kopiert $\Rightarrow 2 \cdot |A|$ Kopiervorgänge.

Best Case Alle Elemente der linken Hälfte sind kleiner als das kleinste Element der rechten Hälfte $\Rightarrow \frac{|A|}{2}$ Kopiervorgänge und das Kopieren von A in B sind $\frac{|A|}{2}$ Kopiervorgänge $\Rightarrow |A|$ Kopiervorgänge.

Worst Case Das kleinste Element der linken Hälfte ist größer, als das größte Element der rechten Hälfte $\Rightarrow |A|$ Kopiervorgänge und das Kopieren von A in B sind $\frac{|A|}{2}$ Kopiervorgänge $\Rightarrow 1,5 \cdot |A|$ Kopiervorgänge.

Elementvergleiche:

Best Case Alle Elemente der linken Hälfte sind kleiner als das kleinste Element der rechten Hälfte oder das kleinste Element der linken Hälfte ist größer, als das größte Element der rechten Hälfte$\Rightarrow \frac{|A|}{2}$ Elementarvergleiche.

Worst Case Das kleinste Element der rechten Hälfte ist kleiner, als das kleinste Element der rechten Hälfte und das größte Element der linken Hälfte ist kleiner als das größte Element der rechten Hälfte. $\Rightarrow |A|$ Elementarvergleiche.
Beide merge Funktionen sind bei den Elementarvergleichen identisch.