Algorithmen und Datenstrukturen

Übungsgruppe 14

Utz Pöhlmann
4poehlma@informatik.uni-hamburg.de
6663579

Louis Kobras
4kobras@informatik.uni-hamburg.de
6658699

Paul Testa
paul.testa@gmx.de
6251548

9. November 2015

Punkte für den Hausaufgabenteil:

3.1	3.2	3.3	3.4	Σ

1 Zettel vom 29. 10. – Abgabe: 09. 11.

1.1 Übungsaufgabe 3.1

[| 3]

Gegeben seien die folgenden Code-Fragmente. Geben Sie eine möglichst dichte asymptotische obere Schranke für die Laufzeit der einzelnen Code-Fragmente jeweils in Abhängigkeit von n an. Begründen Sie Ihre Behauptung. (Es geht hier nicht um die Bedeutung des Codes, nur um die Laufzeit. An der Stelle von $sum = sum + j$ könnte sinnvoller(er) Code stehen. Die Einrückung gibt den Skopus der Schleifenkonstrukte an.)

```
ALGO1()
1  for i = 0 to n
2      for j = n downto 1
3          sum = sum + j
4      for j = 1 to n
5          sum = sum + j
```

```
ALGO2()
1  i = 1
2  while i < 2 · n
3      for j = 1 to i
4          sum = sum + j
5      i = i + 2
```

```
ALGO3()
1  i = 1
2  while i · i < n
3      i = i + 1
4      j = n
5      while j > 1
6          sum = sum + j
7          j = j/2
```

1.1.1 Aufgabe 3.1.1

Der Algorithmus läuft in $O(n^2)$.
Bei beiden inneren **for**-Blöcke laufen jeweils $n-1$ mal durch, sodass der innere Block insgesamt $2 \cdot (n-1) = 2n-2$ mal ausgeführt wird. Die äußere **for**-Schleife läuft gerade n mal; also wird der Block der Länge $2n-2$ n mal ausgeführt. Dies führt zu einer Laufzeit von $n \cdot (2n-2) = 2n^2 - 2n \in \mathcal{O}(n^2)$.

1.1.2 Aufgabe 3.1.2

Die äußere Schleife läuft zwar bis $2 \cdot n$, da die Zählvariable allerdings in Zweierschritten inkrementiert wird, wird die Schleife n mal durchlaufen. Die innere Schleife wird $n \cdot (n-1)$ mal durchlaufen $(1+3+5+7+\cdots+2 \cdot n-3+2 \cdot n-1 = n \cdot (n-1)$. Dies ergibt insgesamt eine Laufzeit von $n \cdot n \cdot (n-1) = n^3 - n^2 \in \mathcal{O}(n^3)$

1.1.3 Aufgabe 3.1.3

Die äußere Schleife wird $\sqrt{n}$ mal durchlaufen, da $i^2 = n \Rightarrow i = \sqrt{n}$. Zeilen 5-7 laufen in $\log n$. Das ergibt eine Gesamtlaufzeit in $\mathcal{O}(\sqrt{n} \cdot \log n)$

1.2 Übungsaufgabe 3.2

[| 2]

Gegeben sei folgender Algorithmus, der ein Array A als Eingabe erwartet (dessen genaue Funktionsweise nachfolgend aber nicht wichtig ist):

Algorithm 1 FUNC(A)

```
1:  if A.lnge < 4 then
2:     return 4
3:  else
4:     sum = 0
5:     for i = 1 to A.laenge do
6:        sum = sum + A[i]
7:     end for
8:     x = A.laenge/4
9:     y =FUNC(A[x + 1..2 · x)
10:    z =FUNC(A[3 · x + 1..A.laenge])
11:    r = y
12:    for i = 1 to A.laenge do
13:       r = r + z · A[i]
14:    end for
15:    return sum + r
16: end if
```

Leiten Sie eine Rekurrenzgleichung für die Laufzeit der Methode FUNC in Abhängigkeit von der Arraygröße n von A ab. Begründen Sie ihre Gleichung.

$$A(n) := \begin{cases} 5, & \text{falls } n < 4 \\ A(\frac{n}{2}) + A(\frac{n}{4}) + 2n + 4, & \text{sonst} \end{cases}$$

Die 5 im ersten Teil der Rekurrenzgleichung erklärt sich durch die ersten beiden Zeilen im Pseudocode.

Der Aufruf in Zeile 8 dauert $\frac{n}{2}$, der in Zeile 9 dauert $\frac{n}{4}$. Dadurch erklärt sich das $A(\frac{n}{2}) + A(\frac{n}{4})$ im zweiten Teil der Rekurrenzgleichung.

Zeile 5 und 6 laufen in n ab.
Zeile 11 und 12 laufen auch in n ab.
Zeile 4, 7, 10 und 13 laufen jeweils mit konstantem Zeitaufwand.
Deshalb ist das, was hinter den $A(x)$-Aufrufen steht, $n + n + c$, wobei c in diesem Fall den Wert 4 hat, da wir vier Zeilen mit konstantem Zeitaufwand im Pseudocode haben.

1.3 Übungsaufgabe 3.3

[| 3]

Gegeben seien die Folgenden Rekurrenzgleichungen T_1, T_2 und T_3, wobei die c_i und d_i Konstanten seien.

$$T_1(n) := \begin{cases} c_1 & \text{für } n = 1 \\ 8 \cdot T_1(\frac{n}{2}) + d_1 \cdot n^3, & \text{sonst} \end{cases}$$

$$T_2(n) := \begin{cases} c_2 & \text{für } n = 1 \\ 5 \cdot T_2(\frac{n}{4}) + d_2 \cdot n^2, & \text{sonst} \end{cases}$$

$$T_3(n) := \begin{cases} c_3 & \text{für } n = 1 \\ 6 \cdot T_3(\frac{n}{3}) + d_3 \cdot n \cdot \log n, & \text{sonst} \end{cases}$$

Bestimmen Sie die Größenordnungen der Funktionen $T_i : \mathbb{N} \to \mathbb{N}$ mittels des Mastertheorems. Ist dies nicht möglich, geben Sie an warum. Geben Sie Ihre Ergebnisse nachvollziehbar an.

1.3.1 Aufgabe 3.3.1

Die Form

$$S(n) := \begin{cases} c, & \text{falls } n = 1 \\ a \cdot T(\frac{n}{b}) + f(n), & \text{falls } n > 1 \end{cases}$$

ist bei

$$T_1(n) := \begin{cases} c_1, & \text{für } n = 1 \\ 8 \cdot T_1(\frac{n}{2}) + d_1 \cdot n^3, & \text{sonst} \end{cases}$$

eingehalten. Das Mastertheorem ist daher anwendbar.

I. $T_1(n) \in \Theta(n^{log_b(a)})$, falls $f(n) \in O(n^{log_b(a)-\epsilon})$ für ein $\epsilon > 0$.

$$\begin{array}{ll} f(n) & \in O(n^{log_b(a)-\epsilon}) \\ d_1 \cdot n^3 & \in O(n^{log_2(8)-\epsilon}) \\ d_1 \cdot n^3 & \in O(n^{3-\epsilon}) \end{array}$$

Hierfür kann kein ϵ gefunden werden. Daher gilt diese Aussage nicht.

II. $T_1(n) \in \Theta(n^{log_b(a)} \cdot log_2(n))$, falls $f(n) \in \Theta(n^{log_b(a)})$.

$$\begin{array}{ll} f(n) & \in O(n^{log_b(a)}) \\ d_1 \cdot n^3 & \in O(n^{log_2(8)}) \\ d_1 \cdot n^3 & \in O(n^3) \end{array}$$

Dies stimmt, daher gilt diese Aussage.

III. $T_1(n) \in \Theta(f(n))$, falls $f(n) \in \Omega(n^{log_b(a)+\epsilon})$ für ein $\epsilon > 0$ **und** $a \cdot f(\frac{n}{b}) \leq \delta \cdot f(n)$ für ein $\delta < 1$ und große n.

$$\begin{array}{ll} f(n) & \in \Omega(n^{log_b(a)+\epsilon}) \\ d_1 \cdot n^3 & \in \Omega(n^{log_2(8)+\epsilon}) \\ d_1 \cdot n^3 & \in \Omega(n^{3+\epsilon}) \end{array}$$

Dies stimmt für alle $\epsilon \geq 0$, also auch für mindestens ein $\epsilon > 0$.

$$\begin{array}{ll} a \cdot f(\frac{n}{b}) \leq \delta \cdot f(n) & \backslash\text{einsetzen} \\ 8 \cdot d_1 \cdot (\frac{n}{2})^3 \leq \delta \cdot d_1 \cdot n^3 & \backslash \sqrt[3]{()} \\ \sqrt[3]{d_1} \cdot 2 \cdot \frac{n}{2} \leq \sqrt[3]{\delta} \cdot \sqrt[3]{d_1} \cdot n & \backslash: \sqrt[3]{d_1} \\ n \leq \sqrt[3]{\delta} \cdot n & \backslash: \text{n (n ist immer positiv, da } n \in \mathbb{N}, \text{ s. Aufgabenstellung)} \\ 1 \leq \sqrt[3]{\delta} & \backslash()^3 \\ 1 \leq \delta & \end{array}$$

Damit ist $\delta \geq 1$ und nicht, wie benötigt, $\delta < 1$. Daher gilt diese Aussage nicht.

Da nur II. gilt, gilt $T_1(n) \in \Theta(n^{log_b(a)} \cdot log_2(n))$, also $T_1(n) \in \Theta(n^3 \cdot log_2(n))$.

1.3.2 Aufgabe 3.3.2

Die Form

$$S(n) := \begin{cases} c, & \text{falls } n = 1 \\ a \cdot T(\frac{n}{b}) + f(n), & \text{falls } n > 1 \end{cases}$$

ist bei

$$T_2(n) := \begin{cases} c_2, & \text{für } n = 1 \\ 5 \cdot T_2(\frac{n}{4}) + d_2 \cdot n^2, & \text{sonst} \end{cases}$$

eingehalten. Das Mastertheorem ist daher anwendbar.

I. $T_2(n) \in \Theta(n^{log_b(a)})$, falls $f(n) \in O(n^{log_b(a)-\epsilon})$ für ein $\epsilon > 0$.

$$\begin{array}{ll} f(n) & \in O(n^{log_b(a)-\epsilon}) \\ d_2 \cdot n^2 & \in O(n^{log_4(5)-\epsilon}) \\ d_2 \cdot n^2 & \in O(n^{1.160964047443681-\epsilon}) \end{array}$$

Hierfür kann kein ϵ gefunden werden. Daher gilt diese Aussage nicht.

II. $T_2(n) \in \Theta(n^{log_b(a)} \cdot log_2(n))$, falls $f(n) \in \Theta(n^{log_b(a)})$.

$$\begin{array}{ll} f(n) & \in O(n^{log_b(a)}) \\ d_2 \cdot n^2 & \in O(n^{log_4(5)}) \\ d_2 \cdot n^2 & \in O(n^{1.160964047443681}) \end{array}$$

Dies stimmt nicht, daher gilt diese Aussage nicht.

III. $T_2(n) \in \Theta(f(n))$, falls $f(n) \in \Omega(n^{log_b(a)+\epsilon})$ für ein $\epsilon > 0$ **und** $a \cdot f(\frac{n}{b}) \leq \delta \cdot f(n)$ für ein $\delta < 1$ und große n.

$$\begin{array}{ll} f(n) & \in \Omega(n^{log_b(a)+\epsilon}) \\ d_2 \cdot n^2 & \in \Omega(n^{log_4(5)+\epsilon}) \\ d_2 \cdot n^2 & \in \Omega(n^{1.160964047443681+\epsilon}) \end{array}$$

Dies stimmt für alle $\epsilon \geq 2 - log_4(5) \approx 0.839035952556319$, also auch für mindestens ein $\epsilon > 0$.

$$\begin{array}{ll} a \cdot f(\frac{n}{b}) \leq \delta \cdot f(n) & \backslash\text{einsetzen} \\ 4 \cdot d_2 \cdot (\frac{n}{5})^2 \leq \delta \cdot d_2 \cdot n^2 & \backslash\sqrt{()} \\ \sqrt{d_2} \cdot 2 \cdot \frac{n}{5} \leq \sqrt{\delta} \cdot \sqrt{d_2} \cdot n & \backslash: \sqrt{d_2} \\ \frac{2}{5} \cdot n \leq \sqrt{\delta} \cdot n & \backslash:\text{n (n ist immer positiv, da } n \in \mathbb{N}\text{, s. Aufgabenstellung)} \\ \frac{2}{5} \leq \sqrt{\delta} & \backslash()^2 \\ \frac{4}{25} \leq \delta & \\ 0.16 \leq \delta & \end{array}$$

Damit gilt $\delta \geq 0.16$. Somit wurde, wie benötigt, mindestens ein $\delta < 1$ gefunden ($0.16 \leq \delta < 1$). Daher gilt diese Aussage.

Da nur III. gilt, gilt $T_2(n) \in \Theta(f(n))$, also $T_2(n) \in \Theta(d_2 \cdot n^2)$, also $T_2 \in \Theta(n^2)$.

1.3.3 Aufgabe 3.3.3

Die Form

$$S(n) := \begin{cases} c, & \text{falls } n = 1 \\ a \cdot T(\frac{n}{b}) + f(n), & \text{falls } n > 1 \end{cases}$$

ist bei

$$T_3(n) := \begin{cases} c_3, & \text{für } n = 1 \\ 6 \cdot T_3(\frac{n}{3}) + d_3 \cdot n \cdot log(n), & \text{sonst} \end{cases}$$

eingehalten. Das Mastertheorem ist daher anwendbar.

I. $T_3(n) \in \Theta(n^{log_b(a)})$, falls $f(n) \in O(n^{log_b(a)-\epsilon})$ für ein $\epsilon > 0$.

$$\begin{array}{ll} f(n) & \in O(n^{log_b(a)-\epsilon}) \\ d_3 \cdot n \cdot log(n) & \in O(n^{log_3(6)-\epsilon}) \\ d_3 \cdot n \cdot log(n) & \in O(n^{1.6309297535714573-\epsilon}) \end{array}$$

Es gilt: $d_3 \cdot n \cdot log(n) < n^{log_3(6)}$ für alle $n > 0$, für $d_3 = 1$.
Da aber nur $d_3 \cdot n \cdot log(n) \leq n^{log_3(6)}$ für alle $n > 1$ (s. Definition T_3) gefordert ist, kann hierfür mindestens ein ϵ gefunden werden.
Für größere d_3 gilt allerdings, dass die n nicht mehr beliebig größer 1 sein dürfen, sondern eine obere Schranke bekommen.
Somit hängt die Lösung stark von d_3 ab und das Mastertheorem ist ungeeignet für die Lösung dieser Aufgabe.

1.4 Übungsaufgabe 3.4

[| 8]

1. Geben Sie für die Funktion *merge* in *mergesort* (siehe Folie 9, Kapitel 3) sinnvollen Pseudocode an. Orientieren Sie sich dabei an der natürlichsprachlichen Formulierung auf Folie 10, Kapitel 3. Ihr Pseudocode sollte diese Formulierung möglichst nah umsetzen. Weisen Sie dann Ihren Pseudocode von *merge* als korrekt nach, indem Sie eine sinnvolle Schleifeninvariante formulieren und nutzen. Nutzen Sie dies dann, um auch *mergesort* selbst als korrekt nachzuweisen. (Siehe auch Folie 11, Kapitel 3.)

2. Gegeben sei eine Sequenz $A = \langle a_1, a_2, \ldots, a_n \rangle$ von Zahlen (als Array). Ein Paar (i, j) mit $i < j$, aber $a_i > a_j$ nennen wir einen *Konflikt* (in A). Gesucht ist nun, gegebendas Array A, die Anzahl der Konflikte in A. Ihre Aufgabe ist es, einen Divide&Conquer Algorithmus zu entwerfen, der das Problem in $\mathcal{O}(n \cdot \log n)$ löst. Beschreiben Sie zunächst Ihre Idee. Geben Sie dann den Algorithmus konkreter an und begründen Sie dann überzeugend, warum Ihr Algorithmus das Problem korrekt löst (sie müssen dazu nicht unbedingt auf die Schleifeninvarianten zurückgreifen). Zuletzt begründen Sie auch, warum die Laufzeit ihres Algorithmus tatsächlich in $\mathcal{O}(n \cdot \log n)$ liegt.

1.4.1 Aufgabe 3.4.1

```
MERGE(A, B)
 1  cur_A = 0
 2  cur_B = 0
 3  C = newlist
 4  while cur_A < A.length  &  cur_B < B.length
 5        if cur_A < A.length
 6              a_i = A[cur_A]
 7        if cur_B < B.length
 8              b_j = B[cur_B]
 9        if a_i < b_j
10              C.append a_i
11              cur_A = cur_A + 1
12        else
13              C.append b_j
14              cur_B = cur_B + 1
15  if cur_B = B.length
16        for i = cur_A to A.length
17              C.append A[i]
18  else
19        for j = cur_B to B.length
20              C.append B[j]
21  return C
```

Korrektheit von MERGE(A, B):
Schleifeninvariante:
Zu Beginn jeder Iteration sind die Listen A, B und C in sich sortiert.

Initialisierung:
Vor der ersten Iteration der **while**-Schleife sind die Listen A und B geordnet, weil sie geordnet „angeliefert“ werden. Die Liste C ist geordnet, weil sie zu diesem Zeitpunkt noch leer ist.

Fortsetzung:
Bei jeder Iteration wird jeweils das kleinste Element aus beiden Listen der Liste C hinzugefügt. Die Listen A und B verlieren kein Element, also sind sie immer noch sortiert. Die Liste C bekommt immer ein neues Element, welches größer oder gleich dem letzten ist. Somit bleibt C auch bei jeder Iteration sortiert.

Terminierung:
Die **while**-Schleife terminiert stets, da bei jeder Iteration mindestens einer der beiden Zeiger um eine Stelle in Richtung Listenende verrückt wird. Damit kommt ein Zeiger - sofern die Listen endlich sind - irgendwann am Ende an.

Neue Initialisierung:
Die **for**-Schleife in Zeile 16 oder 19 verletzt auch nicht die Invariante. Zu Beginn sind alle drei Listen sortiert, aber B - oder A - wurde schon vollständig an C angefügt. Somit sind alle Elemente der verbleibenden Liste A - oder B - größer oder gleich dem letzten - und größten - Element in C.

Neue Fortsetzung:
Da alle Elemente der verbleibenden Liste A - oder B - größer oder gleich jedem Element in C sind, können wir einfach das nächste - und damit kleinste - verbleibende Element aus A - oder B - an C anfügen. Bei

diesem Schritt bleibt A - oder B - sortiert, weil sie nicht verändert wird, und C auch, da sie nur ein weiteres größeres Element angesetzt bekommt.

Neue Terminierung:
Die **for**-Schleife terminiert auch, da der Zähler bei jeder Iteration um 1 erhöht wird. Solange die Liste A - oder B - also nicht unendlich ist, erreicht der Zähler irgendwann ihr Ende.

Wenn die Schleifen abgebrochen sind, gilt, dass die Listen A, B und C sortiert sind. Die Ausgabeliste C also auch. Dies wollten wir zeigen, damit ist dieser Algorithmus korrekt.

Korrektheit von MERGESORT(A, L, R):
Induktion:
Induktion über die Größe der Eingabeliste:

Induktionsanfang:
Hat die Liste die Länge 1, so ist $l = r$. Die Ursprungsliste wird nicht verändert, womit MERGESORT korrekt ist. Denn einelementige Listen sind immer sortiert.

Induktionsannahme:
Ist MERGESORT für die Länge $n \in \mathbb{N}$ wahr, so ist es auch für die Länge $n + 1$ wahr.

Induktionsschritt:
Durch den rekursiven Aufruf, wird MERGESORT(n) für jedes $n > 1$ zu zwei Aufrufen von MERGESORT($\frac{n}{2}$). Somit wird es bei jedem Rekursionsschritt halb so groß und kommt damit irgendwann bei x Aufrufen von MERGESORT(1) an. Dieses gilt nach Induktionsanfang.

1.4.2 Aufgabe 3.4.2

Idee:
Wir nehmen MERGESORT und jedes mal, wenn wir 2 Elemente tauschen müssen, inkrementieren wir den Zähler um 1. Wenn Die Liste sortiert wurde, dann sollte der Zähler die Anzahl der Konflikte anzeigen.

Pseudocode:

```
MERGECOUNT(A, L, R)
1  Zahler = 0
2  if l < r
3        q = (l * r)/2
4        MERGECOUNT(A, L, Q)
5        MERGECOUNT(A, Q +1, R)
6        B = newArray (A[l] to A[q])
7        C = newArray (A[q + 1] to A[r])
8        MERGECOUNT-COUNTER(B, C, ZAEHLER)
9  return Zaehler
```

MERGECOUNT-COUNTER(A, B, ZAEHLER)

```
1  C = newArray
2  while A.length > 0  &  B.length > 0
3      if A[0] > B[0]
4          C.append(B[0])
5          B = cdr(B)
6          Zaehler = Zaehler + A.length
7      else
8          C.append(A[0])
9          A = cdr(A)
10 while A.length > 0
11     C.append(A[0])
12     A = cdr(A)
13 while B.length > 0
14     C.append(B[0])
15     B = cdr(B)
16 return C, Zaehler
```

Korrektheit:

Da die Schleifeninvariante nicht unbedingt nötig ist (s. Aufgabenstellung) folgt hier die rein logische Begründung:
Wir teilen die Liste in der Mitte in 2 gleiche Teile, ohne die Reihenfolge der Elemente zu verändern. Dies tun wir so lange, bis nur noch einelementige Listen da sind. Einelementige Listen sind immer sortiert. Diese Listen werden nun wie folgt zusammengefügt:
Sollte das kleinste - also erste - Element in der rechten Liste (B) kleiner sein, als das kleinste - also erste - Element in der linken Liste (A), dann steht das Element aus B mit allen Elementen aus A in Konflikt. Wir erhöhen also den Konfliktzähler um die Anzahl der Elemente in Liste A.
Wenn allerdings das erste Element von A kleiner ist, als das erste Element von B, existiert kein Konflikt und der Zähler bleibt unangetastet.
Ist eine Liste leer, so sind die restlichen Elemente der anderen Liste nach Vorgehensweise des Verfahrens allesamt größer als das letzte Element der, nun leeren, Liste und sortiert, weswegen sie keine Konflikte mehr vorweisen und direkt an das Ergebnis angehängt werden können. Somit bleibt auch hier der Zähler unangetastet.
Oben beschriebenes passiert bei jeder Rekursion aufs Neue, weshalb der Zähler kontinuierlich weiterwächst - sofern Konflikte vorhanden.
Der Algorithmus terminiert zudem immer, da MERGESORT ebenfalls immer terminiert und sich MERGECOUNT nur im Erhöhen des Zählers von MERGESORT unterscheidet.

Zeitaufwand:

MERGECOUNT und MERGECOUNT-COUNTER verhalten sich genau, wie MERGESORT und MERGE, bis auf eine Erhöhung eines Zählers, was allerdings mit konstantem Zeitaufwand zu bewältigen ist und daher bei der O-Notation nicht ins Gewicht fällt. Somit liegt MERGECOUNT - genauso wie MERGESORT - ebenfalls in $\mathcal{O}(n \cdot log(n))$.