RWTH RHEINISCH-WESTFÄLISCHE TECHNISCHE HOCHSCHULE AACHEN

Lehrstuhl für Informatik IX
Datenmanagement und Exploration
Prof. Dr. T. Seidl

Präsenzübung
Datenstrukturen und Algorithmen
8. 6. 2009

Vorname: _____

Nachname: _____

Matrikelnummer: _____

Studiengang : (bitte ankreuzen)

- ○ CES: Bachelor
- ○ Informatik: Bachelor
- ○ Informatik: Diplom
- ○ Lehramt: Reguläres Fach
- ○ Sonstige: _____
- ○ Mathe: Bachelor

- Schreiben Sie auf jedes Blatt **Vorname**, **Name** und **Matrikelnummer**.
- Geben Sie Ihre Antworten in lesbarer und verständlicher Form an.
- Bitte beantworten Sie die Aufgaben auf den **Aufgabenblättern** (benutzen Sie auch die Rückseiten). Antworten auf Zusatzblättern können nur berücksichtigt werden wenn **Name**, **Matrikelnummer** und **Aufgabennummer** deutlich darauf erkennbar ist.
- Verwenden Sie für die Bearbeitung der Aufgaben **kein eigenes** Papier
- Was nicht bewertet werden soll, kennzeichnen Sie bitte durch **Durchstreichen**.
- Werden Täuschungsversuche beobachtet, so wird die Präsenzsübung mit **nicht bestanden** bewertet. Es sind keine Hilfsmittel zugelassen.
- Geben Sie am Ende **alle Blätter zusammen mit den Aufgabenblättern** ab. Trennen Sie die Blätter nicht durch Entfernung der Heftung.

	erreichbare Punktzahl	erreichte Punktzahl
Aufgabe 1	11	ok
Aufgabe 2	5	ok
Aufgabe 3	6	ok
Aufgabe 4	6	ok
Aufgabe 5	10	ok
Aufgabe 6	5	ok

	erreichbare Punktzahl	erreichte Punktzahl
Aufgabe 7	8	ok
Aufgabe 8	6	ok
Aufgabe 9	7	ok
Aufgabe 10	4	ok
Summe	68	

Vorname	Name	Matr.-Nr.

(1)

Aufgabe 1: (Rekursionsgleichungen) **3+4+4 Punkte**

a) Geben Sie mittels Mastertheorem eine möglichst enge obere Schranke für die folgenden zwei Rekursionsgleichungen an. Tragen Sie dazu in der folgenden Tabelle die Werte für a, b, γ und die Komplexitätsklasse O(T(n)) in nicht-rekursiver Form ein. Rekursionsanfang T(1) = 1 bzw. T(0) = 1.
Zur Information: Das erweiterte Master-Theorem lautet:

- Rekursion

$$T(n) = \begin{cases} c & ,n = 1 \\ a \cdot T\left(\frac{n}{b}\right) + d(n) & ,n > 1 \end{cases}$$ mit d(n) $\in$ O(n^{γ}), $\gamma > 0$

- Dann:

$$T(n) = \begin{cases} O(n^{\gamma}) & a < b^{\gamma} \\ O(n^{\gamma} \log_b n) & a = b^{\gamma} \\ O(n^{\log_b a}) & a > b^{\gamma} \end{cases}$$

	a	b	γ	O(T(n))
T(n) = 2 T(n/4) + $\sqrt{n}$	2	4	½	$O(n^{\frac{1}{2}} \log_4 n)$
T(n)=16 T(n/2)+(n+1)(n+2)^2	16	2	3	$O\left(n^{\log_2 16}\right) = O(n^4)$

b) Gegeben sei die folgende Rekursionsgleichung. Geben Sie die Komplexitätsklasse einer möglichst engen oberen Schranke an.

T(n) = 4 T(n-2) + 3 mit T(0)=1 und n gerade und n≥2

Schnelle Lösung: Das n in der Funktion T(n) verringert sich immer um 2, T vervierfacht aber seinen Wert dabei um 4. Die Vermutung liegt nahe, dass sich die Funktion durch $a * 2^n + b$ ergibt.
Auf diese Idee kommt man auch, wenn man T(n) wie folgt schreibt:
$T(n) = 2^2 * T(n-2) + 2^2 - 1$
Nun kann man durch Einsetzen auf die Lösung kommen:
$T(n) = a * 2^n + b$ sowie

$$T(n) = 4 * T(n-2) + 3 = 4 * (a * 2^{n-2} + b) + 3 = a * 2^2 * 2^n * 2^{-2} + 4 * b + 3$$
$$= a * 2^n + 4b + 3$$

Koeffizientenvergleich liefert, dass für b gelten muss: $b = 4b + 3 \rightarrow b = -1$
Da T(0)=1 sein muss, folgt $T(0) = a * 2^0 - 1 = a - 1 = 1 \rightarrow a = 2$

Vorname	Name	Matr.-Nr.

(2)

Ingesamt ergibt sich also $T(n) = 2 * 2^n - 1 = 2^{n+1} - 1 \in O(2^n)$

c) Gegeben sei die folgende Rekursionsgleichung. Geben Sie die Komplexitätsklasse einer möglichst engen oberen Schranke an.

$$T(n)=T\left(\left\lceil\frac{n}{2}\right\rceil\right)+T\left(\left\lceil\frac{n-1}{2}\right\rceil\right)+\frac{n(n-1)}{2} \quad \text{mit } T(1)=1$$

$$T(n) \leq 2 * T\left(\left\lceil\frac{n}{2}\right\rceil\right)+\frac{n(n-1)}{2}$$

→ Master Theorem: a=2 b=2 gamma=2
→ $a < b^{\gamma}$
→ T(n) ist in $O(n^2)$

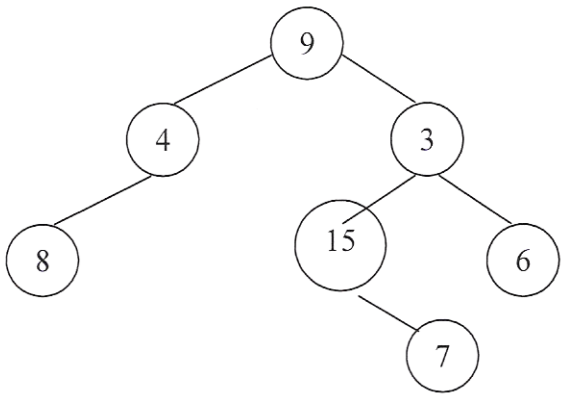
Vorname	Name	Matr.-Nr.

(3)

Aufgabe 2 (Binärbäume) **5 Punkte**

Geben Sie den binären Baum an, dem die folgenden InOrder- bzw. PostOrder-Traversierungen eindeutig entsprechen:

InOrder	8, 4, 9, 15, 7, 3, 6
PostOrder	8, 4, 7, 15, 6, 3, 9



Vorname	Name	Matr.-Nr.

(4)

Aufgabe 3 (Komplexität) **6 Punkte**

Gegeben sei die folgende Methode.

1. Geben Sie kurz an, welche Funktionalität implementiert wird.
2. Geben Sie dann eine möglichst enge obere Schranke für die Zeitkomplexität bzgl. *n* an und begründen Sie diese ebenfalls kurz.

```
String lustigerString(int n){
    if(n == 0)
        return "0";
    else if(n == 1)
        return "1";
    else
       return lustigerString(n/2) + "" + lustigerString(n%2);
                                          //n%2 berechnet n modulo 2
}
```

Lösung:

Es wird die Binärdarstellung einer ganzen Zahl ausgegeben. Dazu wird sukzessive durch 2 geteilt und für Rest 1 eine „1“ in den Ergebnisstring geschrieben, sonst „0“.

Die Rekursionsgleichung ist dann T(n) = T(n/2) + 1, für den Rekursionsaufruf mit der Hälfte und den Rekursionsaufruf mit dem Rest (da T(0)=T(1)=1). Zu beachten ist, dass der modulo Aufruf nur konstante Zeit braucht, da n%2 stets 0 oder 1 ist. Insgesamt ergibt sich daher T(n) = O(ld n).

3 Punkte für das Erkennen der Funktion
2 Punkte für die Begründung (z.B. Rekursionsgleichung)
1 Punkt für das Ergebnis

Vorname	Name	Matr.-Nr.

(5)

Aufgabe 4 (Komplexität) 6 Punkte

Gegeben sei die folgende Java-Routine:

```
int berechne(int links, int rechts) {
      int diff = rechts - links;

      if(diff <= 1) {
            return 1;
      }

      int i = berechne(links, links+ (diff/2));
      int j = berechne(links + (diff/4) + 1, rechts - (diff/4));
      int k = berechne(rechts - (diff/2), rechts);

      int summe = 0;

      for(int m=1; m<=2*diff; m=m+1) {
            summe = summe + (max(i,k)+j)*m;
      }

      return summe;
}
```

Nehmen Sie an, dass die Grundrechenarten +, -, *, / in konstanter Zeit O(1) ausgeführt werden, ebenso wie Zuweisungen = und Vergleiche <=.

In welcher Komplexitätsklasse (in Abhängigkeit von n) liegt der Aufruf berechne(0,n)?

Lösung: Am anschaulichsten ist es, wenn man sich die Aufrufe als Intervalle vorstellt, zu Beginn hat man das Intervall 0....n Grafisch mal wie folgt dargestellt (für n=9):
0 1 2 3 4 5 6 7 8 9
für die Berechnung von i benötigt man dann nur noch die erste Hälfte des Intervalls (links bis diff/2)
0 1 2 3 4 x x x x x
für die Berechnung von j dann eine andere Hälfte des Intervalls und zwar
x x x 3 4 5 6 7 x x
und für k dann die letzte Hälfte
x x x x x 5 6 7 8 9

Insgesamt hat man für diesen Teil des Programms bei der Eingabe der Länge der n die Aufrufe 3*T(n/2)

Für die Komplexität muss dann ferner noch die Schleife beachtet werden. Diese läuft genau 2*diff=2*n mal. Innerhalb der Schleife hat man nur konstanten Aufwand durch die Additionen etc.

Die finale Rekursionsgleichung ist somit: T(n)=3*T(n/2)+2*n
Durch Anwendung des Master-Theorems ergibt sich $O(n^{\log_2 3})$

Vorname	Name	Matr.-Nr.

(6)

Aufgabe 5 (O-Notation) **1+3+3+3 Punkte**

Beweisen oder widerlegen Sie die folgenden Aussagen:

a) $f \in O(h) \wedge g \in O(h) \Rightarrow f \in O(g)$

Diese Aussage gilt nicht, wie das folgende einfache Beispiel zeigt:

Sei $f(n) = n$, $h(n) = n$ und $g(n) = 1$, dann gilt:
$f \in O(h)$ und $g = 1 \in O(h)$ aber nicht $f \in O(g) = O(1)$.

b) $n^2 + 2n + 220 \in O(n^2)$

Diese Aussage gilt. Sei $c = 223$, dann gilt:

$n^2 + 2n + 220 < c * n^2$, für alle $n \in \mathbb{N}$ und somit $n^2 + 2n + 220 \in O(n^2)$

c) $f \in O(g) \Rightarrow f^2 \in O(g^2)$

Diese Aussage ist korrekt. Beweis:

$$
\begin{aligned}
& f \in O(g) \\
\Rightarrow\ & \exists c \in \mathbb{N}, \forall n \in \mathbb{N}: f(n) \leq c * g(n) && \mid\ ^2 \\
\Rightarrow\ & \exists c \in \mathbb{N}, \forall n \in \mathbb{N}: f^2(n) \leq c^2 * g^2(n) && \mid\ c_2 = c^2 \\
\Rightarrow\ & \exists c_2 \in \mathbb{N}, \forall n \in \mathbb{N}: f^2(n) \leq c_2 * g^2(n) \\
\Rightarrow\ & f^2 \in O(g^2)
\end{aligned}
$$

d) $O(n^n) = O((n-1)^{n+1})$

Diese Aussage gilt nicht.
Beweis durch Gegenbeweis. Angenommen die Aussage wäre korrekt, dann gäbe es ein $c \in \mathbb{N}$, sodass für alle $n \in \mathbb{N}$ gilt:

$$
\begin{aligned}
& (n-1)^{n+1} \leq c * n^n && \mid n^n > 0\ \forall n \in \mathbb{N} \\
\Rightarrow\ & c \geq \frac{(n-1)^{n+1}}{n^n}, \quad \forall n \in \mathbb{N}
\end{aligned}
$$

Da jedoch $\lim_{n\to\infty} \frac{(n-1)^{n+1}}{n^n} = \infty$, kann es kein derartiges c geben.
Somit ergibt sich ein Widerspruch zur Annahme, dass die Aussage korrekt ist.
q.e.d.

Vorname	Name	Matr.-Nr.

(7)

Aufgabe 6 (Dynamische Programmierung) **5 Punkte**

Die in der Übung vorgestellte Edit-Distanz $d(s_{|s|}, q_{|q|})$ zweier Buchstaben-Sequenzen s und q ist die minimale Anzahl Operationen (Umbenennen, Einfügen und Entfernen einzelner Zeichen), die benötigt werden, um s in q zu überführen.
Zu einer Sequenz s bezeichne s_i die Sequenz aus den ersten i Zeichen von s.

Rekursionsgleichung: $$d(s_i, q_j) = \begin{cases} j & i = 0 \\ i & j = 0 \\ d(s_{i-1}, q_{j-1}) & s[i] = q[j] \\ \min\begin{cases} d(s_{i-1}, q_{j-1}) \\ d(s_i, q_{j-1}) \\ d(s_{i-1}, q_j) \end{cases} + 1 & sonst \end{cases}$$

Geben Sie die Berechnung der Edit-Distanz für die Sequenzen FOLIE und FLOYD an. Verwenden Sie dafür das Prinzip der dynamischen Programmierung.

Lösung:

a)

			f	o	l	i	e
		0	1	2	3	4	5
f		1	0	1	2	3	4
l		2	1	1	1	2	3
o		3	2	1	2	2	3
Y		4	3	2	2	3	3
D		5	4	3	3	3	4

$d(s_{|s|}, q_{|q|}) = d(\text{FOLIE},\text{FLOYD}) = 4.$

4 Punkte für Teil a – für jeden Fehler 0,5 Punkte Abzug (keine negativen Punkte)
Folgefehler kosten nix

Vorname	Name	Matr.-Nr.

(8)

Aufgabe 7: **4 + 4 Punkte**

Gegeben sei folgendes Array:

4, 3, 8, 2, 5, 9, 6, 7

a) Sortieren sie das Array mit Hilfe von MergeSort.
In Ihrer Darstellung müssen die wesentlichen Schritte des Algorithmus erkennbar sein.

4	3	8	2	5	9	6	7
3	4	2	8	5	9	6	7
2	3	4	8	5	6	7	9
2	3	4	5	6	7	8	9

b) Sortieren sie das Array mit Hilfe von InsertionSort.
In Ihrer Darstellung müssen die wesentlichen Schritte des Algorithmus erkennbar sein.

4	3	8	2	5	9	6	7
2	3	8	**4**	5	9	6	7
2	**3**	8	4	5	9	6	7
2	3	**4**	**8**	5	9	6	7
2	3	4	**5**	**8**	9	6	7
2	3	4	5	**6**	9	**8**	7
2	3	4	5	6	**7**	8	**9**
2	3	4	5	6	7	**8**	9

Vorname	Name	Matr.-Nr.

(9)

Aufgabe 8: (SelectionSort) **6 Punkte**

Implementieren Sie in Java eine Methode, die ein Feld **absteigend** sortiert. D.h., nach Ausführung der Methode für das Array *a* soll gelten $a[0] \geq a[1] \geq ... \geq a[n-1]$.
Ihre Implementierung soll dabei die Idee des in der Vorlesung vorgestellten SelectionSort-Verfahrens umsetzen. Dabei dürfen Sie *swap(a,i,j)* mit der in der Vorlesung vorgestellten Semantik verwenden (Vertauschen des i-ten und des j-ten Eintrags im Array *a*).

```
public static void selectionSort(int[] a){
```

Lösung:

```
static void selectionR(int[] a){

    for (int i = 0; i <= a.length-2; i++) {

        int max = i;

        for (int j = i+1; j <= a.length - 1 ; j++){
            if  (a[j] > a[max])
                max = j;
        }

        swap(a, i, max);

    }

}
```

6 Punkte für eine korrekte Implementierung, d.h. :

1. Umsetzung des SelectionSort-Ansatzes
   (ist dies nicht erfüllt, so werden 0 Punkte vergeben)
2. Sortierung absteigend
   (ist dies nicht erfüllt, der Rest jedoch korrekt, so werden 3 Punkte vergeben)

Für kleinere Syntaxfehler (z.B. Semikolon vergessen) gibt es keinen Punktabzug. Bei gravierenderen Syntaxfehlern Punktabzug (abhängig vom Einzelfall).

Vorname	Name	Matr.-Nr.

(10)

Aufgabe 9: (Höhere Sortierverfahren) **7 Punkte**

Führen Sie für folgendes Array HeapSort durch.

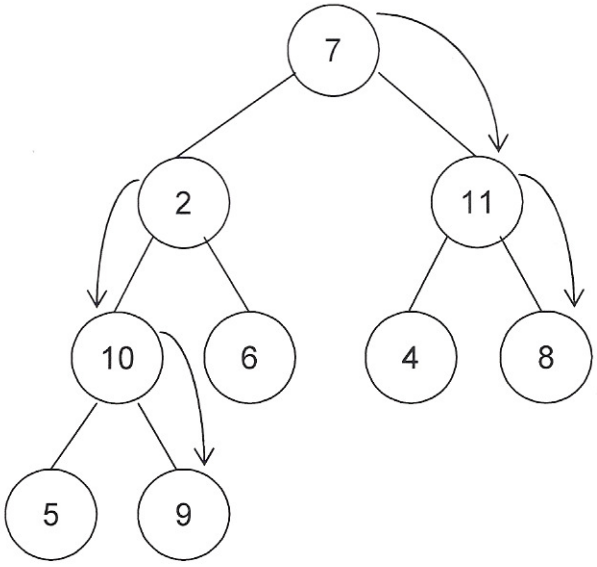
7, 2, 11, 10, 6, 4, 8, 5, 9

a) Stellen Sie dieses Array als Ausgangspunkt für HeapSort in Baumform dar.

b) Tragen Sie für die Heapaufbau-Phase der Reihe nach die einzelnen Versickerungsvorgänge in diesen Baum ein und geben Sie den endgültigen Heap in Arrayform an.

c) Geben Sie die in der Sortierphase auftretenden Heaps sowie das jeweils bereits sortierte Feld an. Hierbei können Sie zwischen Array- und Baum-Darstellung wählen.

a)



b)

11	10	8	9	6	4	7	5	2

c)

10	9	8	5	6	4	7	2	11
9	6	8	5	2	4	7	10	11
8	6	7	5	2	4	9	10	11
7	6	4	5	2	8	9	10	11
6	5	4	2	7	8	9	10	11
5	2	4	6	7	8	9	10	11
4	2	5	6	7	8	9	10	11
2	4	5	6	7	8	9	10	11

Vorname	Name	Matr.-Nr.

Aufgabe 10: (Spezielle Sortierverfahren) **2 +2 Punkte**

a) Erläutern Sie die Unterschiede zwischen dem CountingSort- und dem BucketSort-Verfahren.

Bei CountingSort nutzt man einen Zähler pro Schlüssel während bei BucketSort die Schlüssel in Grupen zusammengefasst werden, um diese durch nur einen Zähler zu repräsentieren.
Man kann CountingSort als Spezialfall von BucketSort verstehen, beidem die Gruppen jeweils aus nur genau einem Schlüssel bestehen.

b) Geben Sie kurz für jedes der beiden Sortierverfahren zwei Anwendungsszenarien an, in denen das Verfahren sinnvoll einsetzbar ist bzw. eher schlecht geeignet ist.

CountingSort:

CountingSort eignet sich z.B. um Ereignisse nach ihrem Wochentag (Montag – Sonntag) zu sortieren.

Mit CountingSort ist es nicht möglich, Telefonbucheinträge nach Namen zu sortieren, da es theoretisch unendlich viele Namen gibt.

BucketSort:

BucketSort eignet sich um z.B. Briefe nach PLZ zu sortieren. PLZ mit gleicher Anfangsziffer werden dann über dieselbe Zählervariable gezählt. Somit erhält man eine Vorsortierung der Briefe die z.B. ausreicht um zu entscheiden an welchen Umschlag-Flughafen der Brief geschickt werden muss.

Wie CountingSort ist auch BucketSort nicht geeignet um eine Vollständige Sortierung nach Namen zu realisieren. Eine Vorsortierung nach dem ersten Buchstaben des Namens wäre aber denkbar.