Klausuraufgaben

Algorithmen und Datenstrukturen

03. Juli 2008

Zur Lösung der nachfolgenden Aufgaben haben Sie 90 Minuten Zeit. Es sind **keine Hilfsmittel** zugelassen!

- Für alle Aufgaben ist der **Lösungsweg** und die Lösung **nachvollziehbar** anzugeben, d.h. Sie müssen mich von Ihrer Lösung und dem Lösungsweg überzeugen! (Dies setzt u.a. eine **saubere Schrift** voraus!). In diesem Sinne werden Antworten ohne Begründung oder zu schlechter Schrift **nicht bewertet**!
- Bitte schreiben Sie zuerst Ihren Namen und Ihre Matrikelnummer zumindest auf das erste Blatt und auf alle losen Blätter zumindest die Matrikelnummer!

Lesen Sie erst alle Aufgaben durch! Sie sind **nicht** nach dem Schwierigkeitsgrad geordnet. Am Ende der Klasur finden Sie Tip's zu den Aufgaben.

Aufgabe	1	2	3	4	5	$\sum$
	Grundlagen	Komplexität	Sortieren	Graphen	Hashing	Summe
mögliche Punkte	22	14	29	20	14	99
erreichte Punkte						

Erreichte Bewertung:

Aufgabe 1 (Grundlagen) *(22P)*

1. **Algorithmus** *(5 Punkte)*

   (a) *Beschreiben Sie den Begriff Algorithmus und erklären Sie die vier allgemeinen Anforderungen an einen solchen Algorithmus.*

   (b) *Erklären Sie folgende vier Eigenschaften eines Algorithmus:* terminiert, determiniert, deterministisch und nicht deterministisch.

2. **Datenabstraktion** *(9 Punkte)*

   (a) *Beschreiben Sie das Prinzip bzw. die Grundidee der Datenabstraktion.*

   (b) *Welche vier Typen von Funktionen benötigt man im Allgemeinen? (unabhängig von der konkreten Datenstruktur als „Schnittstelle") Geben Sie für eine Schlange jeweils ein erklärendes kleines Beispiel an. Es genügt der Name einer Funktion/Methode und eine kurze Erklärung, was sie macht.*

   (c) *In der Vorlesung wurde ein Entwurf einer Schlange vorgestellt, deren Funktionalität mit zwei Stapeln realisiert wurde. Ist es möglich, mit zwei Schlangen die Funktionalität eines Stapels zu realisieren ? Begründen Sie Ihre Antwort, indem Sie auch erklären, wieso die Schlange mittels zwei Stapeln realisiert werden kann!*

3. **Rekursion** *(8 Punkte)*

   (a) *Implementieren Sie folgendes Verfahren in z.B. Java. Dieses Verfahren zur Berechnung der Zahl π ist aus dem Jahr 1976 und stammt von den Herren Salamin und Brent.*

$$a_m := \frac{a_{m-1} + b_{m-1}}{2} \text{ mit } a_0 := 1$$

$$b_j := \sqrt[2]{a_{j-1} * b_{j-1}} \text{ mit } b_0 := \sqrt[2]{0.5}$$

$$\pi_n := \frac{(a_n + b_n)^2}{(1 - \sum_{k=0}^{n}[2^k * (a_k - b_k)^2])}$$

   Beachten Sie: a_n ist gleich der Schreibweise $a(n)$ und damit ist a als Funktion/Methode umzusetzen.

   (b) *Zeigen Sie den Ablauf der rekursiven Aufrufe für π_2 auf. Eine Berechnung ist dabei nicht durchzuführen! Wie oft werden a und b dabei jeweils aufgerufen?*

Aufgabe 2 (Komplexitätstheorie) *(14P)*

1. **Begriffsbestimmung** *(2 Punkte)*
   Beschreiben Sie die drei interessanten Fälle **Bester Fall**, **Schlechtester Fall** *und* **Mittlerer Fall**.

2. **Break-even Punkt** *(6 Punkte) Für die Aufgabe, n Adressen zu sortieren, stehen drei Verfahren zur Verfügung: Eine Implementierung von Heapsort benötigt eine Zeit von* $184 * log_2(n)\mu s$ *, eine Implementierung von Insertion Sort genau* $2 * n^2 \mu s$ *und eine Implementierung von Bubblesort* $n^2 + 8n + 180$ μs*. Für welche Datenbankgrössen n lohnt sich welches der Sortierverfahren? Bestimmen Sie die „Break-even"-Punkte! Für den Break-even-Punkt zwischen Bubblesort und Heapsort kann kurz vor der formalen Auflösung (bzgl.* log_2*) mit den beiden Werten* $n = 22$ *und* $n = 23$ *getestet werden.*

3. **Problembehandlung** *(6 Punkte)*
   In der Vorlesung wurden sechs Möglichkeiten genannt, wie man auf ein komplexes Problem reagieren kann bzw. das Problem handhabbar machen kann. Nennen und erklären Sie diese evtl. auch mit Hilfe eines Beispieles.

Aufgabe 3 (Sortieren) *(29P)*

1. **Teile-und-Herrsche** *(3 Punkte)*

   (a) *Erklären Sie kurz das Teile-und-Herrsche-Prinzip. Beschreiben Sie dazu grob den Ablauf.*

   (b) *Welches Problem könnte bei der Durchführung bestehen ? Denken Sie dabei z.B. an die Praktikumsaufgabe, in der die Teilsumme einer Folge von Zahlen zu bestimmen war (siehe die Tip's am Ende der Klausur dazu).*

2. **Rekursion** *(3 Punkte)*
   Quicksort und Mergesort unterscheidet nicht nur die Anwendung: Quicksort wird zu den „internen Sortierverfahren" gezählt und Mergesort zu den „externen Sortierverfahren".

   Beide arbeiten nach dem Teile-und-Herrsche-Prinzip. Was unterscheidet dennoch beide Algorithmen im Punkt Rekursion? Welche Auswirkung hat dies für eine iterative Variante von Quicksort?

3. **Quicksort** *(8 Punkte)*
   Sotieren Sie die folgende Zahlenreihe mit dem im Tip aufgeführten Quicksort-Algorithmus. Notieren Sie die Zwischenergebnisse nach jeder Tauschoperation.

9	*11*	*40*	*22*	*26*	*43*	*36*	*14*	*4*	*49*

   Geben Sie die Anzahl der Rekursionsaufrufe und die Rekursionstiefe an.

4. **Heapsort** *(15 Punkte)*
   Gegeben sei ein Maximum-Heap, wie in der Vorlesung vorgestellt. Beantworten Sie folgende Fragen und begründen Sie Ihre Antworten:

   (a) *Was versteht man hierbei unter dem Begriff „Heap-Eigenschaft"?*

   (b) *Wie wird die Baumrepräsentation eines Heaps in ein Array kodiert, also wo findet man die Nachfolger eines Knotens im Array?*

   (c) *Was ist die minimale und maximale Anzahl von Elementen in einem Heap der Höhe h?*

   (d) *Wo kann sich in einem Heap das kleinste Element nur befinden?*

   (e) *Welche Elemente in einer unsortierenden Zahlenfolge bilden zu Anfang bereits einen Heap, und warum wird diese Struktur von hinten (rechts) nach vorne (links) in der Array-Repräsentation eines Heaps bzw. von unten nach oben in der Baumrepräsentation eines Heaps als Heap aufgebaut? (Bottom-Up Verfahren)*

Aufgabe 4 (Bäume und Graphen) *(20P)*

1. **Greedy-Algorithmen** *(7 Punkte)*
   Bestimmen Sie für nachfolgendes Problem folgende Elemente des im Tip aufgeführten allgemeinen Greedy-Algorithmus:

   - *Eine Menge von Kandidaten* **C**, *aus der die Lösung konstruiert wird;*
   - *Die Lösungsmenge* **S**$\subseteq C$*;*
   - *Eine boolesche Funktion* **solution**, *die angibt, ob eine Menge von Kandidaten eine legale Lösung des Problems darstellt;*
   - *Eine Testfunktion* **feasible**, *die angibt, ob eine Teillösung noch zu einer kompletten legalen Lösung erweitert werden kann;*
   - *Eine Auswahlfunktion* **select**, *die einen noch unbenutzten Kandidaten liefert, der im Sinne der Greedy-Strategie der erfolgversprechendste ist;*
   - *Eine Zielfunktion* **val**, *die den Wert einer gewissen Lösung angibt.*

   Wenden Sie den daraus resultierenden Algorithmus auf folgendes „Bepackungsproblem“ an: Gegeben sei ein leerer Rucksack mit maximalem Fassungsvermögen von 40kg. Versuchen Sie gemäß Ihrem Algorithmus den Rucksack mit folgenden Teilen zu bepacken: 4kg,9kg,11kg,14kg,22kg,26kg.

   Welche Teile würden Sie „von Hand“ (also ohne erkennbares Prinzip) auswählen ?

2. **AVL-Bäume** *(9 Punkte)*
   Wenn in einem AVL-Baum beim Löschen oder Einfügen an einer Position entdeckt wird, dass die Balance nicht mehr stimmt, so wird um diese Position gemäß den Regeln rotiert. Zeigen Sie: die Höhe dieser Position ist nach der Rotation gleich der Höhe vor dem Einfügen oder Löschen. Es genügt dies für das Einfügen sowie die Rechts- bzw. Linksrotation und die doppelte Rechtsrotation bzw. doppelte Linksrotation zu zeigen.

3. **optimale Wege** *(4 Punkte)*
Führen Sie den Algorithmus von Dijkstra mit dem Graphen in Abbildung 1 (Seite 6) durch (bis zum Abbruch!). Wählen Sie als Ausgangspunkt den Knoten x_1. Fertigen Sie, ähnlich wie in der Vorlesung, eine Dokumentation an, aus der der Ablauf deutlich wird. Bei Wahlmöglichkeiten ist immer die lexikographisch niedrigste Ecke zu wählen.

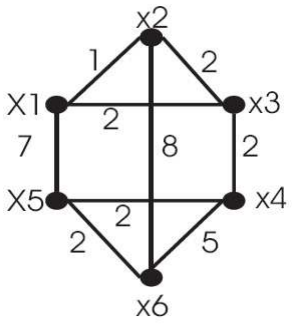

Abbildung 1: Optimaler Weg gesucht!

Aufgabe 5 (Hashing) *(14P)*

1. **Konstruktionsprinzip** *(6 Punkte)*

   (a) *Suchalgorithmen, die mit Hashing arbeiten, bestehen aus zwei Teilen. Welche sind dies?*

   (b) *Welche Ziele verfolgen die einzelnen Teile?*

   (c) *Warum ist Hashing ein gutes Beispiel für einen Kompromiss zwischen Zeit- und Platzbedarf?*

2. **Separate Chaining** *(3 Punkte)*
   Fügen Sie die Zahlen

   $$9, 11, 40, 22, 26, 43, 36, 14, 5$$

   in dieser Reihenfolge unter Verwendung der Hashfunktion $h(k) = k mod 5$ *in eine Hashtabelle der Größe* $N = 5$ *(nummeriert von 0 bis 4) ein. Die Kollisionsbehandlung erfolge durch verkettete Listen.*

   Das Einfügen ist so zu dokumentieren, dass der Ablauf nachvollziehbar ist. Geben Sie z.B. hierzu auch das Ergebnis der Hashfunktion an.

3. **Quadratisches Sondieren** *(5 Punkte)*
   Fügen Sie die Zahlen

   $$9, 11, 40, 22, 26, 43, 36, 14, 5$$

   in dieser Reihenfolge unter Verwendung der Hashfunktion $h(k) = k mod 9$ *in eine Hashtabelle der Größe* $N = 9$ *(nummeriert von 0 bis 8) ein. Die Kollisionsbehandlung erfolge durch quadratisches Sondieren* ($h_i(k) = (h(k) + i^2)$ *mod* 9*).*

   Das Einfügen ist so zu dokumentieren, dass der Ablauf nachvollziehbar ist. Geben Sie z.B. hierzu auch das Ergebnis der Hashfunktion h *bzw. bei Kollisionen der Hashfunktionen* h_i *an.*

Im folgenden sind Hinweise/**Tip**'s zu den Klausuraufgaben gegeben.

- **Teilsumme** Sei $folge := (5, -8, 3, 3, -5, 7, -2, -7, 3, 5)$ eine Folge von ganzen Zahlen. Dann ist eine Teilfolge dieser Folge ein zusammenhängender Teil der Folge. Jede Folge hat zwei extreme Teilfolgen: die leere Teilfolge () und die gesamte Teilfolge (auch als unechte Teilfolge bezeichnet). Teilfolgen wären z.B.: $teilfolge_1 := (3, 3); teilfolge_2 := (-5, 7, -2)$; Keine Teilfolgen wären $keineteilfolge_1 := (5, 3, 3, 7, 3, 5); keineteilfolge_2 := (-8, -5, -2)$; insbesondere darf die ursprüngliche Folge nicht umsortiert werden!

  Gesucht war die Teilfoge mit der maximalen Summe bei Addierung ihrer Elemente.

- **Algorithmus Quicksort**

```
class Datensatz {
      int  key;
      DAT daten;
}
Datensatz a[N]; //wir zählen von 1 bis N

quicksort(int ilinks, int irechts) {
  int pivot,i,j;
  Datensatz tmp;
  if ( irechts > ilinks ) {
     i = ilinks;
     j = irechts-1;
     pivot = a[irechts].key;
     while(1) {
        while((a[i].key < pivot)&&(i < irechts)) i++;
        while((a[j].key > pivot)&&(j > ilinks)) j--;
        if ( i >= j ) break; // verlasse while(1)
        swap(i,j); //vertauschen a[i] mit a[j]
     }
     swap(i,irechts);
     quicksort(ilinks,i-1);
     quicksort(i+1,irechts);
  }
}
```

- **Algorithmus Heapsort**
  Löschen eines Knoten's (Top-Down):

  1. Schreibe das Element mit dem höchste Index an die Position der Wurzel
  2. Vertausche dieses Element so lange mit dem gröSSeren seiner Söhne, bis seine beiden Söhne kleiner sind, oder bis es keine Söhne mehr hat.

  Erstellung eines Heap's (Bottom-Up): Gegeben sei die Folge von Schlüsseln: $k_1, \ldots, k_N$.

  1. In der Reihenfolge $k_{\lfloor \frac{N}{2} \rfloor}, \ldots, k_1$ führe für diese Schlüssel durch:
     (a) Vertausche dieses Element so lange mit dem gröSSeren seiner Söhne, bis seine beiden Söhne kleiner sind, oder bis es keine Söhne mehr hat.

- **Greedy-Algorithmus**

```
function greedy(c) : set
 R := emptyset
 while not solution(R) and C <> emptyset do
 { x := select(C,R)
   C := C - {x}
   if feasible(R + {x}) then
      R := R + {x}
  } // end-while
 if solution(R) then
    return R
   else
    {Es gibt keine Loesung!}
```

- **AVL-Bedingungen** für Rotationen (Siehe Abbildung 2 (Seite 10)):

  1. **Rechtsrotation**: Die Höhe des Teilbaums `R1` ist um 2 niedriger, als die Höhe des Teilbaums mit Wurzel `B`. Der Unterschied sei durch den Teilbaum `L` begründet. Auf dem Teilbaum mit Wurzel `A` wird eine Rechtsrotation durchgeführt.
  2. **Linksrotation**: analog der Rechtsrotation
  3. **Problemsituation links**: (Doppelte Linksrotation) Die Höhe des Teilbaums `L1` ist um 2 niedriger, als die Höhe des Teilbaums mit Wurzel `B`. Der Unterschied sei durch den Teilbaum `L2` begründet. Auf dem Teilbaum mit Wurzel `B` wird eine Rechtsrotation durchgeführt und dann wird auf dem Baum mit Wurzel `A` eine Linksrotation durchgeführt.
  4. **Problemsituation rechts**: (Doopelte Rechtsrotation) analog der Problemsituation links.

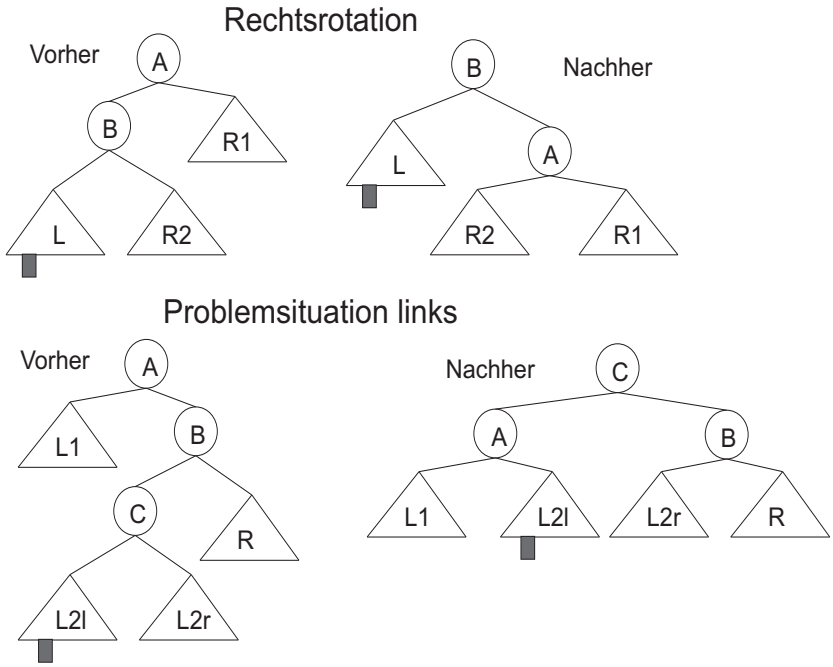

Abbildung 2: Die Rotationsarten eines AVL-Baumes

- **Algorithmus von Dijkstra**

  Vorbereitung l_{ij}: Länge der Kante $x_i x_j$. Falls es keine Kante $x_i x_j$ gibt, sei $l_{ij} := \infty$ Für jede Ecke $x_i \in X$ des zu untersuchenden Graphen werden drei Variable angelegt:

  1. Entf_i gibt die bisher festgestellte kürzeste Entfernung von x_1 nach x_i an. Der Startwert ist 0 für $i = 1$ und ∞ sonst.
  2. Vorg_i gibt den Vorgänger von x_i auf dem bisher kürzesten Weg von x_1 nach x_i an. Der Startwert ist x_1 für $i = 1$ und undefiniert sonst.
  3. OK_i gibt an, ob die kürzeste Entfernung von x_1 nach x_i schon bekannt ist. Der Startwert für alle Werte von i ist *false*.

  Iteration Wiederhole

  - Suche unter den Ecken x_i mit $OK_i = \mathit{false}$ eine Ecke x_h mit dem kleinsten Wert von Entf_i.
  - Setze $OK_h := \mathit{true}$.
  - Für alle Ecken x_j mit $OK_j = \mathit{false}$, für die die Kante $x_h x_j$ existiert:
    - Falls gilt $\mathit{Entf}_j > \mathit{Entf}_h + l_{hj}$ dann
      - Setze $\mathit{Entf}_j := \mathit{Entf}_h + l_{hj}$
      - Setze $\mathit{Vorg}_j := h$

  solange es noch Ecken x_i mit $OK_i = \mathit{false}$ gibt.

Die nachfolgenden Lösungen verstehen sich als Musterlösungen, d.h. auch andere Lösungen bzw. Lösungswege sind möglich.

Lösung 1 (Grundlagen) *(22P)*

1. **Algorithmus** *(5 Punkte)*

   (a) *(1P) Für die Informatik hat der Algorithmus die Bedeutung einer eindeutigen Vorschrift zur Lösung eines Problems mit Hilfe eines Computers.*

   Zu nennen waren zudem die vier Anforderungen an ein Verfahren, das den Namen „Algorithmus“ verdient: (2P)

   Eindeutigkeit *: Jeder elementare Schritt ist unmißverständlich beschrieben und läßt keine Wahlmöglichkeit offen.*

   Endlichkeit *: Die Beschreibung des Algorithmus hat eine endliche Länge (durch Schleifen o.Ä. kann das ablaufende Programm allerdings durchaus unendlich lange brauchen, wenn der Algorithmus fehlerhaft ist).*

   Terminiertheit *: Ein aktuelles Programm, das diesen Algorithmus implementiert, soll in endlicher Zeit beendet sein.*

   Reproduzierbarkeit *: Der Algorithmus soll unabhängig von einer speziellen Implementation sein. Verschiedene (korrekte) Implementationen sollen dasselbe Ergebnis liefern,*

   (b) *(2P) Folgende Eigenschaften kann man u.U. beobachten:*

   - *Liefert der Algoritmus ein Resultat nach endlich vielen Schritten, so sagt man, der Algorithmus* **terminiert**.
   - *Ein Algorithmus heißt* **determiniert**, *falls er bei gleichen Eingaben und Startbedingungen stets dasselbe Ergebnis liefert.*
   - *Ein Algorithmus heißt* **deterministisch**, *wenn zu jedem Zeitpunkt seiner Ausführung höchstens eine Möglichkeit der Fortsetzung besteht.*
   - *Ein Algorithmus heißt* **nicht deterministisch**, *wenn mindestens zu einem Zeitpunkt seiner Ausführung eine Wahlmöglichkeit bei der Fortsetzung besteht.*

2. **Datenabstraktion** *(9 Punkte)*

(a) *(2P) Mit der Datenabstraktion stehen Methoden zur Verfügung, mit denen die Art der Verwendung eines Datenobjektes von den Details seiner Konstruktion aus elementareren Datenobjekten getrennt werden kann, d.h. dass nicht bekannt ist, wie die verwendete Datenstruktur „intern" realisiert ist. Die Grundidee bei der Datenabstraktion besteht darin, die Programme zur Verwendung von zusammengesetzten Datenobjekten so zu strukturieren, dass sie mit „abstrakten Daten" arbeiten. Gleichzeitig wird eine „konkrete" Darstellung der Daten unabhängig von den die Daten verwendenden Programmen definiert.*

(b) *(3P) Die Schnittstelle zwischen diesen beiden Teilen ist eine Menge von Funktionen:*

- *Für die Datenabstraktion werden* **Konstruktoren** *benötigt, die die gewünschte Datenstruktur erzeugen. Beispiel:* `Empty` *erzeugt einen leere Schlange.*
- *Mittels* **Selektoren** *kann auf die erzeugte Datenstruktur zugegriffen werden. Beispiel:* `front` *liefert das älteste Element einer Schlange.*
- **Bedingungen** *können bestimmte Eigenschaften abprüfen. Beispiel:* `isempty` *prüft, ob eine Schlange leer ist.*
- *Mittels* **Mutatoren** *kann die erzeugte Datenstruktur destruktiv verändert werden. Beispiel:* `dequeue` *entfernt das älteste Element einer Schlange.*

(c) *(2P) Bei der Realisierung einer Schlange (FIFO-Speicher) mittels zwei Stapeln (LIFO-Speicher) wird ein Stapel als Eingabestapel verwendet und einer als Ausgabestapel. Durch „umstapeln" von dem Eingabestapel in den Ausgabestapel, sofern der Ausgabestapel leer ist, wird erreicht, dass das älteste Element des Eingabestapels zum jüngsten Element des Ausgabestapels wird. Das älteste Element des Eingabestapels ist jedoch auch das älteste Element der „überliegenden" Schlange und wird somit als erstes ausgegeben werden. Hier wird ausgenutzt, dass LIFO-Speicher die Reihenfolge ihrer gespeicherten Elemente „umdreht".*

(2P) Um mittels Schlangen die Funktionalität eines Stapels zu realisieren, müsste man auch hier die Reihenfolge der gespeicherten Elemente „umdrehen", damit das älteste Element einer Schlange zum jüngsten Element einer anderen Schlange wird. Da Schlangen, also FIFO-Speicher, aber die Reihenfolge der gespeicherten Elemente in der Ausgabe beibehalten, besteht zunächst keine Möglichkeit diese „umzudrehen". Wenn man nun eine Schlange stets leer hält und dort das neuste Element einträgt, leert man die Andere Schlange und fügt die Elemente in die andere Schlange ein. Damit wird das neuste Element stets

das erste eingefügte Element in dieser Schlange sein und somit nach aussen als letztes eingefügtes Element als erstes wieder ausgelesen werden.

3. **Rekursion** *(8 Punkte)*

(a) (4P) Hier ist ein lauffähiges Java-Programm:

```
import java.io.*;
import java.util.*;
import java.lang.Math;

public class rekursion
{ public static double a(int m) { double ergebnis = 0.0;
      if (m == 0) { ergebnis = 1.0; }
         else { ergebnis = (a(m-1) + b(m-1))/2.0; }
      return ergebnis; }

public static double b(int m) { double ergebnis = 0.0;
      if (m == 0) { ergebnis = java.lang.Math.sqrt(0.5); }
         else { ergebnis = java.lang.Math.sqrt(a(m-1)*b(m-1)); }
      return ergebnis; }

public static double pin(int n) { double ergebnis = 0.0;
      double summe = 0.0;
      for (int k=0;!(k==n);k++) {
         summe = summe +
                 java.lang.Math.pow(2.0,(double) k) *
                 java.lang.Math.pow(a(k) - b(k),2.0); }
      ergebnis = java.lang.Math.pow(a(n)+b(n),2.0)/(1.0 - summe);
      return ergebnis; }

public static void main(String args[]) { int myn = 0;
           if (args.length > 0) myn = Integer.parseInt(args[0],10);
           System.out.println ("Ergebnis ist |"+ pin(myn) +"|"); } }
```

(b) (4P) a_n und b_n werden jeweils 11-mal aufgerufen. Es ergibt sich folgende Struktur für die Summenbildung:

```
a_0-in a_0-out
b_0-in b_0-out
a_1-in a_0-in a_0-out b_0-in b_0-out a_1-out
b_1-in a_0-in a_0-out b_0-in b_0-out b_1-out
```

Es ergibt sich folgende Struktur für die Addition:

```
a_2-in a_1-in a_0-in a_0-out b_0-in b_0-out a_1-out
       b_1-in a_0-in a_0-out b_0-in b_0-out b_1-out
a_2-out
b_2-in a_1-in a_0-in a_0-out b_0-in b_0-out a_1-out
       b_1-in a_0-in a_0-out b_0-in b_0-outťb_1-out
b_2-out
```

Das Berechnungsergebnis ist 3.141592646213543.

Lösung 2 (Komplexitätstheorie) *(14P)*

1. **Begriffsbestimmung** *(2 Punkte)*

   Der beste Fall *(best case): Das ist diejenige Struktur der Eingabedaten, für die der Algorithmus am effektivsten ist.*

   Der schlechteste Fall *(worst case): Diejenige Eingabestruktur, für die der Algorithmus am wenigsten effektiv ist.*

   Der mittlere Fall *(average case): Die zu erwartende Effizienz bei beliebiger Strukturierung der Eingabedaten.*

2. **Break-even Punkt** *(6 Punkte) Die Zeiten werden in Relation zueinander gestellt: Zunächst Insertion-Sort und Bubblesort: (3P)*

$$2*n^2 > n^2+8n+180 \quad \leftrightarrow_{-n^2-8n} \quad n^2-8n > 180 \quad \leftrightarrow \quad (n-4)^2-16 > 180 \quad \leftrightarrow_{+16}$$

$$(n-4)^2 > 196 \quad \leftrightarrow_{\sqrt[2]{};n>0} \quad n-4 > 14 \quad \leftrightarrow_{+4} \quad n > 18$$

Der Break-even-Punkt liegt bei $n = 19$, d.h. ab $n = 19$ ist die Implementierung von Bubblesort schneller als die Implementierung von Insertion Sort.

Nun vergleichen wir Heapsort mit Bubblesort: (3P)

$$192*log_2(n) < n^2+8n+180 \quad \leftrightarrow_{*\frac{1}{192}} \quad log_2(n) < \frac{n^2}{192}+\frac{n}{24}+\frac{15}{16} \quad \leftrightarrow_{x^2} \quad n < 2^{\frac{n^2}{192}+\frac{n}{24}+\frac{15}{16}}$$

Wir berechnen nun für $n = 23$ und $n = 22$ die Werte:

$$23 < 2^{2,755+0,958+0,937} \quad \leftrightarrow \quad 23 < 25,106$$

$$22 < 2^{2,521+0,917+0,937} \quad \leftrightarrow \quad 22 < 20,749$$

Der Break-even-Punkt liegt bei $n = 23$, d.h. ab $n = 23$ ist die Implementierung von Heapsort schneller als die Implementierung von Bubblesort.

3. **Problembehandlung** *(6 Punkte)*

- **Natürliche Einschränkung** *des Problems: Evtl. muss „nur“ ein Spezialfall gelöst werden. Hier hilft ein Nachfragen beim Auftraggeber bzw. eine genauere Analyse des gestellten Problems*

  Im Beispiel: Möchte der Kunde wirklich alle Stellungen des Schachspiels oder evtl. nur des Tic-Tac-Toe Spiels ?

- **Erzwungene Einschränkung** *des Problems: „nur“ gutmütige Probleme oder Spezialfälle lassen sich aufschreiben bzw. werden gelöst. Der Auftraggeber muss darauf hingewiesen werden.*

  Im Beispiel: Es werden ausgehend von nur einer Spielsituation alle Stellungen berechnet, die durch Züge der Bauern möglich sind.

- **Redefinition des Problems***: Betrachtet man das Problem aus einer anderen Sichtweise ist die Komplexitätsanalyse einfacher (quasi Vermeidung des „mit Kanonen auf Spatzen schießen“).*

  Im Beispiel: Könnte man das Schachspiel auch als Tic-Tac-Toe Spiel auffassen ? Oder als Problem des minimalen Gerüstes ?

- **Approximationen***: Reduktion der Ansprüche an die Qualität der Antworten; Aufgabe des Optimalitätsanspruches.*

  Im Beispiel: Es werden nur wenige Stellungen berechnet, um daraus abzuschätzen, was wohl der beste Zug sein könnte. Das Ergebnis kann ein sehr guter bis hin zu einem sehr schlechten Zug sein.

- **Zeitbeschränkte Exploration** *des Suchraums: Größe der Instanzen, die in zumutbarer Zeit lösbar sind.*

  Im Beispiel: Es werden ausgehend von einer Spielsituation nur alle Stellungen berechnet, die in den nächsten vier Zügen möglich sind.

- **Veränderung** *des Algorithmus: Mögliche Redundanzen vermeiden bzw. verkleinern und ggf. andere Operationen für die Problemlösung verwenden.*

  Im Beispiel: Es werden keine Stellungen mehrfach berechnet. Durch z.B. eine Stellungs-DB wird zunächst geprüft, ob die Stellung schon einmal bewertet wurde. Die Stellungen selbst werden in anderer Form dargestellt, z.B. binäres Muster und die Züge durch `shift`*-Operationen realisiert.*

Lösung 3 (Sortieren) *(29P)*

1. **Teile-und-Herrsche** *(3 Punkte)*

   (a) *(1,5P)* Teile: *Der Algorithmus teilt ein Problem in Teilprobleme auf.* Herrsche: *Der Algorithmus löst die Teilprobleme u.U. nach dem gleichen Prinzip, also rekursiv.* Zusammenführen: *Die gelösten Teilprobleme werden zu einer Lösung des Problems zusammengeführt.*

   (b) *(1,5P) Bei der Teilung des Problems ist darauf zu achten, dass eine mögliche Lösung nicht geteilt und damit nicht erkannt wird. Im Praktikum war dies eine Teilsumme, die über die Teilungsgrenzen hinweg geht. In einem solchen Fall entsteht zusätzlicher Aufwand, um dieses Problem zu handhaben.*

2. **Rekursion** *(3 Punkte)*
   (1P) Quicksort führt eine Art Endrekursion durch, d.h. sind alle Rekursionsaufrufe erfolgt (ohne deren Beendung!), steht das Ergebnis fest. Die eigentiche Arbeit wird also vor den Rekursionsaufrufen durchgeführt, nach den Rekursionsaufrufen passiert nichts mehr. (1P) Im Praktikum war dies daran zu merken, dass eine echte iterative Variante nicht implementierbar war: die Grenzen der zu sortierenden Intervalle mussten auf einem Stapel gespeichert werden, um einen korrekten Ablauf zu erzeugen.

   (1P) Bei Mergesort wird die eigentliche Arbeit nach den Rekursionsaufrufen durchgeführt, d.h. wenn alle Rekursiuonsaufrufe erfolgt sind, beginnt durch die Zusammenführung, also das mischen *die eigentliche Arbeit des Sortierens. Das Ergebnis steht erst fest, wenn der erste Aufruf abgeschlossen ist.*

3. **Quicksort** *(8 Punkte)*

*(6P) Das Pivotelement wird durch // markiert. Ist bezüglich einem Pivotelement sortiert worden, wird dieses weggelassen. Wird quicksort mit einem Feld der Länge kleiner oder gleich 1 aufgerufen, wird dies mit * markiert. Ganz rechts werden die Rekursionsaufrufe gezählt.*

9	11	40	22	26	43	36	14	4	/49/	0
9	11	40	22	26	43	36	14	4	/49/	0
9	11	40	22	26	43	36	14	/4/	*	2
/4/	11	40	22	26	43	36	14	9		2
*	11	40	22	26	43	36	14	/9/		4
	/9/	40	22	26	43	36	14	11		4
	*	40	22	26	43	36	14	/11/		6
		/11/	22	26	43	36	14	40		6
		*	22	26	43	36	14	/40/		6
			22	26	14	36	43	/40/		6
			22	26	14	36	/40/	43		6
			22	26	14	/36/		*		8
			22	26	14	/36/				8
			22	26	/14/	*				10
			/14/	26	22					10
			*	26	/22/					12
				/22/	26					12
				*	*					14
4	9	11	14	22	26	36	40	43	49	

(2P) Die Anzahl der Rekursionsaufrufe ist 14 (siehe Zählung am rechten Rand in der Tabelle) und die Rekursionstiefe 8 (Anzahl der Rekursionsaufrufe im längsten Zweig), auch wenn in der letzten Stufe eigentlich nichts mehr passiert!

4. **Heapsort** *(15 Punkte)*
Gegeben sei ein Maximum-Heap, wie in der Vorlesung vorgestellt.

(a) (2P) Die „Heap-Eigenschaft“ ist wie folgt durch die Definition eines Heap beschrieben: Eine Folge $F = k_1, k_2, \ldots, k_N$ *von Schlüsseln nennen wir einen* Heap *wenn* $k_i \leq k_{\lfloor \frac{i}{2} \rfloor}$ *für* $2 \leq i \leq N$ *gilt. Gleichbedeutend damit ist* $k_i \geq k_{2i}$ *und* $k_i \geq k_{2i+1}$*, falls* $2i, 2i+1 \leq N$*.*

(b) (1P) Die Baumrepräsentation eines Heaps wird wie folgt in ein Array kodiert: Die Nachfolger von Knoten i *stehen an den Positionen* $2i$ *(linker Nachfolger) und Position* $2i+1$ *(rechter Nachfolger). Entsprechend ist der Vorgänger von Knoten* i *an der Position* $\lfloor \frac{i}{2} \rfloor$ *zu finden. Dies gilt, insofern Nachfolger oder Vorgänger tatsächlich vorhanen sind.*

(c) (6P) Ein Heap ist als Array kontinuierlich aufgebaut, d.h. es gibt keine Lücken im Array (1P). Beim Löschen wird auch stets dass Element mit dem höchsten Index an die erste Position kopiert,um die kompakte Speicherung zu bewahren (1P). Entsprechend ist der zugehörige Baum kompakt aufgebaut: In einem Heap der Höhe h *ist also in der untersten Ebene zumindest ein Knoten vorhanden. Maximal ist in einem Heap der Höhe* h *die unterste Ebene komplett aufgefüllt. (2P) Unter der Verwendung, dass ein vollständiger binärer Baum der Höhe* $h-1$ 2^{h-1} *Knoten beinhaltet, ergibt sich zunächst, dass die Ebene* h 2^{h-1} *Knoten maximal beinhalten kann (1P), womit also ein vollständiger binärer Baum der Höhe* h $2^{h-1} + 2^{h-1} = 2 * 2^{h-1} = 2^{1+h-1} = 2^h$ *Knoten beinhaltet.*

Somit ergibt sich für einen Heap der Höhe h *(1P):*

minimal *:* $2^{h-1} + 1$ *Knoten.*
maximal *:* $2^{h-1} + 2^{h-1} = 2^h$ *Knoten.*

(d) (2P) Das kleinste Element kann sich in einem Heap nur auf der untersten Ebene befinden, d.h. ein Blatt in der Baumrepräsentation sein. Sonst hätte es Nachfolger, die gemäss der „Heap-Eigenschaft“ kleiner sein müssten, als es selbst; was ein Widerspruch zu der Voraussetzung ist, dass es sich hier schon um das kleinste Element handelt.

(e) *(4P) Die Blätter in einer unsortierenden Zahlenfolge bilden zu Anfang bereits einen Heap, da sie keine Nachfolger haben und somit die „Heap-Eigenschaft“ nicht verletzen können. (2P)*

Die Struktur wird von hinten (rechts) nach vorne (links) in der Array-Repräsentation eines Heaps bzw. von unten nach oben in der Baumrepräsentation eines Heaps als Heap aufgebaut, da die grösseren Elemente so gesichert nach links bzw. nach oben wandern können. Denn man kennt in dieser Bottom-Up Vorgehensweise alle bisher im Heap enthaltenen Elemente, was notwendig ist, um zu entscheiden, welches das größte Element ist. Wenn man von links nach rechts wandern würde, kennt man noch nicht das wirklich größte Element. (2P)

Lösung 4 (Bäume und Graphen) *(20P)*

1. **Greedy-Algorithmen** *(7 Punkte)*
   Hier die Definitionen für das zu lösende Problem:

   (a) *(0,5P)Menge von Kandidaten* $C := \{4kg, 9kg, 11kg, 14kg, 22kg, 26kg\}$.

   (b) *(0,5P)Lösungsmenge* $S \subseteq C$*, am Anfang leer, der Rucksack.*

   (c) *(1P)*
   $$solution(S) := \begin{cases} true & falls\ \sum_{x \in S} x \leq 40kg \\ false & sonst \end{cases}$$

   (d) *(1P)Testfunktion* $feasible$*:*
   $$feasible(R) := \begin{cases} true & falls\ \sum_{x \in S} x \leq 40kg \\ false & sonst \end{cases}$$

   (e) *(1P)select:* $select(C) := x$ *falls* $(\nexists y \in C : y < x)$*, d.h.* x *ist minimal*

   (f) *(1P)* val*:* $val(S) := \sum_{x \in S} x$*, das Gewicht des Rucksack's*

   (1,5P) Hier der Ablauf des Algorithmus:

R	C
$\emptyset$	$\{4kg, 9kg, 11kg, 14kg, 22kg, 26kg\}$
$\{4kg\}$	$\{9kg, 11kg, 14kg, 22kg, 26kg\}$
$\{4kg, 9kg\}$	$\{11kg, 14kg, 22kg, 26kg\}$
$\{4kg, 9kg, 11kg\}$	$\{14kg, 22kg, 26kg\}$
$\{4kg, 9kg, 11kg, 14kg\}$	$\{22kg, 26kg\}$
$\{4kg, 9kg, 11kg, 14kg\}$	$\{26kg\}$
$\{4kg, 9kg, 11kg, 14kg\}$	$\emptyset$

   (0,5P) Von Hand würde ich den Rucksack wie folgt bepacken: $R := \{14kg, 26kg\}$.

2. **AVL-Bäume** *(9 Punkte)*
Zunächst konstruieren wir die Situation **vor dem Einfügen** *(2P): Es sei A der Knoten, an dem die Debalance erkannt wird. Der Knoten habe vor dem Einfügen die Höhe $n+1$. Damit hat mindestens einer seiner Nachfolger die Höhe n. Damit nun der Fehler überhaupt erstmalig an diesem Knoten auftreten kann, muss der andere Nachfolger die Höhe $n-1$ haben. Wir legen o.B.d.A. fest: der linke Nachfolger B habe die Höhe n und der rechte Nachfolger $R1$ habe die Höhe $n-1$. Für die Nachfolger von Knoten B gilt, dass sie beide (L und $R2$) die Höhe $n-1$ haben müssen, da sonst hier schon die Debalance auftreten würde.*

Wir nehmen nun eine **Fallunterscheidung** *vor, um die einfache oder doppelte Rotation zu untersuchen und damit die Situation* **nach dem Einfügen** *herzuleiten:*

(a) einfache Rechtsrotation*: (3P) Nach dem Einfügen würden nun also folgende neue Höhen entstehen: Knoten L erhält die Höhe n, und damit Knoten B die Höhe $n+1$. Es wird nun eine Rechtsrotation um A durchgeführt. $R2$ und $R1$ haben beide die Höhe $n-1$, womit der Knoten A die Höhe n erhält, die selbe Höhe, wie Knoten L, womit Knoten B die Höhe $n+1$ erhält, was genau der Höhe an dieser Position vor dem Einfügen entspricht.*

(b) doppelte Rechtsrotation*: (4P) Wir müssen hier noch die Situation vor dem Einfügen zunächst ergänzen: Die Nachfolger ($R2l$, $R2r$) von Knoten $R2$ haben ebenfalls beide die Höhe $n-2$, da sonst die Debalance hier auftreten würde.*

Nach dem Einfügen würden nun also folgende neue Höhen entstehen: Knoten $R2$ erhält die Höhe n, und damit Knoten B die Höhe $n+1$. Wir nehmen o.B.d.A. an, dass nach dem Einfügen der Knoten $R2l$ die Höhe $n-1$ erhält und damit Knoten $R2$, wie gesagt, die Höhe n.

Es wird nun eine Linksrotation um Knoten B durchgeführt und anschliessend eine Rechtsrotation um Knoten A. Im Ergebnis: L und $R2l$ haben beide die Höhe $n-1$, womit B die Höhe n erhält. $R1$ hat die Höhe $n-1$ und $R2r$ die Höhe $n-2$, womit A die Höhe n erhält. Damit erhält $R2$ die Höhe $n+1$, was genau der Höhe an dieser Position vor dem Einfügen entspricht.

Die Höhe an der Position von Knoten A bleibt nach der Rotation erhalten, wenn auch der Knoten A nun an einer neuen Position steht!

3. **optimale Wege** *(4 Punkte)*
Im folgenden beschreibt die erste Zeile den Zustand nach der Initialisierung. Dann wird der Rand eingetragen. Sofern Ergebnisse gesichert sind, wird die Spalte leer gelassen. Dargestellt ist dann: $Entf_i, Vorg_i, OK_i$*. Hier nun der Ablauf:*

x_6	x_5	x_4	x_3	x_2	x_1
$\infty, \bot, F$	$\infty, \bot, F$	$\infty, \bot, F$	$\infty, \bot, F$	$\infty, \bot, F$	$0, \top, T$
$\infty, \bot, F$	$7, x_1, F$	$\infty, \bot, F$	$2, x_1, F$	$1, x_1, F$	$0, \top, T$
$9, x_2, F$	$7, x_1, F$	$\infty, \bot, F$	$2, x_1, F$	$1, x_1, T$	$0, \top, T$
$9, x_2, F$	$7, x_1, F$	$4, x_3, F$	$2, x_1, T$	$1, x_1, T$	$0, \top, T$
$9, x_2, F$	$6, x_4, F$	$4, x_3, T$	$2, x_1, T$	$1, x_1, T$	$0, \top, T$
$8, x_5, F$	$6, x_4, T$	$4, x_3, T$	$2, x_1, T$	$1, x_1, T$	$0, \top, T$
$8, x_5, T$	$6, x_4, T$	$4, x_3, T$	$2, x_1, T$	$1, x_1, T$	$0, \top, T$

Der optimale Weg z.B. von x_1 *nach* x_6 *ist:* $x_1x_3x_4x_5x_6$*.*

Lösung 5 (Hashing) *(14P)*

1. **Konstruktionsprinzip** *(6 Punkte)*

   (a) *Suchalgorithmen, die mit Hashing arbeiten, bestehen aus folgenden zwei Teilen:*

      i. *(1P) Im ersten Schritt transformiert der Algorithmus den Suchschlüssel mithilfe einer Hashfunktion in eine Tabellen-/Speicheradresse. Diese Funktion bildet im Idealfall unterschiedliche Schlüssel auf unterschiedliche Adressen ab. Oftmals können aber auch zwei oder mehrere unterschiedliche Schlüssel zur gleichen Tabellenadresse führen.*

      ii. *(1P) Somit führt eine Hashing-Suche im zweiten Schritt eine Kollisionsbeseitigung durch, die sich mit derartigen Schlüsseln befasst.*

   (b) *Das Ziel der einzelnen Teile ist:*

      i. *(1P) Die Hashfunktion versucht eine eventuell vorhandene Häufung der Schlüssel,d.h. z.B. eine aufsteigende Nummerierung, aufzuheben, also das Ziel ist eine gleichmässige Verteilung der Schlüssel auf alle Tabellen-/Speicherplätze zu erreichen, ohne von der Verteilung der Schlüssel selbst beeinflusst zu werden.*

      ii. *(1P) Für die Kollisionsbeseitigung gibt es zwei Möglichkeiten: Zum Einen kann auf ein anderes Verfahren zurückgegriffen werden, z.B. lineare Listen oder ausgeglichene Bäume. Zum Anderen kann rekursiv gearbeitet werden: Hierbei ist das Ziel, durch die Kollisionsbeseitigung möglichst wenige Kollisionen für die nachfolgenden Einfügevorgänge zu erzeugen.*

   (c) *(2P) Hashing ist ein gutes Beispiel für einen Kompromiss zwischen Zeit- und Platzbedarf. Wenn es keine Speicherbeschränkung gäbe, könnte jede Suche mit nur einem einzigen Speicherzugriff realisiert werden, indem einfach der Schlüssel als Speicheradresse analog zu einer schlüsselindizierten Suche verwendt wird. Allerdings läßt sich dieser Idealzustand meistens nicht verwirklichen, weil der Speicherbedarf besonders bei langen Schlüsseln viel zu groß ist. Gäbe es andererseits keine Zeitbeschränkung, könnten wir mit einem Minimum an Speicher auskommen, indem ein sequenzielles Suchverfahren eingesetzt wird. Hashing erlaubt es, zwischen diesen beiden Extremen ein vertretbares Maß sowohl für die Zeit als auch den Speicherplatz zu finden. Insbesondere können wir jedes beliebige Verhältnis einstellen, indem wir lediglich die Größe der Hashtabelle anpassen; dazu brauchen wir weder Code neu zu schreiben noch auf andere Algorithmen auszuweichen.*

2. **Hashing** *(3 Punkte)*
Die Listen werden in nachfolgender Tabelle durch Klammern dargestellt.

-	-	-	-	*9*	$9 mod 5 = 4$
-	*11*	-	-	*9*	$11 mod 5 = 1$
40	*11*	-	-	*9*	$40 mod 5 = 0$
40	*11*	*22*	-	*9*	$22 mod 5 = 2$
40	*(11,26)*	*22*	-	*9*	$26 mod 5 = 1$
40	*(11,26)*	*22*	*43*	*9*	$43 mod 5 = 3$
40	*(11,26,36)*	*22*	*43*	*9*	$36 mod 5 = 1$
40	*(11,26,36)*	*22*	*43*	*(9,14)*	$14 mod 5 = 4$
(40,5)	*(11,26,36)*	*22*	*43*	*(9,14)*	$5 mod 5 = 0$

3. **Quadratisches Sondieren** *(5 Punkte)*
Die Kollisionsbehandlung wird in eigenen Zeilen beschrieben.

9	-	-	-	-	-	-	-	-	$9 mod 9 = 0$
9	-	*11*	-	-	-	-	-	-	$11 mod 9 = 2$
9	-	*11*	-	*40*	-	-	-	-	$40 mod 9 = 4$
9	-	*11*	-	*40*	-	-	-	-	$22 mod 9 = 4$
9	-	*11*	-	*40*	*22*	-	-	-	$(22 + 1^2) mod 9 = 5$
9	-	*11*	-	*40*	*22*	-	-	*26*	$26 mod 9 = 8$
9	-	*11*	-	*40*	*22*	-	*43*	*26*	$43 mod 9 = 7$
9	-	*11*	-	*40*	*22*	-	*43*	*26*	$36 mod 9 = 0$
9	*36*	*11*	-	*40*	*22*	-	*43*	*26*	$(36 + 1^2) mod 9 = 1$
9	*36*	*11*	-	*40*	*22*	-	*43*	*26*	$14 mod 9 = 5$
9	*36*	*11*	-	*40*	*22*	*14*	*43*	*26*	$(14 + 1^2) mod 9 = 6$
9	*36*	*11*	-	*40*	*22*	*14*	*43*	*26*	$5 mod 9 = 5$
9	*36*	*11*	-	*40*	*22*	*14*	*43*	*26*	$(5 + 1^2) mod 9 = 6$
9	*36*	*11*	-	*40*	*22*	*14*	*43*	*26*	$(5 + 2^2) mod 9 = 0$
9	*36*	*11*	-	*40*	*22*	*14*	*43*	*26*	$(5 + 3^2) mod 9 = 5$
9	*36*	*11*	*5*	*40*	*22*	*14*	*43*	*26*	$(5 + 4^2) mod 9 = 3$