Prof. Dr. Thomas Breuel
Dipl.-Inf. Patrick Michel (AG ST)
Dipl.-Ing. Ludwig Schmidt-Hackenberg

TU Kaiserslautern
Fachbereich Informatik
AG Bildverstehen und Mustererkennung

Lösungshinweise/-vorschläge zum Übungsblatt 9: Software-Entwicklung 1 (WS 2010/11)

Die Hinweise und Vorschläge in diesem Dokument sollen der Lösungsfindung dienen und erheben demnach weder Anspruch auf Vollständigkeit noch Korrektheit. Sollten Sie Fehler finden, würden wir uns freuen, wenn Sie uns diese mitteilen. (Kontaktdaten finden Sie auf unserer Webpräsenz.)

Die auf diesem Blatt zur Lösung verwendeten Sprachmittel beschränken sich auf prozedurales Java mit Verbunden und Feldern. Es wurde außerdem weitestgehend auf verkürzende Java Notationen verzichtet, welche die Lesbarkeit des Codes für geübte Programmierer noch verbessern können.

Aufgabe 1 Felder und Matrizenmultiplikation (Präsenzaufgabe)

Implementieren Sie die unten angegebenen Prozeduren zur Arbeit mit Matrizen.

```
int[][] create(int rows, int cols) {
  if (!(rows > 0 && cols > 0)) {
    return null;
  }

  return new int[rows][cols];
}

void set(int[][] m, int i, int j, int value) {
  m[i][j] = value;
}

int get(int[][] m, int i, int j) {
  return m[i][j];
}

int rows(int[][] m) {
  return m.length;
}

int cols(int[][] m) {
  return m[0].length;
}

void output(int[][] m) {
  for (int i = 0; i < rows(m); i++) {
    for (int j = 0; j < cols(m); j++) {
      IO.print(get(m, i, j));
      IO.print(" ");
    }
    IO.println("");
  }
}
```

Aufgabe 2 Listen (Einreichaufgabe)

Implementieren Sie einen Verbunddatentyp für die Verwaltung von Listen von **double**-Werten in Java.

```
class List { double[] contents; }

List create() {
  List l = new List();
  l.contents = new double[0];
  return l;
}

void append(List l, double elem) {
  double[] c = new double[l.contents.length + 1];
  for (int i = 0; i < l.contents.length; i++) {
    c[i] = l.contents[i];
  }
  c[c.length - 1] = elem;
  l.contents = c;
}

int size(List l) { return l.contents.length; }

double get(List l, int index) { return l.contents[index]; }

void put(List l, int index, double elem) {
  double[] c;
  if (index < 0 || index > l.contents.length) return;

  c = new double[l.contents.length + 1];
  for (int i = 0; i < c.length; i++) {
    if (i < index) {
      c[i] = l.contents[i];
    } else if (i == index) {
      c[i] = elem;
    } else {
      c[i] = l.contents[i - 1];
    }
  }
  l.contents = c;
}

void remove(List l, int index) {
  double[] c;
  if (index < 0 || index >= l.contents.length) return;

  c = new double[l.contents.length - 1];
  for (int i = 0; i < c.length; i++) {
    if (i < index) {
      c[i] = l.contents[i];
    } else {
      c[i] = l.contents[i + 1];
  } }
  l.contents = c;
}

boolean contains(List l, double elem) {
  for (int i = 0; i < l.contents.length; i++ ) {
    if (l.contents[i] == elem) return true;
  }
  return false;
}
```

Da zu dem Zeitpunkt dieser Übung Fehlerbehandlung in Java noch nicht Teil der Vorlesung war, wird mit falschen Parametern nicht besonders gut verfahren und einfach aus der Prozedur gesprungen.

Aufgabe 3 Felder und Matrizenmultiplikation (Einreichaufgabe)

a) Realisieren Sie die Matrizenmultiplikation `int[][] mult(int[][] a, int[][] b)` als Prozedur. Wenn die übergebenen Matrizen nicht zum Multiplizieren geeignet sind, geben Sie `null` zurück.

```
int[][] mult(int[][] a, int[][] b) {
  int[][] res;

  if (cols(a) != rows(b)) {
    return null;
  }

  res = create(rows(a), cols(b));
  for (int i = 0; i < rows(a); i++) {
    for (int j = 0; j < cols(b); j++) {
      int sum = 0;
      // cols(a) == rows(b)
      for (int k = 0; k < cols(a); k++) {
        sum = sum + get(a, i, k) * get(b, k, j);
      }
      set(res, i, j, sum);
    }
  }
  return res;
}
```

b) Implementieren Sie die folgenden Prozeduren analog zu denen von Aufgabe 1 und Aufgabe 3 a).

Die Lösungen von `output` und `mult` ändern sich – außer in der Signatur – überhaupt nicht, da keine der beiden direkt auf die interne Repräsentation zugreift.

Die Lösung wurde platzsparend notiert und drückt deshalb nicht notwendiger Weise einen guten Stil zur Presäntation von Code aus. Dies bezieht sich natürlich auch auf die beiden Funktionen aus Aufgabe 2 und die folgenden Aufgaben.

```
Matrix create(int rows, int cols) {
  Matrix res;

  if (!(rows > 0 && cols > 0)) {
    return null;
  }

  res = new Matrix();
  res.rows = rows;
  res.cols = cols;

  res.data = new int[rows*cols];
  return res;
}

int flatten(Matrix m, int i, int j)          { return i * m.cols + j; }
void set(Matrix m, int i, int j, int value) { m.data[flatten(m, i, j)] = value; }
int  get(Matrix m, int i, int j)             { return m.data[flatten(m, i, j)]; }
int rows(Matrix m)                           { return m.rows; }
int cols(Matrix m)                           { return m.cols; }
```

c) In der praktischen Anwendung von Matrizen treten sogenannte *Bandmatrizen* auf. Implementieren Sie die folgenden Prozeduren:

Auch hier können die Prozeduren `output`, `rows`, `cols` und selbst die `mult` fast unverändert übernommen werden. Außer den Signaturen ändern sich nur die Aufrufe von `get` und `set`, die wenn passend noch den Suffix `B` bekommen.

```
int getRowB(BandMatrix m, int i, int j) {
  return i - j + m.bandwidth;
}
```

```
BandMatrix createB(int rows, int cols, int width) {
  BandMatrix res;

  if (!(rows > 0 && cols > 0)) {
    return null;
  }

  res = new BandMatrix();
  res.rows = rows;
  res.cols = cols;
  res.bandwidth = width;

  res.data = new int[1 + width * 2][cols];
  return res;
}

void setB(BandMatrix m, int i, int j, int value) {
  m.data[getRowB(m, i, j)][j] = value;
}

int getB(BandMatrix m, int i, int j) {
  return Math.abs(i - j) > m.bandwidth ? 0 : m.data[getRowB(m, i, j)][j];
}
```

d) Schreiben Sie ein Programm unter Verwendung der Lösung von Teil a) oder b), das `y` nach folgender Gleichung für ein eingegebenes `n` berechnet. Was berechnet dieses Programm?

```
void main(String[] args) {
  Matrix m = create(2, 2), v = create(2, 1), m2;
  int n;

  set(m, 0, 0, 1);
  set(m, 0, 1, 1);
  set(m, 1, 0, 1);
  set(m, 1, 1, 0);

  set(v, 0, 0, 1);
  set(v, 1, 0, 0);

  IO.print("Geben Sie ein n ein: ");
  n = IO.readInt();
  m2 = potenz(m, n);

  m2 = mult(m2, v);

  IO.println("Ergebnis: " + get(m2, 0, 0));
}
```

```
Matrix potenz(Matrix m, int n) {
  Matrix result = create(2, 2);

  // mit Einheitsmatrix initialisieren
  set(result, 0, 0, 1);
  set(result, 0, 1, 0);
  set(result, 1, 0, 0);
  set(result, 1, 1, 1);

  for (int i = 0; i < n; i++) result = mult(result, m);

  return result;
}
```

Das Programm berechnet die Fibonaccifunktion.

Aufgabe 4 Rekursion und Iteration (Einreichaufgabe)

Wir betrachten nun Ihre Listenimplementierung aus Aufgabe 2 als abstrakten Datentyp, das heißt wir wissen nun nichts mehr über die Struktur des Verbundtyps `List` und können nur über die angegebenen Prozeduren mit Listen interagieren.

a) Schreiben Sie zwei zusätzliche Prozeduren `isSorted` und `insert`, mit denen man testen kann, ob eine Liste sortiert ist, und ein Element an entsprechender Stelle sortiert einfügen kann:

```
boolean isSorted(List l)               // Testet, ob die Liste sortiert ist.
void      insert(List l, double elem)  // Fuegt 'elem' sortiert ein.
```

```
boolean isSorted(List l) {
  for(int i = 0; i < size(l) - 1; i++) {
    if (get(l, i) > get(l, i + 1)) {
      return false;
    }
  }
  return true;
}

void insert(List l, double elem) {
  int index = 0;
  for (; index < size(l) && get(l, index) < elem; index++) { }
  put(l, index, elem);
}
```

b) Schreiben Sie zwei neue Varianten `containsR` und `containsI` der Prozedur `contains`, die mittels binärer Suche bestimmen, ob eine Zahl in der Liste enthalten ist. Die eine Variante soll dabei *rekursiv* programmiert werden, die andere rein *iterativ*.

```
boolean containsR(List l, double elem) {
  return bcRec(l, elem, 0, size(l));
}

boolean bcRec(List l, double elem, int left, int right) {
  if (left < right) {
    int mid = left + (right - left) / 2;
    if (elem < get(l, mid)) { return bcRec(l, elem, left, mid);      } else
    if (elem > get(l, mid)) { return bcRec(l, elem, mid + 1, right); } else
                            { return true; }
  } else return false;
}

boolean containsI(List l, double elem) {
  int left = 0;
  int right = size(l);
  while (left < right) {
    int mid = left + (right - left) / 2;
    if (elem < get(l, mid)) {
      right = mid;
    } else if (elem > get(l, mid)) {
      left = mid + 1;
    } else {
      return true;
    }
  }
  return false;
}
```

c) Schreiben Sie für alle geforderten Funktionen Tests, mit denen Sie sowohl Standard- als auch Randfälle abprüfen können.

Keine Hinweise.

Aufgabe 5 Aufwandsabschätzung (Einreichaufgabe)

a) Geben Sie die rekursive (mathematische) Funktion $C(n)$ an, welche die für die Berechnung der n-ten Fibonacci-Zahl notwendige Anzahl an Funktionsaufrufen `fib` ermittelt.

$$C(n) = \begin{cases} 1 & n \leq 1 \\ 1 + C(n-1) + C(n-2) & n > 1 \end{cases}$$

b) Entfernen Sie die in der im vorherigen Aufgabenteil aufgestellten Funktion C auftretende Rekursion, indem Sie C mit Hilfe der Fibonacci Funktion `fib` selbst ausdrücken. Wie könnte man die Gültigkeit der von Ihnen aufgestellten Funktion beweisen?

Wichtig: Im Beweis nennen wir die von der Java-Funktion `fib` beschriebene mathematische Funktion F.

Die Darstellung von C mit Hilfe von F ist $C(n) = 2 \cdot F(n+1) - 1$. Die Gültigkeit der aufgestellten Formel lässt sich mit Hilfe der *vollständigen Induktion* beweisen.

Zu zeigen ist $C(n) = 2 \cdot F(n+1) - 1$. Bekannt aus Aufgabenteil a) ist die Tatsache, dass $C(n) = 1$ für $n \leq 1$ und $C(n) = 1 + C(n-1) + C(n-2)$ für $n > 1$ gilt. Zusätzlich folgt aus der Definition der Fibonacci-Zahlen $F(n) = F(n-1) + F(n-2)$ für $n > 1$.

Für den Fall $n \leq 1$ können wir die Gleichheit direkt beweisen:

$$\begin{aligned} C(0) &\overset{\text{Def C}}{=} 1 &&= 2 \cdot 1 - 1 \overset{\text{Def F}}{=} 2 \cdot F(1) - 1 \\ C(1) &\overset{\text{Def C}}{=} 1 &&= 2 \cdot 1 - 1 \overset{\text{Def F}}{=} 2 \cdot F(2) - 1 \end{aligned}$$

Für die Induktion brauchen wir die Aussage sowohl für n, also auch für $n-1$. Damit müssen wir im Induktionsanfang bei $n = 2$ zwei Gleichungen zeigen, können in der Voraussetzung und konsequent im Schritt aber auch eine zusätzliche Gleichheit verwenden.

I.A. Den Anfang der Induktion setzen wir dann bei 2, sprich dem ersten $n > 1$:

$$\begin{aligned} C(2) &\overset{\text{Def C}}{=} 1 + 1 + 1 &&= 2 \cdot 2 - 1 \overset{\text{Def F}}{=} 2 \cdot F(3) - 1 \\ C(1) &\overset{\text{Def C}}{=} 1 &&= 2 \cdot 1 - 1 \overset{\text{Def F}}{=} 2 \cdot F(2) - 1 \end{aligned}$$

I.V. Für $n \in \mathbb{N}$ gilt:

$$\begin{aligned} C(n) &= 2 \cdot F(n+1) - 1 \\ C(n-1) &= 2 \cdot F(n) - 1 \end{aligned}$$

I.S. $n \to n+1$:

$$\begin{aligned} C(n+1) &\overset{\text{Def C}}{=} 1 + C(n) + C(n-1) \\ &\overset{\text{IV}}{=} 1 + 2 \cdot F(n+1) - 1 + 2 \cdot F(n) - 1 \\ &= 2 \cdot (F(n+1) + F(n)) - 1 \\ &\overset{\text{Def F}}{=} 2 \cdot F(n+2) - 1 \end{aligned}$$

c) Werten Sie den Ausdruck `fib(5)` unter Verwendung einer Baumdarstellung aus, in der die Knoten die Ergebnisse der jeweiligen (Teil-)Auswertung enthalten. Was lässt sich aus dieser Darstellung im Bezug auf die Effizienz der rekursiven Implementierung ableiten?

In der Baumdarstellung sollte auffallen, dass einige Funktionsaufrufe, beispielsweise `fib 2`, mehrfach auftauchen und somit auch (unnötigerweise) mehrfach berechnet werden. Aufgrund dieser redundanten Berechnungen ist die rekursive Implementierung nicht sehr effizient.

d) Statt auf rekursive Weise lassen sich die Fibonacci-Zahlen mit der *Formel von Moivre-Binet* berechnen:

$$F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right)$$

Verwenden Sie die Formel, um eine obere Schrankenfunktion für C zu ermitteln. Als obere Schrankenfunktion wird in dieser Teilaufgabe eine Funktion f bezeichnet, die für alle $n \in \mathbb{N}$ die Bedingung $C(n) \leq f(n)$ erfüllt. Diskutieren Sie unter Zuhilfenahme der ermittelten Schrankenfunktion die Effizienz der Implementierung.

Hinweis: Ersetzen Sie ausgehend von der in Teilaufgabe b) ermittelten Darstellung für C und der Formel von Moivre-Binet komplexere mathematische Ausdrücke durch einfachere Ausdrücke.

$$\begin{aligned}
C(n) &= 2 \cdot F(n+1) - 1 \\
&= \frac{2}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right) - 1 \\
&< \frac{2}{2} \cdot \left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right) - 1 \\
&< \left(\left(\frac{1+3}{2}\right)^{n+1} + \left|\frac{1-3}{2}\right|^{n+1}\right) - 1 \\
&= \left(\left(\frac{4}{2}\right)^{n+1} + \left|\frac{-2}{2}\right|^{n+1}\right) - 1 \\
&= \left(2^{n+1} + |-1|^{n+1}\right) - 1 \\
&= 2^{n+1}
\end{aligned}$$

Also ist $f(n) = 2^{n+1}$ eine obere Schrankenfunktion für $C(n)$. Damit fällt der Berechnungsalgorithmus in die Komplexitätsklasse der exponentiellen Funktionen.

Aufgabe 6 Verbunde, Bindungen, Effizienz

Kreuzen Sie an, ob folgende Aussagen wahr oder falsch sind. Bereiten Sie diese Aufgabe bis zu Ihrer nächsten Übungsstunde vor, so dass Sie bei Unklarheiten nachfragen und die Antworten diskutieren können.

wahr	falsch	
☒	☐	Mit einem Verbund können Variablen unterschiedlichen Typs zusammengefasst werden.
☐	☒	Die Lebensdauer einer Variablen, die eine Referenz auf einen Verbund speichert, ist an die Lebensdauer des Verbundes gekoppelt. *Genauso wie Felder sind Verbunde in Java Objekte und alle Objekte leben von ihrer Erzeugung bis zum Programmende. Die Lebensdauer von Variablen die Referenzen auf Verbunde speichern, wird wie bei allen anderen Variablen auch bestimmt. Es kann daher passieren, dass alle Variablen mit Referenzen auf einen Verbund, nicht mehr lebendig sind, der Verbund ist dann nicht mehr zugreifbar.*
☐	☒	Folgender Anweisungsblock `{Paar p; p.ersteKomp = 5;}` (`Paar` ist ein Verbundtyp) wird fehlerfrei übersetzt. *Die Variable p ist nur als Referenz auf einen Verbund deklariert, ihr wird jedoch nie eine Referenz zugewiesen, daher kann über sie auch nicht auf einen Verbund oder seine Komponenten zugegriffen werden. In einem so einfachen Fall merkt der Javacompiler dies und meldet sich mit der Meldung "variable p might not have been initialized".*
☐	☒	Ein Geflecht besteht aus Verbunden und Feldern, die sich nicht gegenseitig referenzieren. *Die interessante Eigenschaft eines Geflechts sind ja gerade die gegenseitigen Referenzen.*

☒	☐	Eine Verbundkomponente kann eine Referenz auf den Verbund enthalten zu dem sie gehört. *Zyklische Geflechte sind erlaubt, insbesondere auch Geflechte mit Zyklen der Länge eins, was dann bedeutet, dass eine Komponente auf den Verbund zeigt, in dem sie sich befindet.*
☐	☒	Eine Variable kann überall genutzt werden, egal wo sie deklariert ist. *Variablen können nur innerhalb ihres Deklarations-, Gültigkeits- und Sichtbarkeitsbereiches benutzt.*
☒	☐	Ein Anweisungsblock definiert einen Deklarationsbereich.
☐	☒	Es gibt Java-Programme mit nur einem Deklarationsbereich. *Es gibt mindestens den Deklarationsbereich der das gesamte Programm umfasst und einen der den Rumpf der Main-Prozedur umfasst.*
☒	☐	Globale Variablen haben den größtmöglichen Gültigkeitsbereich.
☐	☒	Variablen aus dem gleichen Deklarationsbereich haben den gleichen Gültigkeitsbereich. *Es ist möglich, dass sie verschiedene Gültigkeitsbereiche haben, da der Gültigkeitsbereich lokaler Variablen erst an der Deklarationsstelle beginnt.*
☐	☒	Auf eine gültige Variable kann in ihrem Deklarationsbereich immer zugegriffen werden. *Die Bindung kann verschattet sein, so dass mit dem gleichen Namen eine andere Variable angesprochen wird.*
☐	☒	Deklarations-, Gültigkeits- und Sichtbarkeitsbereiche ändern sich bei der Programmausführung. *Diese drei Aspekte betreffen die Struktur des Quelltextes und sind statische Aspekte der Programmiersprache.*
☒	☐	Während der Programmausführung sind alle Variablen in ihrem Sichtbarkeitsbereich lebendig.
☒	☐	Während der Programmausführung sind alle Variablen in ihrem Gültigkeitsbereich lebendig.
☐	☒	Die Effizienz eines Algorithmus wird üblicherweise in Abhängigkeit vom Wert der Eingabe bestimmt. *Üblicherweise wird die Effizienz in Abhängigkeit von der* Größe *der Eingabe bestimmt.*
☐	☒	Bei Komplexitätsanalysen erhält man präzise Aussagen über Speicher- und Zeitbedarf eines Programms. *Es werden nur obere und untere Schranken ermittelt. Die präzisen Aussagen erhält man nicht, da alle konstanten Faktoren und der Einfluss der Eingabewerte vernachlässigt werden.*
☒	☐	Die Aufwandsfunktion eines Programms beschreibt die Zahl der Operationen für den ungünstigsten Fall in Abhängigkeit von der Größe der Eingabe.
☒	☐	Für eine Aufwandsfunktion A gilt: $A \in O(A)$
☐	☒	Rechner sind heute so schnell, dass es keine Rolle spielt, ob ein Algorithmus in $O(n^2)$ oder $O(n \log n)$ liegt. *Schon für kleine n macht dies einen großen Unterschied. Selbst* $O(n \log n)$ *verglichen mit* $O(n)$ *macht je nach Größe von n einen großen praktischen Unterschied.*
☒	☐	Verschiedene Implementierungen von Algorithmen der gleichen Zeit-Komplexitätsklasse sind nicht unbedingt gleich schnell. *Die Komplexitätsklasse liefert nur eine grobe Einteilung. In der Praxis kommen alle vernachlässigten Aspekte noch hinzu und die Konstanten können entscheidend sein.*
☐	☒	Ein Algorithmus der Zeit-Komplexitätsklasse $O(n^3)$ ist immer schneller als einer der Klasse $O(2^n)$. *Für große n gilt dies, aber für kleine n kann ein Algorithmus aus einer ungünstigerer Komplexitätsklasse durchaus schneller sein.*
☒	☐	Es gilt $O(n^2) \neq O(n^{10})$.
☐	☒	Es gilt $O(log_2\ n) \neq O(log_{10}\ n)$.