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

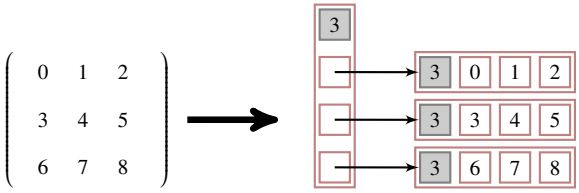
TU Kaiserslautern
Fachbereich Informatik
AG Bildverstehen und Mustererkennung

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

Ausgabe: in der Woche vom 20.12.10 bis zum 07.01.11
Abgabe: in der Woche vom 10.01. bis zum 14.01.11
Abnahme: max. zwei Tage nach der Übung

Aufgabe 1 Felder und Matrizenmultiplikation (Präsenzaufgabe)

Eine zweidimensionale Matrix kann mittels einfach verschachtelter Felder dargestellt werden (engl. *ragged 2D arrays*). Im Diagramm steht eine graue Box für die von Java intern verwaltete Länge eines Feldes (zum Beispiel: 3).



Implementieren Sie die unten angegebenen Prozeduren zur Arbeit mit Matrizen. `create` erstellt eine neue Matrix mit den übergebenen Ausmaßen. `set` setzt den einzelnen Wert m_{ij} in einer Matrix M neu, `get` liefert den Wert m_{ij}. `rows` liefert die Zahl der Zeilen der Matrix zurück, `cols` analog die Spalten (engl. *columns*). `output` soll eine Matrix zweidimensional als Text ausgeben.

```
int[][]  create(int rows, int columns)                { ... }
void        set(int[][] m, int i, int j, int value)   { ... }
int         get(int[][] m, int i, int j)              { ... }
int        rows(int[][] m)                            { ... }
int        cols(int[][] m)                            { ... }
void     output(int[][] m)                            { ... }
```

Aufgabe 2 Listen (Einreichaufgabe)

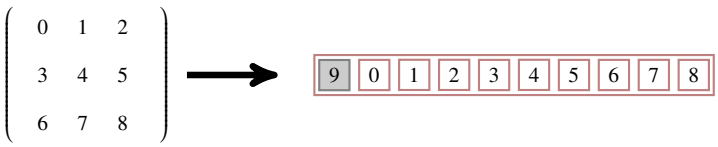
Implementieren Sie einen Verbunddatentyp für die Verwaltung von Listen von **`double`**-Werten in Java. Die Liste soll mit Hilfe eines Arrays realisiert werden, d.h. alle Elemente der Liste werden in einem Array der passenden Größe verwaltet. Folgende Verbunde und Prozeduren sollen Sie realisieren:

```
class List { ... }                    // der zu entwerfende Verbunddatentyp
List      create()                    // Erzeugt eine neue leere Liste.
void      append(List l, double elem) // Haengt ein Element an die Liste an.
int         size(List l)              // Liefert die Anzahl der Listenelemente.
double       get(List l, int index)   // Liefert das Element an der Stelle 'index'.
void         put(List l, int index, double elem) // Fuegt 'elem' als 'index' ein.
void      remove(List l, int index)   // Entfernt das Element mit 'index'.
boolean contains(List l, double elem) // Testet, ob 'elem' enthalten ist.
```

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. Verwenden Sie die bereits definierten Hilfsprozeduren aus Aufgabe 1 a).

b) Zweidimensionale Matrizen lassen sich auch mit einem eindimensionalen Feld darstellen. Wir benutzen dafür den Verbundtyp `Matrix`:



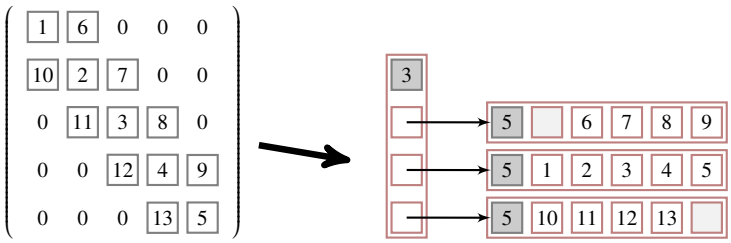

```
class Matrix {
  int rows, cols;
  int[] data;
}
```

Wie lautet die Formel, um von Zeile `i` und Spalte `j` auf den zugehörigen Feldindex zu kommen? Implementieren Sie die folgenden Prozeduren analog zu denen von Aufgabe 1 und Aufgabe 3 a).

```
Matrix  create(int zeilen, int spalten)            { ... }
void       set(Matrix m, int i, int j, int value)  { ... }
int        get(Matrix m, int i, int j)             { ... }
int       rows(Matrix m)                           { ... }
int       cols(Matrix m)                           { ... }
void    output(Matrix m)                           { ... }
Matrix    mult(Matrix a, Matrix b)                 { ... }
```

c) In der praktischen Anwendung von Matrizen treten sogenannte *Bandmatrizen* auf. Diese zeichnen sich durch zwei Dreiecke mit Nullen, links unten und rechts oben aus. Das *Band* entlang der Diagonale ist dann die wesentliche Information, die wir nun speichern und bearbeiten wollen.

Die Geometrie einer solchen Matrix wird also nicht nur durch die Zahl der Zeilen und Spalten, sondern auch durch die Zahl der Diagonalen, die nicht Null sind, bestimmt. Dabei ist die Bandbreite die Zahl der diagonalen Reihen auf einer Seite der Hauptdiagonalen. Die Grafik zeigt eine Bandmatrix mit der Breite eins:




```
class BandMatrix {
  int rows, cols;
  int bandwidth;
  int[][] data;
}
```

Implementieren Sie die folgenden Prozeduren:

```
BandMatrix  createB(int rows, int cols, int width)          { ... }
int         getRowB(BandMatrix m, int i, int j)             { ... }
void           setB(BandMatrix m, int i, int j, int value)  { ... }
int            getB(BandMatrix m, int i, int j)             { ... }
int           rowsB(BandMatrix m)                           { ... }
int           colsB(BandMatrix m)                           { ... }
void        outputB(BandMatrix m)                           { ... }
Matrix        multB(Matrix a, BandMatrix b)                 { ... }
```

Hinweis: Zur Adressierung des internen Feldes können Sie die Spaltennummer direkt verwenden, müssen die Zeilennummer jedoch konvertieren. Diese Berechnung wird durch die Prozedur `getRowB` gekapselt.

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?

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}$$

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.
```

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*. Verwenden Sie dabei folgende Vorlage:

```
boolean containsI(List l, double elem) { ... }
boolean containsR(List l, double elem) {
  return bcRec(l, elem, 0, size(l));
}
boolean bcRec(List l, double elem, int left, int right) { ... }
```

c) Schreiben Sie für alle geforderten Funktionen Tests, mit denen Sie sowohl Standard- als auch Randfälle abprüfen können. Das bedeutet beispielsweise mit `insert` mindestens je einmal ein Element an den Anfang, ans Ende und innerhalb der Liste einzufügen. Bei der binären Suchen sollten sie u.a. sicherstellen, dass alle Zweige Ihrer Implementierung einmal durchlaufen und auf Korrektheit geprüft werden.

Aufgabe 5 Aufwandsabschätzung (Einreichaufgabe)

Wir betrachten die Folge der Fibonacci-Zahlen, die durch die folgende Funktion berechnet wird:

```
int fib(int n) {
  switch(n) {
    case 0: return 0;
    case 1: return 1;
    default: return fib(n-1) + fib(n-2);
} }
```

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.

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?

Hinweis: Zur Ermittlung der Formel kann es unter Umständen hilfreich sein, jeweils die Werte von C und `fib` zueinander in Beziehung zu setzen, indem man zum Beispiel die jeweiligen Funktionswerte für $n \in \{0, \ldots, 5\}$ berechnet und die dadurch entstehende Tabelle genauer betrachtet.

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?

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.

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.
☐	☐	Folgender Anweisungsblock `{Paar p; p.ersteKomp = 5;}` (`Paar` ist ein Verbundtyp) wird fehlerfrei übersetzt.
☐	☐	Ein Geflecht besteht aus Verbunden und Feldern, die sich nicht gegenseitig referenzieren.
☐	☐	Eine Verbundkomponente kann eine Referenz auf den Verbund enthalten zu dem sie gehört.
☐	☐	Eine Variable kann überall genutzt werden, egal wo sie deklariert ist.
☐	☐	Ein Anweisungsblock definiert einen Deklarationsbereich.
☐	☐	Es gibt Java-Programme mit nur einem Deklarationsbereich.
☐	☐	Globale Variablen haben den größtmöglichen Gültigkeitsbereich.
☐	☐	Variablen aus dem gleichen Deklarationsbereich haben den gleichen Gültigkeitsbereich.
☐	☐	Auf eine gültige Variable kann in ihrem Deklarationsbereich immer zugegriffen werden.
☐	☐	Deklarations-, Gültigkeits- und Sichtbarkeitsbereiche ändern sich bei der Programmausführung.
☐	☐	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.
☐	☐	Bei Komplexitätsanalysen erhält man präzise Aussagen über Speicher- und Zeitbedarf eines Programms.
☐	☐	Die Aufwandsfunktion eines Programms beschreibt die Zahl der Operationen für den ungünstigsten Fall in Abhängigkeit von der Länge 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.
☐	☐	Verschiedene Implementierungen von Algorithmen der gleichen Zeit-Komplexitätsklasse sind nicht unbedingt gleich schnell.
☐	☐	Ein Algorithmus der Zeit-Komplexitätsklasse $O(n^3)$ ist immer schneller als einer der Klasse $O(2^n)$.
☐	☐	Es gilt $O(n^2) \neq O(n^{10})$.
☐	☐	Es gilt $O(log_2\ n) \neq O(log_{10}\ n)$.