Programmierkurs Java

Dr.-Ing. Dietrich Boles

Aufgaben zu UE 7 + 8 - Arrays

(Stand 23.11.2020)

Aufgabe 1:

Wir simulieren die Abstimmung bei "Deutschland sucht den Superstar". Die Zuschauer können anrufen und ihren Favoriten aus einer bestimmten Anzahl an Sängerinnen und Sängern wählen. Auf dem TV ausgeben wird letztendlich pro Sängerin bzw. Sänger die Anzahl an Zuschauern, die für sie bzw. ihn gestimmt haben, und zwar prozentual.

Schreiben Sie Java-Programm, in dem der Benutzer zunächst die Anzahl an Sängern und anschließend für jeden Sänger die Anzahl an Anrufen eingeben muss. Speichern Sie die Daten in einem geeigneten Array ab. Anschließend soll das Ergebnis der Abstimmung in Form eines Balkendiagramms auf den Bildschirm ausgegeben werden. Und zwar sollen entsprechend der prozentualen Verteilung der Telefonanrufe jeweils Balken aus `*`-Zeichen (100 % entsprechen dabei 100 *-Zeichen) sowie anschließend der absolute Wert der Telefonanrufe auf den Bildschirm ausgegeben werden.

Beispiel für einen Programmablauf (Eingaben stehen in <>):

```
Anzahl der Saenger (> 0): <4>
Anrufe für Saenger 1 (>= 0): <50>
Anrufe für Saenger 2 (>= 0): <50>
Anrufe für Saenger 3 (>= 0): <40>
Anrufe für Saenger 4 (>= 0): <60>

Abstimmungsergebnis:
************************* 50
************************* 50
******************** 40
****************************** 60
```

Aufgabe 2:

Schreiben Sie eine Funktion, die (horizontale) Histogramme auf dem Bildschirm darstellt. Die Funktion bekommt ein eindimensionales int-Array (mit nicht-negativen Werten) als Parameter übergeben und soll entsprechend lange *-Zeilen auf den Bildschirm zeichnen.

Beispiel:

`f([3, 1, 2, 2, 4, 1, 0, 3, 4])` produziert

```
***
*
**
**
****
*

***
****
```

Integrieren Sie diese Funktion in ein Java-Programm, mit dessen Hilfe diese Funktion getestet werden kann.

Aufgabe 3:

Schreiben Sie ein Programm zur Matrizenmultiplikation (siehe http://de.wikipedia.org/wiki/Matrix_%28Mathematik%29#Matrizenmultiplikation).

Aufgabe 4:

Bei dieser Aufgabe sollen Sie einen einfachen Fahrtroutenplaner implementieren. Stellen Sie sich dazu ein Netz von Städten vor, das durch Straßen miteinander verbunden ist. Schreiben Sie ein Programm, was genau folgendes tut:

- Zunächst wird die Anzahl *anzahl* an Städten eingelesen.
- Anschließend werden die Namen von *anzahl* Städten eingelesen.
- Danach wird die Anzahl *direkt* der Direktverbindungen zwischen einzelnen Städten eingelesen (Verbindungen sind bidirektional)
- Anschließend werden die *direkt* Direktverbindungen eingelesen, und zwar in der Form Ausgangsstadt, Zielstadt.
- Danach soll das Programm in einer Endlosschleife als Auskunftssystem dienen. Für jede Auskunft sollen jeweils zwei Städtenamen eingelesen werden. Das Programm berechnet dann, ob eine Verbindung (auch indirekte!) zwischen den beiden Städten existiert.

Achten Sie bei den Nutzereingaben auf mögliche Fehler!

Beispiel:

```
Vorbereitung:
5
Oldenburg Bremen Hamburg Frankfurt München
3
Oldenburg Bremen
Bremen Hamburg
Bremen Frankfurt

Auskunft:
Oldenburg Hamburg (Ausgabe: Verbindung existiert)
Frankfurt Oldenburg (Ausgabe: Verbindung existiert)
Hamburg München (Ausgabe: keine Verbindung)
```

Aufgabe 5:

Vermutlich wurden sie im alten China entdeckt – jene harmonischen Anordnungen von Zahlen in einem quadratischen Schema derart, dass die Summe der Elemente jeder Zeile, jeder Spalte und der beiden Diagonalen gleich einer Konstanten ist. Sie sollen im Folgenden ein Java-Programm entwickelt, das magische Quadrate erzeugt, und zwar mit einem Algorithmus, den schon die alten Chinesen entdeckt haben. Das Programm soll zunächst die Zahl der Zeilen und Spalten des Quadrats einlesen (>= 3, ungerade, <=99), dann das Magische Quadrat berechnen und es anschließend auf den Bildschirm ausgeben.

Der Algorithmus funktioniert so, dass mit der Zahl 1 angefangen wird. Anschließend wird versucht, die jeweils nächst höhere Zahl im Quadrat zu platzieren.

- Die Zahl 1 wird in das Feldelement senkrecht unter der Mitte des Quadrats eingetragen.
- Steht eine Zahl im Feldelement `quadrat[zeile][spalte]`, so kommt ihr Nachfolger in das Feldelement `quadrat[zeile+1][spalte+1]`, sofern dieses Feldelement nicht schon besetzt ist.
- Ist ein solches Feldelement `quadrat[zeile’][spalte’]` schon besetzt, versucht man es von dem besetzten Feldelement ausgehend mit dem Feldelement `quadrat[zeile’+1][spalte’-1]`, und wiederholt dies solange, bis man gemäß dieser Regel ein unbesetztes Feldelement findet (siehe bspw. die Platzierung der 6 im unten stehenden Beispiel).
- Läuft der Zeilenindex unten aus dem Feld heraus, beginnt er wieder oben; läuft er rechts aus dem Feld heraus, beginnt er wieder links.

Bei der Eingabe von 5 wird das folgende magische Quadrat erzeugt:

```
11 24  7 20  3
 4 12 25  8 16
17  5 13 21  9
10 18  1 14 22
23  6 19  2 15
```

Aufgabe 6:

Schreiben Sie eine Funktion, die eine quadratische (n x n) - Matrix um 90° nach rechts rotiert.

```
1 2 3         7 4 1
4 5 6   ->    8 5 2
7 8 9         9 6 3
```

Schreiben Sie weiterhin ein Java-Programm, das zunächst eine Zahl n und anschließend n x n Zahlen von der Tastatur einliest und diese in einer n x n-Matrix abspeichert. Zunächst soll die Originalmatrix auf dem Bildschirm ausgegeben werden. Anschließend soll mit dieser Matrix die Rotationsfunktion aufgerufen werden. Zum Schluss soll die rotierte Matrix auf den Bildschirm ausgegeben werden.

Aufgabe 7:

Implementieren Sie folgende Funktionen:

1. Eine Funktion, die zwei eindimensionale int-Arrays als Parameter übergeben bekommt und überprüft, ob die darin gespeicherten Werte jeweils gleich sind

   Beispiel: f([1, 4, 3], [1, 4, 3]) == true

2. Eine Funktion, die eine n x n Matrix mit int-Werten als Parameter übergeben bekommt und die überprüft, ob die Summe aller Reihen und Spalten und der beiden Diagonalen identisch ist.

   Beispiel: f([1,4,1],[2,2,2],[3,0,3]) == true

3. Eine Funktion, die drei eindimensionale int-Arrays als Parameter übergeben bekommt und die eine Matrix zurückliefert, bei der die drei Arrays die Zeilen bilden.

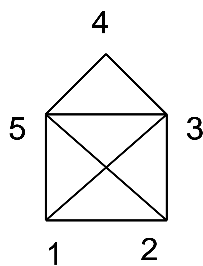
   Beispiel: f([1], [1,3,4], [2, 3]) == [[1], [1, 3, 4], [2, 3]]

4. Eine Funktion, die drei eindimensionale int-Arrays als Parameter übergeben bekommt und die eine Matrix zurückliefert, bei der die einzelnen Werte den Werten der Arrays entsprechen (wo ist der Unterschied zu Teilaufgabe (3)).

   Beispiel: f([1], [1,3,4], [2, 3]) == [[1], [1, 3, 4], [2, 3]]

Aufgabe 8:

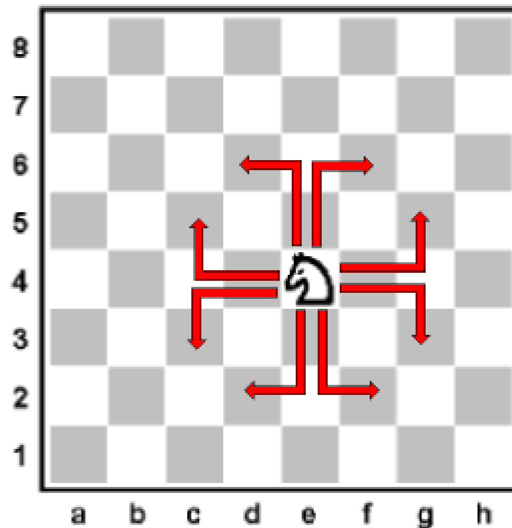
Sie haben in Ihrer Kindheit bestimmt mal das „Haus des Nikolaus“ gezeichnet. Hierbei geht es darum, das folgende Haus so zu zeichnen, dass weder der Stift abgesetzt wird noch eine Linie zweimal gezeichnet wird.



Schreiben Sie ein Programm, das alle Möglichkeiten zum Zeichnen des Haus des Nikolaus ausgibt, wenn mit dem Zeichnen in der linken unteren Ecke begonnen wird. Geben Sie jeweils die entsprechende Nummernfolge aus, z.B. 153125432

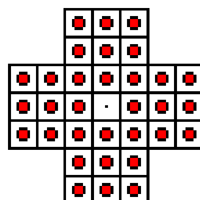
Aufgabe 9:

Stellen Sie sich ein Schachbrett (8x8 Felder) vor. Auf diesem Brett sitzt in der linken unteren Ecke ein einzelner Springer (Pferd). Versuchen Sie eine Folge von Zügen zu finden, so dass der Springer jedes Feld des Brettes genau einmal besucht. Geben Sie diese Folge auf den Bildschirm aus.



Aufgabe 10:

Schreiben Sie ein Java-Programm, das das Solitaire-Spiel löst. Die Abbildung skizziert das Spielfeld in der Ausgangssituation. Es ist ein Ein-Personen-Spiel. Ein Spielzug sieht so aus, dass der Spieler jeweils eine beliebige Figur auswählt, bei der ein Nachbarfeld besetzt und das dahinter liegende Feld frei ist. Der Spieler nimmt die Figur, platziert sie auf dem freien Feld und entfernt die übersprungene Figur vom Spielfeld. Ziel des Spiels ist es, eine Folge von Spielzügen zu finden, so dass zum Schluss nur noch eine einzige Spielfigur auf dem Spielfeld vorhanden ist. Ihr Programm soll eine solche Folge finden und die Spielzüge der Folge ausgeben.



Aufgabe 11:

Sie kennen sicher das Spiel *Sudoku*: Das Spiel besteht aus einem Gitterfeld mit 3 × 3 Blöcken, die jeweils in 3 × 3 Felder unterteilt sind, insgesamt also 81 Felder in 9 Reihen und 9 Spalten. In einige dieser Felder sind schon zu Beginn Ziffern zwischen 1 und 9 eingetragen. Typischerweise sind 22 bis 36 Felder von 81 möglichen vorgegeben. Ziel des Spiels ist es nun, die leeren Felder des Puzzles so zu vervollständigen, dass in jeder der je neun Zeilen, Spalten und Blöcke jede Ziffer von 1 bis 9 genau einmal auftritt.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

1. Schreiben Sie ein Programm, das ein vorgegebenes Sudoku löst und die Lösung auf den Bildschirm ausgibt. Sie können dabei das gegebene Sudoku als fest kodierte Matrix im Sourcecode repräsentieren.
2. Schreiben Sie ein Programm, das ein neues Sudoku erstellt.

Aufgabe 12:

Implementieren Sie einen interpretativen Hamster-Simulator (www.java-hamster-modell.de) mit ASCII-Repräsentation. Nutzen Sie folgende Symbole:

>	Hamster (Blickrichtung OST)
^	Hamster (Blickrichtung NORD)
<	Hamster (Blickrichtung WEST)
v	Hamster (Blickrichtung SUED)
o	Kachel mit mindestens einem Korn
#	Kachel mit Mauern

Nutzen Sie zur internen Repräsentation des Territoriums ein zwei-dimensionales Array. Der Anfangszustand des Territoriums kann im Sourcecode festgelegt werden.

Das Programm soll zunächst das Anfangsterritorium auf dem Bildschirm ausgeben. Anschließend kann der Nutzer wiederholt die Hamster-Befehle „vor", „linksUm", „nimm" bzw. „gib" sowie den Befehl „quit" eingeben. Nach Eingabe eines Hamster-Befehls wird dabei jeweils der Befehl ausgeführt und das resultierende Territorium auf den Bildschirm ausgegeben. Die Eingabe von „quit" beendet das Programm.

Beispielhafte Ausgabe des Territoriums:

```
+-+-+-+-+-+-+
| |o| | |#| |
+-+-+-+-+-+-+
| | |>| |o| |
+-+-+-+-+-+-+
| | |#|#| | |
+-+-+-+-+-+-+
```

Aufgabe 13:

In der heutigen „elektronischen Welt“ wird es immer wichtiger, Daten zu verschlüsseln, damit sie nicht in die Hände Fremder gelangen. Verschlüsselungsalgorithmen gibt es allerdings bereits sehr viel länger als es Computer gibt. Ein solches Verschlüsselungsverfahren nennt sich „Cäsar-Verschlüsselung“, benannt nach Julius Cäsar, der es als erster benutzt haben soll. Um diese Cäsar-Verschlüsselung geht es in dieser Aufgabe.

Die Cäsar-Verschlüsselung eines Textes bestehend aus Buchstaben funktioniert folgendermaßen: Verschiebe jeden Buchstaben um eine bestimmte vorgegebene Verschiebungsdistanz im Alphabet nach hinten. Beträgt die Verschiebungsdistanz beispielsweise 1 wird aus einem ‚a’ ein ‚b’, aus einem ‚b’ ein ‚c’, … und aus einem ‚z’ ein ‚a’ (d.h. am Ende des Alphabets wird wieder vorne begonnen). Beträgt die Verschiebungsdistanz 3 wird aus einem ‚a’ ein ‚d’, aus einem ‚b’ ein ‚e’, … und aus einem ‚z’ ein ‚c’.

Aufgabe (a): Schreiben Sie eine Funktion `verschluesseln` mit folgender Signatur `static char[] verschluesseln(char[] str, int verschiebung`). Die Funktion soll das übergebene char-Array entsprechend der Cäsar-Verschlüsselung verschlüsseln, und zwar mit einer Verschiebungsdistanz (beliebiger int-Wert, u.U. auch negativ!), der im zweiten Parameter übergeben wird. Dabei soll das übergebene Array selbst nicht verändert werden. Stattdessen sollen die verschlüsselten Zeichen in einem intern erzeugten Array gespeichert werden, das als Ergebniswert der Funktion geliefert wird. Verschlüsselt werden sollen dabei nur Kleinbuchstaben, alle anderen Zeichen sollen unverändert bleiben. Beispiel:

```
char[] text = {’a’, ’1’, ’z’};
char[] verschluesselterText = verschluesseln(text, 3);
// Ergebnis: verschluesselterText entspricht {’d’, ’1’, ’c’}
```

Aufgabe (b): Schreiben Sie ein Programm, das die Funktion `verschluesseln` ein einziges (!) Mal aufruft, um folgenden Text mit einer Verschiebungsdistanz von 3 zu verschlüsseln (Achten Sie darauf, dass der Text aus 2 Zeilen besteht!):

```
Mein Login ist "karl";
mein Passwort ist "meier"
```

Einen String `str` in ein char-Array zu verwandeln, funktioniert folgendermaßen:

```
char[] zeichen = str.toCharArray();
```

Ein char-Array `zeichen` in einen String umzuwandeln, funktioniert folgendermaßen:

```
String str = new String(zeichen);
```

Folgendes Ergebnis sollte auf dem Bildschirm erscheinen:

```
Mhlq Lrjlq lvw "nduo";
phlq Pdvvzruw lvw "phlhu"
```

Aufgabe 14:

Sie alle kennen aus Ihrer Kindheit das so genannte **Slider-Spiel**. Es ist ein Spiel für eine Person. Das Spielgerät ist eine Tafel mit 4 * 4 Feldern. In dieser Tafel sind auf

15 Feldern Plättchen mit den Ziffern 1 bis 15 platziert. Ein Feld ist leer. Die Plättchen sind verschiebbar. Gegeben eine bestimmte Ausgangsstellung der Plättchen ist es das Ziel, die Plättchen in der Reihenfolge 1 bis 15 anzuordnen.

Schreiben Sie ein Programm, mit dem ein Benutzer das Slider-Spiel spielen kann. Beachten und behandeln Sie falsche Benutzereingaben. Achten Sie auf einen sauberen Programmentwurf (prozedurale Zerlegung)! Wählen Sie aussagekräftige Bezeichner! Die Ausgangsstellung können Sie beliebig vorgeben.

Beispielablauf (Benutzereingaben in <>):

```
+--+--+--+--+
|11|12|13|14|
+--+--+--+--+
| 1| 2| 3| 4|
+--+--+--+--+
| 8| 7| 6| 5|
+--+--+--+--+
|  | 9|10|15|
+--+--+--+--+
Zeile: <3>
Spalte: <1>
+--+--+--+--+
|11|12|13|14|
+--+--+--+--+
| 1| 2| 3| 4|
+--+--+--+--+
| 8| 7| 6| 5|
+--+--+--+--+
| 9|  |10|15|
+--+--+--+--+
...

+--+--+--+--+
|  | 1| 2| 3|
+--+--+--+--+
| 4| 5| 6| 7|
+--+--+--+--+
| 8| 9|10|11|
+--+--+--+--+
|12|13|14|15|
+--+--+--+--+
Fertig!
```

Aufgabe 15:

Bei dieser Aufgabe geht es um das Mischen zweier sortierter Arrays. Implementieren Sie dazu folgende Funktion:

```
/**
  Die Funktion mischt die beiden uebergebenen Arrays zu
  einem neuen Array;
  Vorbedingung: Die beiden uebergebenen Arrays sind sortiert.
  Nachbedingung: Das gelieferte Array enthaelt alle Elemente der
    beiden uebergebenen Arrays und ist sortiert. Die beiden
    uebergebenen Arrays bleiben unveraendert.
  @param menge1 sortiertes Array (!= null)
  @param menge2 sortiertes Array (!= null)
  @return sortiertes Array mit allen Elemente der uebergebenen
          Arrays
*/
```

```java
public static int[] mischen(int[] menge1, int[] menge2);
```

Beispiel:

```java
int[] array1 = {1,3,3,5,6,9};
int[] array2 = {2,3,5,7,8,9,10};
int[] ergebnis = mischen(array1, array2);
// ergebnis == {1,2,3,3,3,5,5,6,7,8,9,9,10}
```

Mögliches Verfahren: Legen Sie für beide Arrays jeweils einen aktuellen Index an. Durchlaufen Sie die beiden Arrays. Vergleichen Sie die beiden Elemente des jeweils aktuellen Index. Fügen Sie das jeweils kleinere Element in das neue Array ein und erhöhen Sie den dazu gehörenden aktuellen Index.

Aufgabe 16:

Bei dieser Aufgabe geht es um das Ordnen einer ungeordneten Matrix (zweidimensionales Array). Ordnen bedeutet: Nach Aufruf der Methode sind die Elemente in der Matrix von oben nach unten und von links nach rechts in aufsteigender Größe sortiert. Implementieren Sie dazu folgende Funktion:

```java
public static void matrixOrdnen(int[][] matrix);
```

Beispiel:

```java
int[][] matrix = {{2, 4, 6},
                  {7, 3},
                  {2, 9, 8, 4}};
matrixOrdnen(matrix);
// matrix == {{2, 2, 3},
              {4, 4},
              {6, 7, 8, 9}};
```

Ein möglicher Algorithmus: Kopieren Sie die Elemente der Matrix in ein genügend großes eindimensionales Array, sortieren Sie dies und kopieren Sie anschließend die Elemente wieder zurück in die Matrix.

Aufgabe 17:

Implementieren Sie eine Funktion, die die transitive Hülle einer übergebenen Matrix berechnet:

```java
public static boolean[][] transitiveHuelle(boolean[][] matrix)
```

Sie können voraussetzen, dass die übergebene Objektvariable ungleich null ist und auf ein n x n-Array verweist, mit n >= 1. Die Performance Ihrer Funktion ist unwichtig!

Die **transitive Hülle** th(m) einer Matrix m sei dabei folgendermaßen definiert:

> Besitzt ein Element m[i][j] der Matrix den Wert true, dann bedeutet dies: Es existiert eine direkte Verbindung von i nach j. Besitzt ein Element m[i][j] der Matrix den Wert false, dann bedeutet dies: Es existiert keine direkte Verbindung von i nach j. In th(m), der transitiven Hülle von m, besitzen genau die Elemente th(m)[i][j] den Wert true, die eine direkte oder indirekte Verbindung zwischen i und j kennzeichnen. *Indirekt* bedeutet: es gibt ein m

und es gibt einen Pfad ($v_1, v_2, ..., v_m$) mit $i = v_1$ und $j = v_m$ und $m[v_j][v_{j+1}] = true$ für alle j = 1, ... m-1.

Beispiel:

```
m = F T F T      th(m) = T T T T
    F F T F              F T T F
    F T F F              F T T F
    T F F F              T T T T
```

Aufgabe 18:

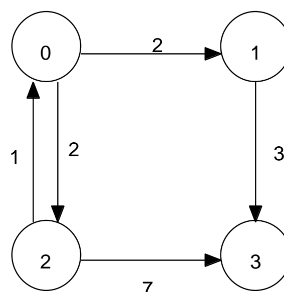
Implementieren Sie eine Funktion zur Lösung des „all pair shortest path problems“, also dem Finden der Länge der jeweils kürzesten Pfade zwischen allen Knoten in einem gerichteten Graphen (Sie können sich vorstellen, dass die Knoten des Graphen Städte und die Kanten des Graphen Einbahnstraßen zwischen den Städten repräsentieren.):

```
public static int[][] kuerzesteEntfernung(int[][] matrix)
```

Übergeben wird eine Matrix m, in der m[i][j] die Länge einer Kante zwischen einem Knoten i und einem Knoten j angibt (also die Länge einer Einbahnstraße zwischen den beiden Städten). Existiert keine Kante zwischen den Knoten i und j, so enthält m[i][j] den Wert –1. Sie können voraussetzen, dass die übergebene Objektvariable m ungleich null ist und auf ein n x n-Array verweist, mit n >= 1. Die Performance Ihrer Funktion ist unwichtig!

Berechnet und zurückgeliefert werden soll eine Matrix k, in der k[i][j] die kürzeste Entfernung zwischen den Knoten i und j bzgl. der Entfernungsangaben in der Matrix m angibt.

Beispiel:




```
m =  0  2  2 -1      k =  0  2  2  5
    -1  0 -1  3          -1  0 -1  3
     1 -1  0  7           1  3  0  6
    -1 -1 -1  0          -1 -1 -1  0
```

Aufgabe 19:

Das „Game-of-Life“ wird auf einem schachbrettartigen Feld gespielt, das eine „Bevölkerung“ von „toten“ und „lebenden“ Zellen darstellt. Jede Zelle kann „überleben“, „sterben“ oder „geboren“ werden. Die schrittweise Entwicklung von einem Stellungsbild zum nächsten erfolgt gemäß einiger Regeln, die berücksichtigen, wie viele lebende Nachbarzellen eine Zelle hat.

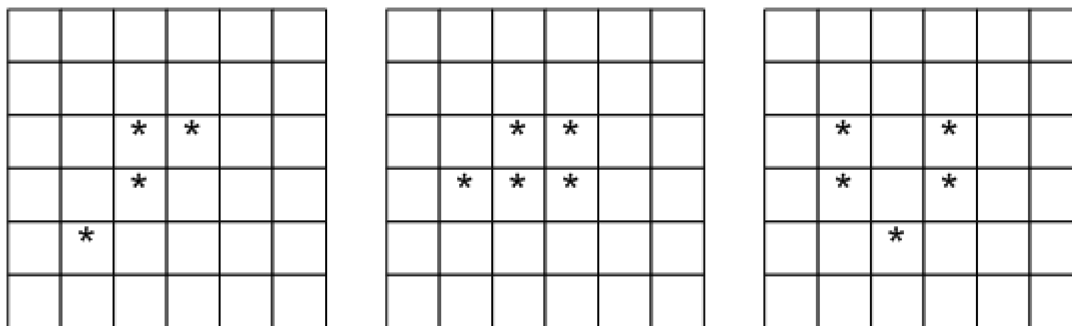
Bei dem Feld handelt es sich um ein Torus-förmiges Feld, bei dem alles, was das Feld nach unten verlässt, oben wieder herauskommt und umgekehrt, und alles, was das Feld nach links verlässt, rechts wieder eintritt und umgekehrt, d.h. alle Zellen haben jeweils genau 8 Nachbarzellen.

Die Regeln, nach denen sich die Population von einer Stellung zur nächsten entwickelt, sind:

1. Für eine Zelle x, die gerade tot ist, gilt: Wenn x genau 3 lebende Nachbarzellen hat, wird x neu geboren; sonst bleibt x tot.
2. Für eine Zelle x, die gerade lebendig ist, gilt: Wenn x weniger als 2 lebende Nachbarn hat, stirbt x an Vereinsamung; wenn x 2 oder 3 lebende Nachbarzellen hat, bleibt x in der nächsten Stellung lebendig. In allen anderen Fällen stirbt x an Überbevölkerung.

Alle Veränderungen gemäß dieser Regeln geschehen gleichzeitig. Die Simulation beginnt mit einer bestimmten eingelesenen Verteilung von lebenden und toten Zellen.

Beispiel:



Population 1 Population 2 Population 3

Aufgabe: Schauen Sie sich die folgende Implementierung des Game-of-Life an. Implementieren Sie die fehlende Funktion *calcNextGeneration* der Klasse *Generations* und testen Sie dann Ihr Programm.

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;

import javax.swing.JFrame;
import javax.swing.JPanel;

public class GameOfLife {

	// Ausgangsfeld
	static int[][] world = {
			{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
			{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
			{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
			{ 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0 },
			{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
			{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
			{ 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
			{ 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
```

```
                    { 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0 },
                    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0 },
                    { 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0 },
                    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 } };

    public static void main(String[] args) {
        JFrame frame = new JFrame("Game of Life");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new BorderLayout());
        WorldPanel panel = new WorldPanel(GameOfLife.world);
        panel.setPreferredSize(new  Dimension(GameOfLife.world[0].length
* 20,
                GameOfLife.world.length * 20));
        frame.add(panel, BorderLayout.CENTER);
        frame.pack();
        new Simulation(panel).start();
        frame.setVisible(true);
    }
}

class WorldPanel extends JPanel {

    private int[][] world;

    public WorldPanel(int[][] world) {
        this.world = world;
    }

    public int[][] getWorld() {
        return this.world;
    }

    public void setWorld(int[][] world) {
        this.world = world;
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        for (int r = 0; r < this.world.length; r++) {
            for (int s = 0; s < this.world[r].length; s++) {
                if (this.world[r][s] == 0) {
                    g.setColor(Color.WHITE);
                } else {
                    g.setColor(Color.DARK_GRAY);
                }
                g.fillRect(s * 20, r * 20, 20, 20);
            }
        }
    }
}

class Simulation extends Thread {

    WorldPanel panel;
```

```
	public Simulation(WorldPanel panel) {
		this.panel = panel;
	}

	public void run() {
		try {
			Thread.sleep(1000);
			while (true) {

	this.panel.setWorld(Generations.calcNextGeneration(this.panel
								.getWorld()));
				this.panel.repaint();
				Thread.sleep(500);
			}
		} catch (Exception exc) {
			exc.printStackTrace();
		}
	}
}

class Generations {

	/**
	 * Liefert ein Feld mit der nächsten Generation gemäß der
	 * Game-of-Life-Regeln.
	 *
	 * @param world
	 *            das aktuelle Feld; 1 entspricht: Zelle ist lebendig; 0
	 *            entspricht: Zelle ist tot
	 * @return ein Feld mit der nächsten Generation;1 entspricht: Zelle
ist
	 *         lebendig; 0 entspricht: Zelle ist tot
	 */
	static int[][] calcNextGeneration(int[][] world) {

	}

}
```

Aufgabe 20:

Definieren Sie eine Funktion, die als einzigen Parameter ein Array mit `double`-Werten übergeben bekommt und die den Mittelwert dieser `double`-Werte als `double`-Wert zurück liefert.

Aufgabe 21:

Definieren Sie eine Funktion, die zwei `int`-Arrays als Parameter übergeben bekommt und ein Ergebnis vom Typ `int` liefert. Die Funktion soll die Summe derjenigen Elemente des ersten Arrays berechnen, die durch die Elemente des zweiten Arrays indiziert werden.

Beispiel:

Erstes Array = {23, 4, 12, 14, 5, 9}; Zweites Array = {1, 2, 5}

Berechnung: 4 + 12 + 9 = 25

Lieferung des Wertes 25

Aufgabe 22:

In einem Kreis stehen *n* Kinder (durchnumeriert von 1 bis *n*). Mit Hilfe eines *m*-silbigen Abzählreims wird das jeweils *m*-te unter den noch im Kreis befindlichen Kindern ausgeschieden, bis kein Kind mehr im Kreis steht.

Schreiben Sie ein Java-Programm, das nach Vorgabe von *n* (positive Zahl) und *m* (positive Zahl) die Nummern der Kinder in der Reihenfolge ihres Ausscheidens angibt.

Beispiel*:* Für *n*=6 und *m*=5 ergibt sich die Folge 5, 4, 6, 2, 3, 1.

Aufgabe 23:

Implementieren Sie eine Funktion, die als Parameter ein int-Array übergeben bekommt. Die Funktion soll das

(a) größte,

(b) zweitgrößte

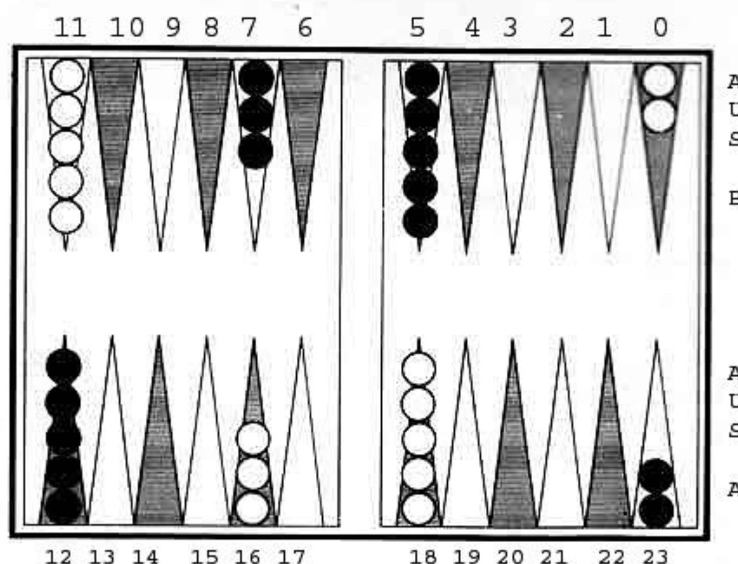
Element des Arrays ermitteln und als Funktionswert liefern. Dabei darf das Array nicht verändert werden.

Aufgabe 24:

Bei dieser Aufgabe sollen Sie ein Programm entwickeln, durch das zwei menschliche Spieler am Computer gegeneinander ein vereinfachtes Backgammon-Spiel spielen können.

Regeln

Das Backgammon-Spielbrett besteht aus 24 Feldern, „Zungen“ genannt, von denen sich jeweils 12 auf einer Seite befinden. Gespielt wird mit 15 weißen Steinen, die Spieler A gehören, und 15 schwarzen Steinen, die Spieler B gehören. Die Startaufstellung der Steine sehen Sie in der folgenden Abbildung.



Spieler A zieht seine Steine gegen, Spieler B mit dem Uhrzeigersinn. Beide versuchen, alle ihre Steine in ihr jeweiliges „Aus“ zu bringen. Wem dies als erstem gelingt, hat das Spiel gewonnen.

Die Spieler ziehen immer abwechselnd. Spieler A beginnt das Spiel. Dazu würfelt er mit einem einzelnen Würfel (Zahlen 1 bis 6). Entsprechend der gewürfelten Zahl, darf er einen seiner Steine entsprechend viele Zungen in seine entsprechende Zugrichtung fortbewegen. Würfelt Spieler A bspw. in der Startaufstellung eine 4, dürfte er bspw. einen Stein von Zunge 0 auf Zunge 4 bewegen. Würfelt Spieler B anschließend eine 5, dürfte er bspw. einen Stein von Zunge 7 auf Zunge 2 verschieben. Gezogen wird dabei immer der oberste Stein einer Zunge. Steine landen im „Aus“, wenn sie über die letzte Zunge des entsprechenden Spielers hinausgezogen werden können.

Beim Ziehen gelten folgende Einschränkungen:

- Auf jeder Zunge dürfen sich zu jedem Zeitpunkt maximal 5 Steine befinden.
- Ein Stein darf nicht auf eine Zunge gezogen werden, die gerade durch einen oder mehrere gegnerische Steine besetzt ist.

Kann ein Spieler, wenn er an der Reihe ist, nicht ziehen, muss er passen. Passen ist darüber hinaus prinzipiell immer möglich.

Alle anderen Regeln des Original-Backgammon-Spiels gelten bei dieser vereinfachten Variante nicht!

Hinweise zur Umsetzung:

- Wählen Sie als Datenstruktur zur Repräsentation des Spielfeldes ein int-Array der Länge 24. Jedes Array-Element repräsentiert dabei eine Zunge. Beachten Sie, dass das Spiel prinzipiell auch funktionieren soll, wenn eine beliebige andere Länge als 24 (die aber durch 2 teilbar sein muss) gewählt wird. Eine Implementierung des Spiels auf der Grundlage einer festgelegten Codierung der Felder und Züge ist nicht erlaubt!
- Steine von Spieler A auf einer Zunge werden durch eine entsprechend hohe positive Zahl, Steine von Spieler B durch eine entsprechend hohe negative Zahl repräsentiert. Die 0 deutet an, dass die Zunge momentan leer ist. In der Darstellung werden Steine von Spieler A durch ein ‚+’, Steine von Spieler B durch ein ‚*’ repräsentiert.
- Ein Spielzug entspricht der Angabe des Indexes der entsprechenden Zunge, von der ein Stein gezogen werden soll (also die Werte 0 bis 23). Passen wird durch die Eingabe von `-1` signalisiert.
- Geben Sie das Spielbrett analog zu der Darstellung auf der nächsten Seite aus.
- Der Spielablauf lässt sich folgendermaßen skizzieren:

  Ausgabe des Spielbrettes
  Solange das Spiel noch nicht beendet ist, tue folgendes:

  > *Würfeln;*
  > *Zugeingabe des Spielers, der an der Reihe ist*
  > *Überprüfung des Zuges auf Gültigkeit und ggfls. Wiederholung*
  > *Ausführen des Spielzugs*
  > *Ausgabe des Spielbrettes*

Bekanntgabe des Siegers

Beispiel für einen Programmablauf (in <> stehen Benutzereingaben):

```
 11 10  9  8  7  6  5  4  3  2  1  0
  +           *     *              +
  +           *     *              +
  +           *     *
  +                 *
  +                 *

  *                 +
  *                 +
  *           +     +
  *           +     +              *
  *           +     +              *
 12 13 14 15 16 17 18 19 20 21 22 23
Würfel = 6; Spieler A, Zungennummer eingeben: <0>

 11 10  9  8  7  6  5  4  3  2  1  0
  +           *  +  *              +
  +           *     *
  +           *     *
  +                 *
  +                 *

  *                 +
  *                 +
  *           +     +
  *           +     +              *
  *           +     +              *
 12 13 14 15 16 17 18 19 20 21 22 23
Würfel = 2; Spieler B, Zungennummer eingeben: <11>
Falscher Zug! Bitte wiederholen!
Würfel = 2; Spieler B, Zungennummer eingeben: <12>

 11 10  9  8  7  6  5  4  3  2  1  0
  +  *        *  +  *              +
  +           *     *
  +           *     *
  +                 *
  +                 *

                    +
  *                 +
  *           +     +
  *           +     +              *
  *           +     +              *
 12 13 14 15 16 17 18 19 20 21 22 23
Würfel = 3; Spieler A, Zungennummer eingeben:

...
```

Aufgabe 25:

Implementieren Sie einen einfachen Taschenrechner, der auf der *Umgekehrten Polnischen Notation* (UPN) (auch Postfix-Notation genannt) basiert. Der Taschenrechner soll die binären Operationen +, -, *, / und % auf int-Werten unterstützen. Benutzer können nur einstellige Zahlen, also Zahlen zwischen 0 und 9,

eingeben. Das Programm soll enden, sobald der Benutzer das Zeichen ‚e' (für exit) eingibt.

Bei der UPN werden eingelesene Zahlen jeweils oben auf einen Stapel (maximal N Elemente, N hier 20) gelegt. Wird ein Operator eingegeben, werden die beiden oberen Elemente vom Stapel entfernt, darauf die Operation angewendet, das Ergebnis ausgegeben und das Ergebnis wieder auf den Stapel gelegt.

`9 + (7 - 2) / 4` in der Infix-Notation entspricht bspw. `9 7 2 - 4 / +` in der UPN

Beispiel für einen Programmablauf:

```
9
7
2
-
→ 5
4
/
→ 1
+
→ 10
e
```

Beachten Sie bitte mögliche Fehlerfälle und reagieren Sie darauf durch entsprechende Ausgaben!

Aufgabe 26:

Implementieren Sie in Java eine boolesche Funktion `istElement` mit folgender Signatur:

```
static boolean istElement(String[] vektor, String element)
```

Übergeben wird der Funktion als erster Parameter ein nach Größe sortiertes vollständig gefülltes Array mit Zeichenketten (ungleich `null`) und als zweiter Parameter eine einzelne Zeichenkette (ungleich `null`). Die Funktion soll überprüfen, ob die einzelne Zeichenkette wertmäßig im Array vorhanden ist.

Der Überprüfungsalgorithmus ist dabei vorgegeben, und zwar soll das durch die Übungsaufgaben bekannte Halbierungsverfahren genutzt werden: Es wird das mittlere Array-Element bestimmt (bei einer gerade Anzahl an Elementen eines der beiden mittleren Elementen) und mit dem `element`-Parameter verglichen. Sind die beiden wertgleich, kann `true` geliefert werden. Ist der Wert des `element`-Parameters kleiner, wird derselbe Algorithmus auf die erste Hälfte des Arrays angewendet. Ist der Wert des `element`-Parameters größer, wird derselbe Algorithmus auf die zweite Hälfte des Arrays angewendet.

Implementieren Sie eine vollständig rekursive Lösung, d.h. es dürfen keine Wiederholungsanweisungen benutzt werden

Nutzen Sie zum Vergleichen von `String`-Objekten die Instanzmethode `public int compareTo(String obj)` der Klasse `java.lang.String`. Sie liefert einen Wert kleiner als 0, wenn das `String`-Objekt, für das sie aufgerufen wurde, kleiner

als das als Parameter übergebene `String`-Objekt ist. Wenn die beiden `String`-Objekte gleich sind, wird 0 zurückgegeben, ansonsten ein positiver Wert.

```
String str1 = …
String str2 = …
int vergleich = str1.compareTo(str2);
```

Aufgabe 27:

ISBN-Nummern sind eindeutige Kennzeichner von Büchern. Bis 2006 waren die ISBN-Nummern 10stellig, heute werden 13stellige ISBN-Nummern verwendet. Allerdings gibt es einen Algorithmus, um 13stellige ISBN-Nummern in äquivalente 10stellige ISBN-Nummern umzurechnen. Dieser Algorithmus hat folgende Gestalt:

- Streiche die ersten 3 Ziffern
- Übernehme die nächsten 9 Ziffern
- Ersetze die letzte Ziffer durch ein so genanntes Prüfzeichen `p`.

Das Prüfzeichen `p` ergibt sich aus den 9 Ziffern z_i (i = 1, …, 9) dabei auf folgende Art und Weise:

Sei $S = \left(\sum_{i=1}^{9} (z_i * i) \right) \% 11$, dann ist p = ‚X', falls S = 10, ansonsten ist p = S.

Beispiele:

```
ISBN13 = 9781234567880 ISBN10 = 1234567881
ISBN13 = 9781234567890 ISBN10 = 123456789X
ISBN13 = 9781200000001 ISBN10 = 1200000005
```

Aufgabe: Implementieren Sie die folgende Funktion zur Umrechnung von ISBN13 in ISBN10

```
static char[] getISBN10(char[] isbn13)
```

Sie können dabei davon ausgehen, dass das als Parameter übergebene char-Array eine korrekte ISBN13 enthält.

Aufgabe 28:

Schreiben Sie eine Funktion, die beim Schachspiel eingesetzt werden könnte, und zwar soll mit Hilfe der Funktion überprüft werden, ob sich zwei Damen gegenseitig werfen können (zwei Damen können sich genau dann werfen, wenn sie in derselben Spalte, derselben Reihe oder derselben Diagonale stehen).

Die Funktion soll als Parameter eine quadratische Matrix mit boolean-Werten übergeben bekommen, in der genau zwei Felder den Wert true habe. Die beiden

Felder sind die von den Damen besetzten Felder. Die Funktion soll genau dann true liefern, wenn die Damen sich werfen können.

Aufgabe 29:

Masken sind char-Arrays, bei denen das Zeichen `?` die Bedeutung eines Jokers beim Vergleich mit einem anderen char-Array hat. Jeder Joker in der Maske steht für genau ein beliebiges Zeichen im char-Array. Zum Beispiel passt die Maske `a?b?` auf das char-Array `afbh`, dagegen nicht auf `abcd` oder `acb`.

Schreiben Sie eine **rekursive** Funktion `match`, die als Argumente eine Maske und ein char-Array erhält und beide vergleicht. `match` liefert genau dann `true`, wenn die Maske zum char-Array passt, ansonsten `false`.

Schreiben Sie ein kleines Testprogramm für die Funktion `match`.

Wenn Sie eine rekursive Lösung nicht hinbekommen, können Sie auch eine iterative Funktion implementieren.

Aufgabe 30:

Implementieren Sie in Java eine Funktion `plattmachen`, die ein beliebiges 2-dimensionales int-Array als Parameter übergeben bekommt und ein 1-dimensionales int-Array liefert, in das das als Parameter übergebene Array Zeile für Zeile von links nach rechts kopiert worden ist. Beachten Sie bitte auch Sonderfälle!

Beispiel:

Parameter (2-dimensionales Array):

```
[2, 3, 4, 5]
[6, 7, 8, 9]
[3, 5, 1, 8]
```

Resultat (1-dimensionales Array):

```
[2, 3, 4, 5, 6, 7, 8, 9, 3, 5, 1, 8]
```

Aufgabe 31:

Implementieren Sie das folgende kleine Spiel, bei dem ein Computer gegen einen Menschen spielt:

Das Spiel wird solange gespielt, bis der Mensch verloren oder 10 Punkte erreicht hat. Es besteht aus mehreren Runden. In jeder Runde generiert der Computer zufällig einen Kleinbuchstaben (‚a' … ‚z') und gibt diesen auf den Bildschirm aus. Der Benutzer muss dann innerhalb von jeweils 3 Sekunden überprüfen, ob genau dieser Buchstabe bereits zum zweiten, vierten, sechsten (also einem Vielfachen von 2) Mal auf dem Bildschirm erschienen ist. In diesem Fall (und nur in diesem) muss er eine beliebige Taste drücken. Hat er Recht, bekommt er einen Punkt. Hat er Unrecht oder überschreitet die Zeitgrenze von 3 Sekunden, hat er unmittelbar verloren. Erreicht er 10 Punkte, hat er gewonnen.

Achtung: Implementieren Sie das Spiel auf eine imperative Art und Weise. Sie können dabei folgende Klasse benutzen:

```
public class Util {

      /**
       * Blockiert das Programm für eine bestimmt Zeit und
       * ueberprueft, ob während dieser Zeit eine Taste
       * gedrückt wurde.
       *
       * @param secs
       *            Anzahl an Sekunden, die gewartet werden soll
       * @return liefert genau dann true, wenn innerhalb von
       *            secs-Sekunden eine Taste gedrückt wurde
       */
      public static boolean keyPressed(int secs)
}
```

Aufgabe 32:

Implementieren Sie eine Java-Funktion, die für ein char-Array überprüft, ob die dort enthaltenen runden Klammern den Regeln einer vollständigen Klammerung wie bei einem Ausdruck entsprechen. Das heißt: Für jede öffnende Klammer '(' muss eine nachfolgende schließende Klammer ')' existieren und die runden Klammern müssen korrekt verschachtelt sein. Zeichen außer den runden Klammern sollen ignoriert werden.

Folgendes sind korrekt geklammerte Zeichenketten: "()", "", "(()(a)(()((c))))". Diese nicht: "(()", "a (()()) a)", "ww)(".

Aufgabe 33:

Implementieren Sie in Java eine boolesche Funktion `istEnthalten`, die zwei `int`-Arrays übergeben bekommt. Die Funktion soll genau dann `true` liefern, wenn das zweite Array elementweise im ersten Array enthalten ist. Dabei gilt: Ein Array *a* ist genau dann elementweise in einem Array *b* enthalten, wenn alle Elemente von *a* in derselben Reihenfolge und ohne Unterbrechungen auch in *b* enthalten sind. Sie können davon ausgehen, dass keine `null`-Werte als Parameter übergeben werden. Die Nutzung von Strings ist nicht erlaubt.

Beispiel:

```
int[] b = {2, 4, 3, 7, 5, 6};
int[] a1 = {3, 7, 5};
int[] a2 = {3, 7, 6};
```

`a1` ist in `b` enthalten, `a2` ist nicht in `b` enthalten.

Aufgabe 34:

Euro-Banknoten der ersten Serie haben eine eindeutige Seriennummer, die aus einem führenden Großbuchstaben, einer Zahl mit 10 Ziffern und einer Prüfziffer bestehen.

Beispiel: `Z 6016220022 6`

Der führende Buchstabe codiert die nationale Zentralbank (NZB), die den Geldschein in Umlauf gebracht hat. Sie wird NZB-Nummer genannt.

Die Prüfziffer berechnet sich wie folgt:

- Der Buchstabe wird durch seine Position im lateinischen Alphabet ersetzt (bei A also 1, bei Z 26)
- Es wird die Quersumme dieser Positionszahl und der 10 Ziffern berechnet (im Beispiel 2+6+6+0+1+6+2+2+0+0+2+2 = 29)
- Von der Quersumme wird der ganzzahlige Rest zum nächst kleineren Vielfachen von 9 bestimmt (Modulo 9) (im Beispiel 2)
- Der Rest wird von 8 subtrahiert. Das Resultat ist die Prüfziffer (im Beispiel 6). Es sei denn das Resultat ist 0, dann ist die Prüfziffer 9.

Implementieren Sie in Java eine Funktion, die als ersten Parameter eine NZB-Nummer als Buchstaben und als zweiten Parameter eine 10-elementiges Array mit int-Werten übergeben bekommt. Die Funktion soll die Prüfziffer der entsprechenden Euro-Banknote berechnen und als int-Wert zurückliefern. Sie können davon ausgehen, dass die übergebenen Parameter korrekt sind.

Aufgabe 35:

Bei dieser Aufgabe sollen Sie ein Programm entwickeln, mit dem Nutzer ihre Erinnerungsfähigkeit trainieren können. Die Spielidee wurde von dem bekannten Spiel *Senso* (http://de.wikipedia.org/wiki/Senso_(Spiel)) übernommen. Der Programmablauf sieht folgendermaßen aus:

- Es werden maximal *n* Runden gespielt (konkret sei n = 50).
- In jeder Runde wird folgendes gemacht:
    - Das Programm ermittelt eine Zufallszahl zwischen 0 und 9.
    - Anschließend werden nacheinander die Zufallszahlen der bisherigen Runden in der entsprechenden Reihenfolge durch ein Leerzeichen getrennt in einer Zeile auf den Bildschirm ausgegeben. Nach der Ausgabe jeder Zahl wird dabei 1 Sekunde gewartet.
    - Nach der Ausgabe aller Zahlen wird der Bildschirm gelöscht.
    - Nun ist der Nutzer an der Reihe. Er muss sich alle ausgegebenen Zahlen merken und sie in der entsprechenden Reihenfolge eingeben.
    - Sobald der Benutzer einen Fehler macht, wird das Programm beendet und die Anzahl der erfolgreich absolvierten Runden auf den Bildschirm ausgegeben.
- Schafft der Nutzer alle n Runden, wird das Programm ebenfalls mit einer entsprechenden Erfolgsmeldung beendet.

Beispiel für einen Programmablauf (Benutzereingaben stehen in Klammern (<>)):

```
1
„Bildschirm wird gelöscht"
<1>
1 5
„Bildschirm wird gelöscht"
<1>
```

```
<5>
1 5 4
„Bildschirm wird gelöscht“
<1>
<5>
<4>
1 5 4 1
„Bildschirm wird gelöscht“
<1>
<5>
<4>
<1>
1 5 4 1 9
„Bildschirm wird gelöscht“
<1>
<5>
<1>
Fehler! Sie haben 4 Runden erfolgreich überstanden
```

Sie können dabei folgende Funktionen benutzen:

```
	// wartet sec Sekunden
	static void wait(int sec) {
		try {
			Thread.sleep(1000 * sec);
		} catch (Exception exc) {
		}
	}

	// loescht die Console (nicht wirklich)
	static void clearScreen() {
		for (int i = 0; i < 50; i++) {
			System.out.println();
		}
	}

	// liefert Zufallszahl zwischen 0 und max (einschließlich)
	static int getRandomNumber(int max) {
		return (int) (Math.random() * (max + 1));
	}
```

Aufgabe 36:

Die Polybios-Chiffre ist eine monographische bipartite monoalphabetische Substitution. Sie überträgt Zeichen in Zeichengruppen. Zur Übersetzung in Zeichenpaare sucht man das gewünschte Einzelzeichen (Buchstaben) in einer Polybios-Matrix heraus. Aus den Koordinaten des Buchstabens erhält man die gesuchte Kodierung. Aus dem Klartext HALLO wird beispielsweise der Geheimtext 2311313134. (aus Wikipedia)

Entwerfen und implementieren Sie ein Programm, das Folgendes leistet:

- Einlesen eines Klartextes
- Verschlüsseln des Klartextes mit Hilfe der Polybios-Tafel
- Ausgabe des Geheimtextes

Polybios-Tafel:

```
  1 2 3 4   5
1 a b c d   e
```

```
2 f g h i/j k
3 l m n o   p
4 q r s t   u
5 v w x y   z
```

Aufgabe 37:

Gegeben ist eine Tabelle mit acht Zeilen und zehn Spalten. In der Tabelle ist in den Zeilen und Spalten u. a. das Wort HUND viermal, KATZE zweimal und MAUS dreimal enthalten.

```
R X O J K H C K U H
S K I M A U S U H U
B N H A T N C H U N
R H M U Z D H U N D
M A U S E K A T Z E
A N H E N U F E T T
U D I N O H U N D E
D N U H T E R G L Z
```

Beachten Sie den Hinweis: Die Zeilen sind von links nach rechts und die Spalten von oben nach unten zu lesen. Entwerfen und implementieren Sie ein Programm, das Folgendes leistet:

- Einlesen einer Tabelle mit maximal 20 Zeilen und maximal 20 Spalten,
- Einlesen eines Wortes,
- Ermitteln der Anzahl des Vorkommens dieses Wortes in der Tabelle,
- Ausgabe der Anzahl.

Aufgabe 38:

In der Vorlesung haben Sie den Bubblesort-Algorithmus zum Sortieren von Arrays kennen gelernt. Es gibt viele weitere Sortier-Algorithmen; siehe bspw. http://de.wikipedia.org/wiki/Sortierverfahren. Implementieren Sie folgende Funktion mehrfach durch Anwendung verschiedener Sortieralgorithmen:

```
public void sortieren(int[] zahlen)
```

Aufgabe 39:

Bei dieser Variante des so genannten Nim-Spiels sind n Reihen mit n Streichhölzern vorhanden. Zwei Spieler nehmen abwechselnd Streichhölzer aus einer der Reihen weg. Wie viele sie nehmen, spielt keine Rolle; es muss mindestens ein Streichholz sein und es dürfen bei einem Zug nur Streichhölzer einer einzigen Reihe genommen werden. Derjenige Spieler, der den letzten Zug macht, also die letzten Streichhölzer wegnimmt, gewinnt.

Implementieren Sie diese Variante des Nim-Spiels in Java, so dass zwei menschliche Spieler gegeneinander antreten können.

Aufgabe 40:

In dieser Aufgabe geht es um die Verschlüsselung von Texten. Klartexte über einem Klartextalphabet werden durch die Anwendung von Verschlüsselungsalgorithmen in Geheimtexte über einem Geheimtextalphabet überführt. Verschlüsselungsverfahren sollen gewährleisten, dass nur Befugte bestimmte Botschaften lesen können.

Eine der beiden grundlegenden Verschlüsselungsklassen ist die *Transposition*. Dabei werden die Zeichen einer Botschaft (des Klartextes) umsortiert. Jedes Zeichen bleibt zwar unverändert erhalten, jedoch wird die Stelle, an der es steht, geändert. Als sehr einfaches und anschauliches Beispiel einer geregelten Transposition soll hier die *Gartenzaun-Transposition* dienen: Die Buchstaben des Klartextes werden abwechselnd auf zwei Zeilen geschrieben, so dass der erste auf der oberen, der zweite auf der unteren, der dritte Buchstabe wieder auf der oberen Zeile steht und so weiter. Der Geheimtext entsteht, indem abschließend die Zeichenkette der unteren Zeile an die der oberen Zeile angefügt wird. (aus Wikipedia)

Beispiel:

Klartext: `Gartenzaun`

Verfahren:

```
G r e z u
 a t n a n
```

Geheimtext: `Grezuatnan`

Aufgabe:

Implementieren Sie die folgenden zwei Funktionen mit Hilfe der Gartenzaum-Transposition:

```
// liefert den verschluesselten Text
static char[] verschluesseln(char[] klartext);

// liefert den entschluesselten Text
static char[] entschluesseln(char[] geheimtext);
```

Aufgabe 41:

Implementieren Sie das folgende kleine Taktikspiel für *n* (>=2) Spieler.

Lesen Sie anfangs die Anzahl *n* der teilnehmenden Spieler ein. Alle *n* Spieler besitzen je einen Haufen von 100 Kugeln. Gespielt werden 10 Runden. In jeder Runde wählt jeder Spieler (geheim) eine bestimmte Anzahl (>= 0) an Kugeln aus, die von seinem Haufen entfernt werden. Jeweils der bzw. die Spieler mit der größten Anzahl an gewählten Kugeln gewinnt/gewinnen die Runde und einen Punkt. Sieger des Spiels ist der/sind die Spieler, der/die nach 10 Runden die meisten Punkte hat/haben.

Orientieren Sie sich bei der Implementierung des Spiels an folgendem Beispielablauf (Benutzereingaben in <>). Behandeln Sie fehlerhafte Benutzereingaben adäquat.

```
Anzahl an Spielern: <3>

Runde 1 von 10 Runden
Spieler 1: Wie viele Kugeln von 100? <10>
Spieler 2: Wie viele Kugeln von 100? <9>
Spieler 3: Wie viele Kugeln von 100? <5>
Spieler 1 hat die Runde (mit) gewonnen und bekommt einen Punkt!
Spielstand nach Runde 1:
Spieler 1 hat 1 Punkte und 90 Restkugeln
Spieler 2 hat 0 Punkte und 91 Restkugeln
Spieler 3 hat 0 Punkte und 95 Restkugeln

Runde 2 von 10 Runden
Spieler 1: Wie viele Kugeln von 90? <8>
Spieler 2: Wie viele Kugeln von 91? <8>
Spieler 3: Wie viele Kugeln von 95? <6>
Spieler 1 hat die Runde (mit) gewonnen und bekommt einen Punkt!
Spieler 2 hat die Runde (mit) gewonnen und bekommt einen Punkt!
Spielstand nach Runde 2:
Spieler 1 hat 2 Punkte und 82 Restkugeln
Spieler 2 hat 1 Punkte und 83 Restkugeln
Spieler 3 hat 0 Punkte und 89 Restkugeln

...

Spielstand nach Runde 10:
Spieler 1 hat 5 Punkte und 0 Restkugeln
Spieler 2 hat 4 Punkte und 0 Restkugeln
Spieler 3 hat 5 Punkte und 0 Restkugeln

Endstand:
Spieler 1 ist der bzw. einer der Sieger des Spiels!
Spieler 3 ist der bzw. einer der Sieger des Spiels!
```

Aufgabe 42:

Gegeben sei ein beliebig großes Array *zahlen* mit beliebigen int-Werten sowie ein positiver int-Wert *divisor*. Berechnen Sie die Anzahl an Teilfolgen von Elementen des Arrays (*n* aufeinanderfolgende Elemente, *n* >= 1), deren Summe durch *divisor* teilbar ist, und geben Sie die Anzahl aus.

Hinweis: Die gegebenen Zahlenwerte brauchen Sie nicht einlesen, sondern können sie im Programm in Form von Konstanten vorgeben.

Beispiel:

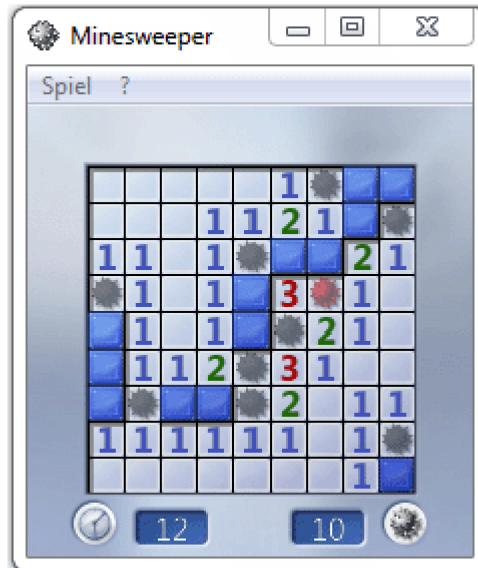
```
final int[] zahlen = { 2, 1, 2, 1, 1, 2, 1, 2 };
final int divisor = 4;

Ausgabe = 6
```

Aufgabe 43:

Sie kennen sicher das Spiel *Minesweeper*. Es ist ein Computerspiel, bei dem ein Spieler durch logisches Denken herausfinden muss, hinter welchen Feldern einer N * M-Matrix Mienen versteckt sind.

Während des Spiels wird beim Anklicken eines Feldes angezeigt, wie viele Mienen auf Nachbarfeldern platziert sind. Jedes Feld hat dabei 8 Nachbarfelder, Felder an den Rändern entsprechend weniger. Ziel des Spiels ist es, alle sicheren Felder zu finden. Sobald man auf ein Mienenfeld klickt, hat man verloren.



Bei dieser Aufgabe geht es nicht um die Implementierung des Spiels selber. Vielmehr wird in einer M*N-Ausgangsmatrix mit char-Zeichen der Anfangszustand eines Minesweeper-Spiels repräsentiert. Mienen werden dabei durch ein ‚*'-Zeichen und sichere Felder durch ein ‚.'-Zeichen repräsentiert. Ihre Aufgabe ist es, für diese gegebene Matrix die Minesweeper-Lösung zu finden und auf den Bildschirm auszugeben. Im folgenden Beispiel wird links eine 4*4-Ausgangsmatrix und rechts die zu berechnende und auszugebende Lösung skizziert.

```
*...    *100
....    2210
.*..    1*10
....    1110
```

Hinweis: Gehen Sie bei Ihrer Implementierung von einer im Quellcode vorgegebenen Ausgangsmatrix aus (die aber prinzipiell beliebig sein kann).

Beispiel 1:

```
public static void main(String[] args) {
      char[][] feld = { { '*', '.', '.', '.' },
                        { '.', '.', '.', '.' },
                        { '.', '*', '.', '.' },
                        { '.', '.', '.', '.' } };
      // Ihre Lösung

}
```

Produziert die folgende Bildschirmausgabe:

```
*100
2210
1*10
1110
```

Beispiel 2:

```
public static void main(String[] args) {
      char[][] feld = { { '*', '*', '.', '.', '.' },
                        { '.', '.', '.', '.', '.' },
                        { '.', '*', '.', '.', '.' } };

      // Ihre Lösung

   }
```

Produziert die folgende Bildschirmausgabe:

```
**100
33200
1*100
```

Aufgabe 44:

Diese Aufgabe besteht in der Entwicklung eines Interpreters für die esoterische Programmiersprache *Brainfuck*.

Esoterische Programmiersprachen:

Esoterische Programmiersprachen sind Programmiersprachen, die nicht primär für den praktischen Einsatz entwickelt wurden, sondern ungewöhnliche Sprachkonzepte umsetzen. Sie reizen mit spektakulären und zuweilen sogar genialen Ideen, wie dem kleinstmöglichen Compiler, einer Arithmetik ohne Plus, Mal, Durch und Minus, dem kleinsten Symbolvorrat oder sogar der Unsichtbarkeit der Programme. Mit Esoterik im umgangssprachlichen Sinn haben "esoterische Programmiersprachen" allerdings nichts zu tun. Die Beschäftigung mit esoterischen Programmiersprachen kann zu tieferem Verständnis seriöser Programmiersprachen sowie zur Verbesserung strukturellen Denkens führen. Abhängig vom verfolgten Konzept können esoterische Programmiersprachen Konzepte für Sprachdesign und/oder Systemdesign demonstrieren. Link: http://de.wikipedia.org/wiki/Esoterische_Programmiersprache

Brainfuck:

Brainfuck ist eine esoterische Programmiersprache, die vom Schweizer Urban Müller um 1993 entwickelt wurde. Die Sprache wird manchmal auch *Brainf*ck*, *Brainf**** oder *BF* genannt. Link: http://de.wikipedia.org/wiki/Brainfuck

Lexikalik von Brainfuck:

Brainfuck-Quelltext darf beliebige Zeichen enthalten.

Syntax von Brainfuck:

Brainfuck besitzt acht Befehle, jeweils bestehend aus einem einzigen Zeichen:

```
> < + - . , [ ]
```

Andere im Quelltext vorkommende Zeichen (z. B. Buchstaben, Zahlen, Leerzeichen, Zeilenumbrüche) werden ignoriert und können so als Quelltextkommentar verwendet werden. Folgende EBNF definiert die Syntax von Brainfuck:

```
<BF-Programm> ::= { <Befehl> }
<Befehl>      ::= „>“ | „<“ | „+“ | „-“ | „.“ | „,“ | <Schleife>
<Schleife>    ::= „[“ <BF-Programm> „]“
```

Semantik von Brainfuck:

Grundlage von Brainfuck ist ein Speicher und ein Speicherelementzeiger. Im Falle der PVL wird der Speicher durch ein Array namens `speicher` mit 100 Elementen (Zellen) vom Typ `int` gebildet. Der Speicherelementzeiger namens `zeiger` ist eine Variable vom Typ `int`, in der letztendlich Indizes auf den Speicher verwaltet werden. Bei Start eines Programms werden die einzelnen Speicherelemente sowie der Zeiger mit dem Wert 0 initialisiert.

Die Semantik von Brainfuck ist dann folgendermaßen definiert:

>	`++zeiger;`	Inkrementiert den Zeiger
<	`--zeiger`	Dekrementiert den Zeiger
+	`++(speicher[zeiger]);`	Inkrementiert den Wert des aktuellen Speicherelementes
-	`--(speicher[zeiger]);`	Dekrementiert den Wert des aktuellen Speicherelementes
.	`IO.print((char)speicher[zeiger]);`	Gibt den Wert des aktuellen Speicherelementes als ASCII-Zeichen auf die Standardausgabe aus
;	`speicher[zeiger] = IO.readChar();`	Liest ein Zeichen von der Standardeingabe und speichert dessen ASCII-Wert im aktuellen Speicherelement
[	`while (speicher[zeiger] != 0) {`	Springt hinter den passenden]-Befehl, wenn sich im aktuellen Speicherelement der Wert 0 befindet
]	`}`	Sprint hinter den passenden [-Befehl zurück, wenn sich im aktuellen Speicherelement ein Wert ungleich 0 befindet

Beispiel (aus Wikipedia):

Das folgende Brainfuck-Programm gibt „Hello World!“ und einen Zeilenumbruch aus.

```
++++++++++
[
 >+++++++>++++++++++>+++>+<<<<-
]                   Schleife zur Vorbereitung der Textausgabe
>++.                    Ausgabe von 'H'
>+.                     Ausgabe von 'e'
+++++++.                'l'
.                       'l'
+++.                    'o'
>++.                    Leerzeichen
<<+++++++++++++++.      'W'
>.                      'o'
+++.                    'r'
------.                 'l'
--------.               'd'
>+.                     '!'
>.
```

Zur besseren Lesbarkeit ist dieser Brainfuckcode auf mehrere Zeilen verteilt und kommentiert worden. Brainfuck ignoriert alle Zeichen, die keine Brainfuckbefehle sind.

Zunächst wird die erste (die „nullte“) Zelle des Speichers auf den Wert 10 gesetzt (speicher[0] = 10). Die Schleife am Anfang des Programms errechnet dann mit Hilfe dieser Zelle weitere Werte für die zweite, dritte, vierte und fünfte Zelle. Für die zweite Zelle wird der Wert 70 errechnet, welcher nahe dem ASCII-Wert des Buchstaben 'H' (ASCII-Wert 72) liegt. Die dritte Zelle erhält den Wert 100, nahe dem Buchstaben 'e' (ASCII-Wert 101), die vierte den Wert 30 nahe dem Wert für Leerzeichen (ASCII-Wert 32), die fünfte den Wert 10, welches dem ASCII-Zeichen „Line Feed“ entspricht und als Zeilenumbruch interpretiert wird.

Die Schleife errechnet die Werte, indem einfach auf die zu anfangs mit 0 initialisierten Zellen 10-mal 7, 10, 3 und 1 addiert wird. Nach jedem Schleifendurchlauf wird speicher[0] dabei um eins verringert, bis es den Wert 0 hat und die Schleife dadurch beendet wird.

Am Ende der Schleife hat das Datenfeld dann folgende Werte:

speicher[0] = 0; speicher [1] = 70; speicher [2] = 100;

speicher [3] = 30; speicher [4] = 10;

Als nächstes wird der Zeiger auf die zweite Zelle des Datenfelds (speicher[1]) positioniert und der Wert der Zelle um zwei erhöht. Damit hat es den Wert 72, welches dem ASCII-Wert des Zeichens 'H' entspricht. Dieses wird daraufhin ausgegeben. Nach demselben Schema werden die weiteren auszugebenden Buchstaben mit Hilfe der durch die Schleife initialisierten Werte, sowie der bereits verwendeten Werte, errechnet.

Aufgabe:

Entwickeln Sie ein Java-Programm, das einen Brainfuck-Interpreter implementiert. Konkret soll das Programm folgendes tun:

1. Der Benutzer wird nach dem Namen einer Datei gefragt, die ein Brainfuck-Programm enthält.
2. Die Datei wird eingelesen.
3. Der eingelesene Brainfuck-Quelltext wird zeichenweise interpretiert, d.h. Befehl für Befehl ausgeführt.

Wird im interpretierten Brainfuck-Programm ein Syntaxfehler entdeckt, wird die Ausführung des Programms unmittelbar mit einer entsprechenden Fehlermeldung beendet. Dasselbe trifft für Laufzeitfehler zu. Ein Laufzeitfehler kann auftreten, wenn auf eine nicht vorhandene Speicherzelle (bspw. speicher[-1]) zugegriffen werden soll.

Aufgabe 45:

(a) Schreiben Sie ein Programm, in dem ein int-Array erzeugt wird. Im Array soll das größte Element ermittelt und der entsprechende Wert auf den Bildschirm ausgegeben werden.

(b) Schreiben Sie eine Funktion, die ein int-Array als Parameter übergeben bekommt. Die Funktion soll das größte Element im Array ermitteln und liefern. Schreiben Sie ein Testprogramm für die Funktion, in dem die Funktion aufgerufen und der ermittelte Wert auf den Bildschirm ausgegeben wird.

(c) Schreiben Sie eine Funktion, die eine int-Matrix als Parameter übergeben bekommt. Die Funktion soll das größte Element der Matrix ermitteln und liefern. Schreiben Sie ein Testprogramm für die Funktion, in dem die Funktion aufgerufen und der ermittelte Wert auf den Bildschirm ausgegeben wird.

Aufgabe 46:

Teilaufgabe 1: Schreiben Sie eine Funktion, die ein int-Array als Parameter übergeben bekommt. Die Funktion soll die Reihenfolge der Werte umdrehen, Beispiel:

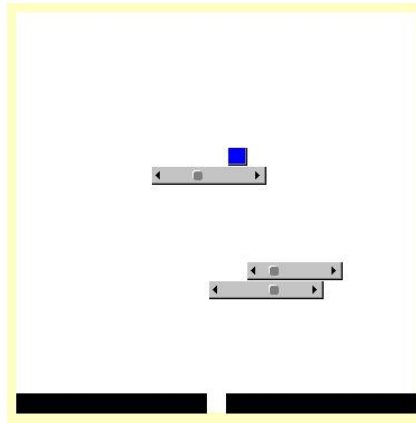
Array anfangs = {2, 4, 8, -3, 5}   Array hinterher = {5, -3, 8, 4, 2}

Teilaufgabe 2: Dieselbe Aufgabe. Nur diesmal soll die Funktion das übergebene Array nicht verändern, sondern das entsprechende Array als Funktionswert liefern.

Aufgabe 47:

Das „Balkenspiel“ ist ein Logikspiel für einen Spieler. Es geht darum, durch geschicktes Verschieben von Balken einen Ball in ein Zielfeld fallen zu lassen. Sie können das Spiel gerne mal online ausprobieren:

http://www.kleine-onlinespiele.de/Balken/balken.htm



Entwickeln Sie ein Java-Programm, das es einem Benutzer erlaubt, an einer Konsole das Balkenspiel zu spielen.

In unserem Fall ist die Ausgangssituation in einer Textdatei abgespeichert. Der Ball wird durch ein „*“, die Balken durch Ziffern („0“ bis „9“), der Grund durch ein „+“ und alle anderen Felder durch ein Leerzeichen repräsentiert. Die Größe des Spielfeldes ergibt sich aus der Anzahl an Reihen und Spalten der Datei. Sie können davon ausgehen, dass die Datei immer korrekte Spielfelder enthält und alle Reihen gleich viele Spalten besitzen. Obiges Spielfeld würde bspw. in etwa durch folgende Datei repräsentiert:

```

      *
    1111

       222
     3333

 ++++ +++++
```

Lesen Sie die Datei mit Hilfe der Funktion `readFileAsCharMatrix` ein, wobei *dateiname* ein String ist, der die Datei kennzeichnet:

```
char[][] spielbrett = IO.readFileAsCharMatrix(dateiname);
```

Der Programmablauf sieht folgendermaßen aus:

Der Benutzer wird zunächst nach dem Namen einer Datei gefragt, die eine Ausgangssituation enthält. Die Datei wird eingelesen. Bis zum Spielende wird dann der Benutzer in einer Schleife aufgefordert, einen Balken und eine Richtung anzugeben und der angegebene Balken wird um ein Feld in die angegebene Richtung verschoben, wobei gegebenenfalls der Ball mitverschoben wird und eventuell runter fällt. Das Spielende ist erreicht, wenn der Ball die Lücke am unteren Rand erreicht.

Hinweise: Balken und der Ball können beliebig weit aber nicht über den Rand hinaus verschoben werden. Sie müssen nicht kontrollieren, ob es in einer bestimmten Situation überhaupt noch möglich ist, zu gewinnen.

Orientieren Sie sich, was den Programmablauf als auch was die Ein- und Ausgaben angeht, an folgendem Beispiel (Eingaben stehen in <>):

```
     *
   1111

      222
    3333

++++ +++++

Balken wählen: <1>
Richtung wählen (l=links, r=rechts): <l>


     *
  1111

      222
    3333

++++ +++++

Balken wählen: <1>
Richtung wählen (l=links, r=rechts): <l>



 1111

     *222
    3333

++++ +++++

Balken wählen: <2>
Richtung wählen (l=links, r=rechts): <l>



 1111

    *222
    3333

++++ +++++

Balken wählen: <3>
Richtung wählen (l=links, r=rechts): <r>



 1111

     222
     3333

++++*+++++

Hurra! Geschafft!
```

Aufgabe 48:

Bei der in dieser Aufgabe betrachteten Variante des Hangman- bzw. Galgenmännchen-Spiels spielt der Computer gegen einen menschlichen Spieler. Der Computer wählt zunächst ein geheimes Wort aus. Der Spieler muss dieses Wort dann innerhalb von 10 Spielrunden erraten. In jeder Runde teilt der Spieler dem Computer zunächst ein Zeichen mit. Der Computer verrät darauf hin, ob und wenn ja an welcher Stelle bzw. an welchen Stellen das Zeichen im geheimen Wort vorkommt.

Implementieren Sie bitte genau folgenden Algorithmus. Halten Sie sich dabei auch an die Ausgaben im unten stehenden Beispiel:

- Der Computer wählt ein geheimes Wort aus.
- Es werden 10 Spielrunden gespielt (es sei denn, der Mensch hat vorher das Wort erraten). Der Ablauf jeder Spielrunde sieht so aus:
    - Der Computer gibt das bisher erratene Wort aus. Bisher nicht erratene Zeichen des geheimen Wortes werden dabei durch einen Unterstrich („_“) ersetzt.
    - Der Computer fordert den Mensch auf, ein Zeichen einzugeben.
    - Der Computer überprüft das Zeichen:
        - Hatte der Mensch das Zeichen vorher bereits schon mal eingegeben, wird er darauf hingewiesen („Dummkopf“).
        - Ist das Zeichen nicht im geheimen Wort enthalten, wird der Mensch darauf hingewiesen („Pech gehabt“).
        - Ansonsten wird das Zeichen als geraten vermerkt. Es wird weiterhin überprüft, ob damit das komplette geheime Wort erraten wurde. In diesem Fall wird der Mensch beglückwünscht und das Programm beendet.
- Sind die 10 Spielrunden vorbei und hat der Mensch das geheime Wort nicht erraten, wird eine entsprechende Meldung ausgegeben („Leider verloren“) und das Programm beendet.

Beispiel für einen möglichen Programmablauf. Benutzereingaben stehen in Klammern (<>). Das geheime Wort ist „Java“:

```
Runde 1. Bisher geraten: ____. Was wählst du für ein Zeichen?<a>
Runde 2. Bisher geraten: _a_a. Was wählst du für ein Zeichen?<k>
Pech gehabt! k kommt im Wort nicht vor!
Runde 3. Bisher geraten: _a_a. Was wählst du für ein Zeichen?<v>
Runde 4. Bisher geraten: _ava. Was wählst du für ein Zeichen?<k>
Dummkopf! k hast du schon mal getippt!
Runde 5. Bisher geraten: _ava. Was wählst du für ein Zeichen?<J>
Gratulation! Du hast das Wort 'Java' erraten!
```

Aufgabe 49:

Bei dieser Aufgabe sollen Sie ein Programm entwickeln, durch das zwei menschliche Spieler am Computer gegeneinander ein kleines Spiel namens *BreakThrough* spielen können.

Regeln:

BreakThrough ist ein Spiel, das von zwei Personen auf einem Spielbrett von n Reihen und m Spalten (n und m = 5, 6, ... oder 9) gespielt wird.

Die Spieler (A und B) haben jeweils 2 * m gleichartige Figuren in ihrer Spielerfarbe (auf dem Brett). Sie werden zu Beginn auf den beiden Reihen aufgestellt, die dem jeweiligen Spieler am nächsten sind (wie bei der Schach-Grundstellung).

Die Spieler ziehen abwechselnd, A beginnt. Wer am Zug ist, muss ziehen, Passen ist nicht erlaubt.

Ein Zug wird gemacht, indem man eine seiner Figuren auf das Feld direkt vor der Figur zieht oder auf ein Feld diagonal vor der Figur. Das Feld, auf das gezogen wird, muss leer oder vom Gegner besetzt sein. Im letzten Fall wird die gegnerische Figur durch den Zug geschlagen, d. h. vom Brett genommen. Ziehen und Schlagen geht immer nur ein Feld weit, man kann keine Felder überspringen.

Man gewinnt, indem man eine seiner Figuren auf die hinterste Reihe auf der gegnerischen Seite bringt, oder indem man alle gegnerischen Figuren schlägt.

Aufgabe:

Schreiben Sie ein Java-Programm, mit dessen Hilfe zwei menschliche Spieler das Spiel BreakThrough spielen können.

Hinweise:

Der generelle Spielablauf lässt sich folgendermaßen skizzieren:

- Aufbau des Ausgangsspielbretts
- Ausgabe des Spielbretts
- Spieler A beginnt
- Solange das Spiel nicht beendet ist, tue folgendes
    - Spielzug einlesen
    - Spielzug überprüfen
    - Falls Spielzug nicht korrekt ist, Spiel beenden und Sieger verkünden
    - Falls Spielzug korrekt ist,
        - Spielzug ausführen
        - Ausgabe des Spielbretts
        - Spielerwechsel
- Sieger verkünden

Beispiel für einen Programmablauf mit n = 7 und m = 8 (Benutzereingaben in grün):

```
  Spieler A
  0 1 2 3 4 5 6 7
0 + + + + + + + +
1 + + + + + + + +
2 . . . . . . . .
3 . . . . . . . .
4 . . . . . . . .
5 o o o o o o o o
```

```
6 o o o o o o o o
  Spieler B

Spieler A ist am Zug!
von Reihe: 1
von Spalte: 0
nach Reihe: 2
nach Spalte: 0

  Spieler A
  0 1 2 3 4 5 6 7
0 + + + + + + + +
1 . + + + + + + +
2 + . . . . . . .
3 . . . . . . . .
4 . . . . . . . .
5 o o o o o o o o
6 o o o o o o o o
  Spieler B

Spieler B ist am Zug!
von Reihe: 5
von Spalte: 0
nach Reihe: 4
nach Spalte: 1

  Spieler A
  0 1 2 3 4 5 6 7
0 + + + + + + + +
1 . + + + + + + +
2 + . . . . . . .
3 . . . . . . . .
4 . o . . . . . .
5 . o o o o o o o
6 o o o o o o o o
  Spieler B

Spieler A ist am Zug!
von Reihe: 2
von Spalte: 0
nach Reihe: 3
nach Spalte: 1

  Spieler A
  0 1 2 3 4 5 6 7
0 + + + + + + + +
1 . + + + + + + +
2 . . . . . . . .
3 . + . . . . . .
4 . o . . . . . .
5 . o o o o o o o
6 o o o o o o o o
  Spieler B

Spieler B ist am Zug!
von Reihe: 4
von Spalte: 1
nach Reihe: 3
nach Spalte: 1

  Spieler A
  0 1 2 3 4 5 6 7
0 + + + + + + + +
```

```
1 . + + + + + + +
2 . . . . . . . .
3 . o . . . . . .
4 . . . . . . . .
5 . o o o o o o o
6 o o o o o o o o
  Spieler B

...
```

Aufgabe 50:

Ein „Klumpen" in einem 1-dimensionalen Array ist eine Aufeinanderfolge von zwei oder mehr benachbarten Elementen mit demselben Wert.

Implementieren Sie eine Funktion, die die Anzahl an Klumpen in einem als Parameter übergebenen Array berechnet und als Funktionswert liefert.

Beispiele:

```
[1, 2, 2, 3, 4, 4]      → 2
[1, 1, 2, 1, 1]         → 2
[1, 1, 1, 1]            → 1
[1, 2, 3, 4, 5]         → 0
```

Aufgabe 51:

Dreidel ist ein jüdisches Glückspiel für Kinder mit N Spielern (N >= 2), bei dem um Bonbons gespielt wird. Zum Spiel benötigt wird der Dreidel, das ist ein Würfel mit vier Seiten. Auf den vier Seiten befinden sich die Werte 1, 2, 3 und 4. Es existiert ein Bonbon-Behälter (die „Kasse") mit anfangs 0 Bonbons. Die Spieler selbst haben anfangs ein Startkapital von jeweils M (M > 0) Bonbons. Die Spieler kommen abwechselnd an die Reihe und würfeln den Würfel. Je nachdem welche Seite nach oben zeigt, passiert folgendes:

1. Es passiert nichts.
2. Der Spieler, der gewürfelt hat, gewinnt den gesamten Kasseninhalt. Danach muss jeder Spieler eines seiner Bonbons in die Kasse legen.
3. Der Spieler, der gewürfelt hat, gewinnt die Hälfte (abgerundet) der Kasse.
4. Der Spieler, der gewürfelt hat, muss eines seiner Bonbons in die Kasse legen.

Sobald ein Spieler keine eigenen Bonbons mehr hat, scheidet er aus. Das Spiel endet, wenn keiner oder nur noch ein Spieler Bonbons besitzt.

Vorgabe:

Im Folgenden wird ein Programmrahmen vorgestellt, mit dem der Computer ein Dreidel-Spiel mit N=5 Spielern simuliert.

```
public class Dreidel {

     static final int START_KAPITAL = 10; // Bonbons für jeden Spieler

     static final int ANZAHL_SPIELER = 5; // Anzahl an Spielern
```

```
	// Hauptprogramm
	public static void main(String[] args) {
		// Initialisierung
		int[]  spieler  =  new  int[ANZAHL_SPIELER];  //  Bonbons  jedes
Spielers
		for (int i = 0; i < spieler.length; i++) {
			spieler[i] = START_KAPITAL;
		}

		// Spieldurchführung
		int aktSpieler = 0; // Index des aktuellen Spielers
		int kasse = 0; // Bonbons in der Kasse
		while (!spielende(spieler)) {
			int dreidel = 1 + (int) (Math.random() * 4); // wuerfeln
			kasse = spielzug(spieler, aktSpieler, kasse, dreidel);
			ausgeben(spieler, aktSpieler, dreidel, kasse);
			aktSpieler = naechsterSpieler(spieler, aktSpieler);
		}
	}



	// Ausgabe von aktuellen Spielinformationen
	private  static  void  ausgeben(int[]  spieler,  int  aktSpieler,  int
dreidel,
			int kasse) {
		System.out.println("Spielstand: ");
		System.out.println("Gewuerfelt: " + dreidel);
		System.out.println("Aktueller   Spieler   =   Spieler   "   +
aktSpieler);
		for (int i = 0; i < spieler.length; i++) {
			System.out.println("Spieler " + i + ": " + spieler[i] + "
Bonbons");
		}
		System.out.println("Kassenstand: " + kasse + " Bonbons");
		IO.readChar("<RETURN> drücken");

	}

	/**
	 * Ueberprueft das Spielende
	 *
	 * @param spieler
	 *            die Anzahl an Bonbons der einzelnen Spieler
	 *  @return  genau  dann  true,  wenn  kein  oder  nur  noch  ein  Spieler
Bonbons
	 *         besitzt
	 */
	static boolean spielende(int[] spieler) {
	}

	/**
	 *  ermittelt  den  naechsten  Spieler  gemaess  folgender  Regeln:  (1)  es
wird
	 * reihum gezogen (2) Spieler ohne Bonbons sind bereits ausgeschieden
	 *
	 * @param spieler
	 *            Anzahl an Bonbons der einzelnen Spieler
	 * @param aktSpieler
	 *            Array-Index des Spielers, der zuletzt an der Reihe war
```

```
     * @return Array-Index des Spielers, der als naechster an der Reihe
ist
     */
    static int naechsterSpieler(int[] spieler, int aktSpieler) {
    }

    /**
     * fuehrt einen Spielzug gemaess der Dreidel-Spielregeln aus
     *
     * @param spieler
     *            Anzahl an Bonbons der einzelnen Spieler
     * @param aktSpieler
     *            Array-Index des Spielers, der an der Reihe ist
     * @param kasse
     *            Anzahl an Bonbons in der Kasse (vor dem Spielzug)
     * @param dreidel
     *            der gewuerfelte Dreidel-Wert
     * @return Anzahl an Bonbons in der Kasse (nach dem Spielzug)
     */
    static int spielzug(int[] spieler, int aktSpieler, int kasse, int
dreidel) {
    }

}
```

Aufgabe:

Schauen Sie sich den Programmrahmen an und implementieren Sie die 3 fehlenden Methoden `spielende`, `naechsterSpieler` und `spielzug`. Am vorgegebenen Programmrahmen dürfen Sie dabei nichts ändern.

Aufgabe 52:

Aufgabe ist es, einen interessanten Algorithmus zu programmieren, der zur Ermittlung von Quadratzahlen benutzt werden kann: Ein Engelchen steht vor einem Haus mit N Türen, nummeriert von 1 bis N (Bsp. N = 10). Zu Beginn des Spiels sind alle Türen geschossen. Das Engelchen läuft nun am Haus entlang und öffnet jede zweite Tür. Ist es am Ende angelangt, geht es zurück zum Anfang. Jetzt geht es zu jeder dritten Tür: Sind diese offen, dann schließt es sie; sind diese geschlossen, dann öffnet es sie. Am Ende angelangt, geht es wieder zurück und kümmert sich dann um jede vierte Tür und ändert den Zustand analog. Das macht das Engelchen so lange, bis die Schrittweite gleich der Anzahl an Türen ist, d.h. im letzten Durchgang dreht es nur den Zustand der letzten Tür um.

Aufgabe: Implementieren Sie den Engelchen-Algorithmus. Repräsentieren Sie dabei den Zustand der N-Türen in einem entsprechend großen boolean-Array. Geben Sie den Anfangszustand und den Zustand der Türen nach jedem Durchgang aus. Ihr Programm sollte korrekt sein, wenn am Ende genau die Türen mit Quadratzahlnummern geschlossen sind. Orientieren Sie sich bei der Form der Ausgabe an folgendem Beispiel für N=10:

```
Die Tueren haben folgende Zustaende:
Tuer 1: geschlossen
Tuer 2: geschlossen
Tuer 3: geschlossen
Tuer 4: geschlossen
Tuer 5: geschlossen
Tuer 6: geschlossen
```

```
Tuer 7: geschlossen
Tuer 8: geschlossen
Tuer 9: geschlossen
Tuer 10: geschlossen

...

Die Tueren haben folgende Zustaende:
Tuer 1: geschlossen
Tuer 2: offen
Tuer 3: offen
Tuer 4: geschlossen
Tuer 5: offen
Tuer 6: offen
Tuer 7: offen
Tuer 8: offen
Tuer 9: geschlossen
Tuer 10: offen
```

Aufgabe 53:

Rechtakien ist ein rechteckiges Gebiet, das sich aus einzelnen Kacheln zusammensetzt. Die Kachel in der Nordwest-Ecke hat die Zeilennummer 0 und die Spaltennummer 0.

Das Wetter in Rechtakien wird durch Wolken bestimmt, die genau eine Kachel groß sind. Es gibt zwei Typen von Wolken: Nord-Süd-Wolken und West-Ost-Wolken. Die Wolken rücken getaktet über Rechtakien vor, und zwar Nord-Süd-Wolken von Norden nach Süden 2 Kacheln pro Takt und West-Ost-Wolken von Westen nach Osten 3 Kacheln pro Takt. Am unteren und rechten Rand verlassen die Wolken Rechtakien. Es regnet auf genau den Kacheln, wo sich nach einem Vorrücken sowohl eine Nord-Süd- als auch eine West-Ost-Wolke befinden. Wolken, aus denen es regnet, lösen sich auf.

Bei der in der folgenden Abbildung dargestellten Ausgangssituation (Nord-Süd-Wolken haben grauen Hintergrund, West-Ost-Wolken haben schwarzen Hintergrund) regnet es genau auf den Kacheln(x=14/y=7) und (11/9).

Ausgangspunkt der Wettervorhersage ist die aktuelle Wetterlage. Sie ist bezüglich dieser Aufgabe in einer Datei gespeichert. Nord-Süd-Wolken werden dabei durch ein großes ‚W' und West-Ost-Wolken durch ein kleines ‚w' gekennzeichnet. Kacheln ohne Wolken sind durch einen Punkt (‚.') gekennzeichnet. Ausgehend von einer solchen Wetterlage lässt sich dann ermitteln, wo es in Rechtakien regnen wird, wobei vorausgesetzt wird, dass keine neuen Wolken entstehen. Die Simulation ist beendet, wenn sich keine Wolke mehr über Rechtakien befindet.

Aufgabe: Schreiben Sie ein Programm, das die aktuelle Wetterlage in Rechtakien aus einer Datei einliest und die Wettervorhersage ausgibt. Orientieren Sie sich bei der Form der Ausgabe an folgendem Beispiel. Die Regenkacheln werden in der Form <Spaltennummer> / <Zeilennummer> ausgegeben:

Beispiel: Gegeben sei folgender Inhalt in der Ausgangsdatei (entspricht obiger Abbildung)

```
........................
........................
.......WW...............
...........W............
........................
..............W.........
........................
...........w............
........................
..w..............w......
..w..............w......
.....w............w.....
.................w......
............W...........
........................
........................
```

Dann produziert das Programm die Ausgabe:

```
Es regnet auf folgenden Kacheln:
14/7
11/9
```

Hinweise:

- Sie können davon ausgehen, dass eine eingelesene Datei eine korrekte Wetterausgangslage in Rechtakien beschreibt, also rechteckig groß ist und nur gültige Buchstaben enthält.

Aufgabe 54:

Gegeben ein 2-dimensionales Array (Matrix) *A* mit int-Werten. Eine Zahl *x* heißt **B-Zahl** von *A*, wenn *x* Element von A ist und es gilt: *x* ist die kleinste Zahl in ihrer Spalte und die größte Zahl in ihrer Zeile.

Beispiel:

```
1 3 8 4 6 6
4 9 7 9 1 5
2 2 6 5 4 1
3 3 7 3 3 4
1 2 6 7 8 9
```

Die Zahl 6 in der dritten Spalte und dritten Zeile ist die einzige B-Zahl der gegebenen Matrix.

Aufgabe: Implementieren Sie eine boolesche Funktion namens `enthaeltBZahl` mit einem 2-dimensionalen int-Array (Matrix) als Parameter, die genau dann `true` liefert, wenn die übergebene Matrix mindestens eine B-Zahl enthält. Kommentieren Sie kurz ihren gewählten Algorithmus.

Hinweise:

- Sie können davon ausgehen, dass alle Zeilen der übergebenen Matrix gleich viele Spalten aufweisen.
- Die Performanz ihres gewählten Algorithmus spielt keine Rolle.

Aufgabe 55:

Unter einer symmetrischen Matrix versteht man in der linearen Algebra eine quadratische 2-dimensionale Matrix, die symmetrisch zur Hauptdiagonalen ist. Als Diagonale werden dabei diejenigen Elemente bezeichnet, die in einer quadratischen Matrix auf einer gedachten Linie, die von oben schräg nach unten geht, liegen. Die Hauptdiagonale ist die Diagonale, die von oben links nach unten rechts verläuft.

Beispiele für symmetrische 2-dimensionale Matrizen:

```
1 3 8 4 6        2 3 4 5      1 2       5
3 9 7 9 1        3 1 6 9      2 1
8 7 6 5 4        4 6 3 2
4 9 5 3 3        5 9 2 5
6 1 4 3 8
```

Aufgabe: Implementieren Sie eine boolesche Funktion

```
static boolean istSymmetrisch(int[][] matrix)
```

die eine quadratische Matrix übergeben bekommt und genau dann `true` liefert, wenn die Matrix symmetrisch ist. Die Funktion soll dabei möglichst effizient sein, d.h. nicht mehr Vergleiche durchführen, als unbedingt notwendig.

Aufgabe 56:

Bei einem **Survo-Rätsel** ist es die Aufgabe, die Zahlen 1, 2, ..., m*n so in eine m × n - Matrix (m > 0, n > 0) einzutragen, dass jede von diesen Zahlen nur einmal verwendet wird und dass die Zeilen- und Spaltensummen den Zahlen entsprechen, welche unterhalb und auf der rechten Seite der Matrix gegeben sind. (siehe http://de.wikipedia.org/wiki/Survo_Puzzle).

Beispiel für ein gelöstes Survo-Rätsel: (m = 3, n = 4)

```
12  6  2 10    30
 8  1  5  4    18
 7  9  3 11    30

27 16 10 25
```

Stellen Sie sich nun vor, Sie schreiben ein Java-Programm zum Lösen von Survo-Rätseln. Darin benötigen Sie eine Funktion *istSurvoLoesung*, die überprüft, ob zu einem Survo-Rätsel eine gültige Lösung gefunden wurde. Diese Funktion habe folgende Gestalt:

```
static boolean istSurvoLoesung(int[][] matrix,
                               int[] reihenSummen,
                               int[] spaltenSummen)
```

Der erste Parameter ist eine m*n-Matrix, der zweite Parameter ein Array der Länge m mit den Summenwerten der einzelnen Reihen und der dritte Parameter ein Array der Länge n mit den Summenwerten der einzelnen Spalten. Die Funktion soll genau dann *true* liefern, wenn die Parameter ein gelöstes Survo-Rätsel enthalten, d.h. wenn gilt:

- Jede Zahl zwischen 1 und m*n ist genau einmal in der Matrix *matrix* enthalten
- die Summe der Zahlen aller Reihen i der Matrix ist gleich *reihenSummen[i]*
- die Summe der Zahlen aller Spalten j der Matrix ist gleich *spaltenSummen[j]*

Das folgende Programm sollte also *true* auf die Konsole ausgeben:

```
int[][] matrix = { { 12, 6, 2, 10 },
                   {  8, 1, 5,  4 },
                   {  7, 9, 3, 11 } };
int[] reihenSummen = { 30, 18, 30 };
int[] spaltenSummen = { 27, 16, 10, 25 };
System.out.println(istSurvoLoesung(matrix, reihenSummen,
                                   spaltenSummen));
```

Aufgabe:

Gegeben ist folgende Teilimplementierung der Funktion:

```
// ueberprueft, ob die uebergebenen Zahlen eine korrekte Loesung eines
// Survo-Raetsels darstellen
// Voraussetzung: Die Parameter repraesentieren ein korrektes
// Survo-Raetsel,
// d.h. die Groessen der uebergebenen Arrays sind korrekt

static boolean istSurvoLoesung(int[][] matrix, int[] reihenSummen,
                               int[] spaltenSummen) {
           if (!checkZahlen(matrix)) {
                 return false;
```

```
        }
        if (!checkReihenSummen(matrix, reihenSummen)) {
              return false;
        }
        if (!checkSpaltenSummen(matrix, spaltenSummen)) {
              return false;
        }
        return true;
  }

  // liefert genau dann true, wenn gilt: Habe die
  // Matrix matrix m Reihen und n
  // Spalten. Dann ist jede Zahl zwischen 1 und m*n genau
  // einmal in matrix
  // enthalten.
  static boolean checkZahlen(int[][] matrix) {
        boolean[] zahlen =
          new boolean[matrix.length * matrix[0].length];
        for (int r = 0; r < matrix.length; r++) {
              for (int s = 0; s < matrix[r].length; s++) {
                    if (matrix[r][s] < 1
                                ||
                      matrix[r][s] > matrix.length * matrix[0].length) {
                          return false;
                    }
                    if (zahlen[matrix[r][s]]) {
                          return false;
                    }
                    zahlen[matrix[r][s]] = true;
              }
        }
        return true;
  }

  // liefert genau dann true, wenn fuer alle Reihen
  // i (0..AnzahlReihen-1) der
  // Matrix gilt:
  // die Summe der Zahlen der Reihe i der Matrix ist
  // gleich reihenSummen[i]
  static boolean checkReihenSummen(int[][] matrix, int[] reihenSummen)
  {
  }

  // liefert genau dann true, wenn fuer alle Spalten
  // i (0..AnzahlSpalten-1)
  // der Matrix gilt:
  // die Summe der Zahlen der Spalte i der Matrix ist
  // gleich spaltenSummen[i]
  static boolean checkSpaltenSummen(int[][] matrix,
                                    int[] spaltenSummen) {
  }
```

Teilaufgabe 1:

Implementieren Sie die fehlende Funktionen *checkReihenSummen* und *checkSpaltenSummen*.

Teilaufgabe 2:

Die Funktion *checkZahlen* enthält zwei Fehler, die zum Werfen einer *ArrayIndexOutOfBoundsException* führen können. Finden und korrigieren Sie die entsprechenden Fehler.

Aufgabe 57:

Ein Ausdruck kann aus Zahlen und Operatoren bestehen. Er berechnet und liefert einen Wert. In dieser Aufgabe wird ein Ausdruck durch ein char-Array dargestellt, wobei als gültige char-Zeichen die Ziffern (0, 1, …, 9) sowie die zweistelligen Operatoren +, -, *, / und % gelten. Die Operatoren haben dabei alle dieselbe Priorität und sind linksassoziativ. Zahlen setzen sich aus mehreren aufeinander folgenden Ziffern zusammen.

Ein „gültiger Ausdruck" im Sinne dieser Aufgabe beginnt immer mit einer Zahl und endet mit einer Zahl. Zwischen zwei Zahlen steht immer genau ein Operator. Ein gültiger Ausdruck enthält mindestens einen Operator.

Teilaufgabe 1: Implementieren Sie in Java eine Seiteneffekt-freie Funktion

```
static int berechne(char[] ausdruck)
```

der ein gültiger Ausdruck als char-Array übergeben wird und die daraufhin den Wert des Ausdrucks berechnet und als Funktionswert liefert. Sie können davon ausgehen, dass gültige Ausdrücke übergeben werden (bspw. keine Zahlen, die nicht mehr als int-Wert repräsentierbar sind) und dass eine fehlerfreie Auswertung der Ausdrücke möglich ist (bspw. keine Division durch 0).

Teilaufgabe 2: Implementieren Sie in Java eine Prozedur

```
static void print(char[] ausdruck)
```

die den Ausdruck, gefolgt von einem =-Zeichen, gefolgt von dem berechneten Wert des Ausdrucks auf die Console schreibt. Nutzen Sie zur Berechnung des Wertes dazu die in Teilaufgabe 1 implementierte Funktion.

Test:

Das folgende kleine Testprogramm sollte die Ausgabe

```
113+23=136
28-10/5*7%9=3
```

erzeugen:

```
public static void main(String[] args) {
  char[] ausdruck1 = { '1', '1', '3', '+', '2', '3' };
  print(ausdruck1);
  char[] ausdruck2 = { '2', '8', '-', '1', '0', '/', '5', '*', '7', '%',
'9' };
  print(ausdruck2);
}
```

Aufgabe 58:

In dieser Aufgabe geht es um die Verschlüsselung von Texten. Klartexte über einem Klartextalphabet werden durch die Anwendung von Verschlüsselungsalgorithmen in Geheimtexte über einem Geheimtextalphabet überführt. Verschlüsselungsverfahren sollen gewährleisten, dass nur Befugte bestimmte Botschaften lesen können.

Die *Skytale* ist das älteste bekannte militärische Verschlüsselungsverfahren (aus Wikipedia). Von den Spartanern wurden bereits vor mehr als 2500 Jahren geheime Botschaften nicht im Klartext übermittelt. Zur Verschlüsselung diente ein (Holz-)Stab mit einem bestimmten Durchmesser (Skytale).

Um eine Nachricht zu verfassen, wickelte der Absender ein Pergamentband oder einen Streifen Leder wendelförmig um die Skytale, schrieb die Botschaft längs des Stabs auf das Band und wickelte es dann ab. Das Band ohne den Stab wird dem Empfänger überbracht. Fällt das Band in die falschen Hände, so kann die Nachricht nicht gelesen werden, da die Buchstaben scheinbar willkürlich auf dem Band angeordnet sind. Der richtige Empfänger des Bandes konnte die Botschaft mit einer identischen Skytale (einem Stab mit dem gleichen Durchmesser) lesen. Der Durchmesser des Stabes ist somit der geheime Schlüssel bei diesem Verschlüsselungsverfahren.

Die Implementierung des Skytale-Verfahrens wird an einem Beispiel erläutert:

Klartext: **DIESERKLARTEXTISTJETZTZUVERSCHLUESSELN**

Schlüssel **n**: **3**

Zur **Verschlüsselung** wird der Klartext zeilenweise in eine Matrix (zweidimensionales Array) mit n Spalten eingetragen. In übrig bleibende Felder wird ein Leerzeichen eingetragen.

```
DIE
SER
KLA
RTE
XTI
STJ
ETZ
TZU
VER
SCH
LUE
SSE
LN
```

Der Geheimtext ergibt sich durch das spaltenweise Lesen der Matrix. Hier:

```
DSKRXSETVSLSLIELTTTTZECUSNERAEIJZURHEE
```

Zur **Entschlüsselung** wird der Geheimtext spaltenweise in eine Matrix mit n Spalten eingetragen. Der Klartext ergibt sich durch das zeilenweise Lesen der Matrix.

Aufgabe:

Definieren und implementieren Sie folgende zwei Funktionen:

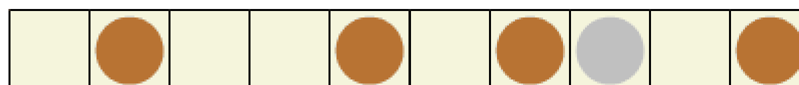
```
/**
 *  verschluesselt  den  uebergebenen  Klartext  mit  dem  uebergebenen
Schluessel
 *
 * @param klartext
 *            der Klartext
 * @param schluessel
 *            der Schluessel
 * @return der erzeugte Geheimtext
 */
static String verschluesseln(String klartext, int schluessel);

/**
 * entschluesselt den uebergebenen Geheimtext mit dem uebergebenen
 * Schluessel
 *
 * @param geheimtext
 *            der Geheimtext
 * @param schluessel
 *            der Schluessel
 *
 * @return der entschluesselte Text
 */
static String entschluesseln(String geheimtext, int schluessel);
```

Aufgabe 59:

Das "Silver-Dollar-Game" ist ein Spiel für zwei Spieler. Es wird auf einem Spielbrett gespielt, das aus einer Reihe an n Feldern besteht. Auf jedem der Felder liegt anfangs entweder kein oder ein Dollar. Genau ein Dollar ist dabei ein silberner Dollar. Die anderen sind golden.

Start



Die beiden Spieler absolvieren immer abwechselnd Spielzüge. Es besteht Zugzwang. Dabei gibt es zwei Typen von Spielzügen:

- R-Spielzug: Der Spieler wählt ein Feld aus, auf dem sich ein Dollar befindet. Er verschiebt dann den Dollar um ein oder mehrere Felder nach links. Dabei darf der Dollar jedoch nicht über andere Dollars hinweg verschoben oder außerhalb des Spielbrettes platziert werden.

- S-Spielzug: Der Spieler entfernt den am weitesten links platzierten Dollar vom Spielbrett.

Das Spiel ist beendet, wenn ein Spieler einen fehlerhaften Zug durchführt oder wenn der silberne Dollar vom Spielbrett entfernt wurde. Gewinner ist im ersten Fall der Gegenspieler und im zweiten Fall der Spieler, der den silbernen Dollar entfernt hat.

Aufgabe: Implementieren Sie ein (imperatives) Java-Programm, mit dem zwei menschliche Spieler auf der Konsole gegeneinander das Silver-Dollar-Game spielen können.

Das anfängliche Spielbrett kann dabei im Sourcecode in der folgenden Form fest vorgegeben werden. Ihr Programm sollte prinzipiell aber auch mit einer anderen Anzahl an Feldern und einer anderen anfänglichen Belegung mit Dollars funktionieren:

```
char[] brett = { '.','1','.','.','1','.','1','2','.','1'};
```

Dabei bedeuten:

`'.'`: leeres Feld

`'1'`: goldener Dollar

`'2'`: silberner Dollar

Orientieren Sie sich bezüglich des generellen Programmablaufes und der Ein- und Ausgaben an folgendem beispielhaften Ablauf der Musterlösung (Benutzereingaben in <>):

```
.1..1.12.1
Spieler 1 ist am Zug!
R- oder S-Spielzug? (R/S)<S>
....1.12.1
Spieler 2 ist am Zug!
R- oder S-Spielzug? (R/S)<R>
Von Feldindex?<4>

Nach Feldindex?<0>
1.....12.1
Spieler 1 ist am Zug!
R- oder S-Spielzug? (R/S)<S>
......12.1
Spieler 2 ist am Zug!
R- oder S-Spielzug? (R/S)<R>
Von Feldindex?<6>
Nach Feldindex?<2>
..1....2.1
Spieler 1 ist am Zug!
R- oder S-Spielzug? (R/S)<S>
.......2.1
Spieler 2 ist am Zug!
R- oder S-Spielzug? (R/S)<S>
```

```
.........1
Spielende! Spieler 2 hat gewonnen!
```

Aufgabe 60:

Das Spiel *Neighborfields* ist ein Spiel für 2 Spieler: Spieler 1 und Spieler 2. Spielfeld ist eine Menge von N (N >= 1) Feldern in einer Reihe. Die Felder sind anfangs leer. Die Spieler führen abwechselnd Spielzüge aus. Es besteht Zugzwang. Spieler 1 beginnt. In jedem Spielzug wählt der Spieler, der an der Reihe ist, zwei nebeneinander liegende leere Felder aus und markiert sie mit seiner Nummer. Der erste Spieler, der keinen Spielzug mehr machen kann, verliert unmittelbar.

Aufgabe:

Implementieren Sie ein imperatives Java-Programm, mit dem zwei menschliche Spieler an einer Console gegeneinander das Spiel *Neighborfields* spielen können. Beachten Sie dabei folgendes:

- Für Benutzereingaben bitte die bekannte Klasse *IO* benutzen.
- Die Menge an Feldern N wird anfangs eingelesen.
- Benutzereingaben (sei es beim Einlesen von N oder beim Einlesen eines Spielzugs) werden vom Programm auf Gültigkeit überprüft. Bei einer fehlerhaften Eingabe weist das Programm darauf hin und fordert den Benutzer solange zur Wiederholung der Eingabe hin, bis die Eingabe gültig ist.

Orientieren Sie sich bezüglich des generellen Programmablaufes und der Ein- und Ausgaben an folgendem beispielhaften Ablauf der Musterlösung (Benutzereingaben in <>):

```
Anzahl an Feldern (> 0)? <7>
Feld: .......
Spieler 1 ist am Zug!
Index des linken Feldes? <1>
Feld: .11....
Spieler 2 ist am Zug!
Index des linken Feldes? <6>
Ungueltiges Feld! Index des linken Feldes? <5>
Feld: .11..22
Spieler 1 ist am Zug!
Index des linken Feldes? <2>
Ungueltiges Feld! Index des linken Feldes? <3>
Feld: .111122
Spieler 2 hat verloren!
```

Aufgabe 61:

Implementieren Sie eine Funktion

```
static void reihenTausch(int[][] matrix)
```

der eine beliebige 2-dimensionale int-Matrix übergeben wird und die die Reihen der Matrix umsortiert. Das folgende Testprogramm

```java
public class UE7Aufgabe61 {

	// Randfaelle werden nicht betrachtet!
	public static void main(String[] args) {
		int[][] matrix = { {1, 2 },{2, 4, 3, 9 },{3, 5, 6 }, {2} };
		print(matrix);
		reihenTausch(matrix);
		print(matrix);
	}

	static void reihenTausch(int[][] matrix) {
		// todo
	}

	static void print(int[][] matrix) {
		for (int r = 0; r < matrix.length; r++) {
			for (int s = 0; s < matrix[r].length; s++) {
				System.out.print(matrix[r][s] + " ");
			}
			System.out.println();
		}
		System.out.println();
	}

}
```

soll bspw. folgende Ausgabe produzieren:

```
1 2
2 4 3 9
3 5 6
2

2
3 5 6
2 4 3 9
1 2
```

Aufgabe 62:

Implementieren Sie folgende Funktion:

`static int anzahlElemente(int[][] matrix)`

Die Funktion soll die Anzahl an Elementen der als Parameter übergebenen Matrix berechnen und liefern. Wichtig: Berücksichtigen Sie bei der Berechnung alle möglichen Sonderfälle in Bezug auf den aktuellen Parameter!

Beispiel: Die Matrix `{{1, 2, 3},{1, 2, 3}}` besitzt 6 Elemente.

Aufgabe 63:

Implementieren Sie eine Methode

```java
static int[][] transponieren(int[][] matrix)
```

die die *Transponierte Matrix* der als Parameter übergebenen Matrix liefert. Sie können davon ausgehen, dass die übergebene Matrix eine reguläre n * m Matrix (n > 0, m > 0; ohne Sonderfälle!) ist

Die Transponierte Matrix T einer n * m Matrix M ist dabei eine m * n Matrix, bei der die erste (zweite, …) Spalte von T der ersten (zweiten, …) Zeile von M entspricht. M wird sozusagen an ihrer Hauptdiagonalen gespiegelt.

Beispiele:

```
M = { { 1, 2 }, { 3, 4 }, { 5, 6 } }     → T = { {1, 3, 5}, {2, 4, 6} }
M = { {1, 2, 3}, {4, 5, 6} }             → T = { {1, 4}, {2, 5}, {3, 6} }
```

Aufgabe 64:

Sowing ist ein Brettspiel für 2 Spieler: Spieler 1 und Spieler 2. Gespielt wird auf einem Brett mit einer Reihe von N Mulden (hier N = 12). In jeder Mulde liegen anfangs M Samen (hier M = 3). Spieler 1 spielt von links nach rechts, Spieler 2 spielt von rechts nach links. Beide Spieler ziehen abwechselnd. Es besteht Zugzwang. In jedem Zug muss ein Spieler den gesamten Inhalt einer nicht leeren Mulde in seine Zugrichtung auf die darauf folgenden Mulden verteilen. Dabei wird in jede Mulde ein Samen gelegt, bis alle Samen verteilt sind. Der Inhalt einer Mulde darf nur dann verteilt werden, wenn genug Mulden in Zugrichtung liegen und der letzte Samen in eine nicht leere Mulde fallen würde. Der Spieler, der als erster keinen Zug mehr machen kann, verliert. Ein Unentschieden ist nicht möglich.

Implementieren Sie ein imperatives Java-Programm, mit dem zwei menschliche Spieler an einer Console gegeneinander das Spiel *Sowing* spielen können. Orientieren Sie sich bezüglich des generellen Programmablaufes und der Ein- und Ausgaben an folgendem beispielhaften Ablauf der Musterlösung (Benutzereingaben in <>):

```
  0  1  2  3  4  5  6  7  8  9 10 11
  3  3  3  3  3  3  3  3  3  3  3  3
Spieler 1 ist am Zug!
Muldennummer: <2>
  0  1  2  3  4  5  6  7  8  9 10 11
  3  3  0  4  4  4  3  3  3  3  3  3
Spieler 2 ist am Zug!
Muldennummer: <3>
Ungueltig! Muldennummer: <4>
  0  1  2  3  4  5  6  7  8  9 10 11
  4  4  1  5  0  4  3  3  3  3  3  3
Spieler 1 ist am Zug!
Muldennummer: <0>
Ungueltig! Muldennummer: <13>
Ungueltig! Muldennummer: <7>
  0  1  2  3  4  5  6  7  8  9 10 11
  4  4  1  5  0  4  3  0  4  4  4  3
...
```

Aufgabe 65:

Implementieren Sie eine Prozedur **`fuelleMatrixAlternierend`**, die als Parameter eine Referenz auf ein 2-dimensionales Array (Matrix) mit int-Werten enthält. Von „oben“ nach „unten“ sollen die einzelnen Reihen der Matrix durchlaufen werden und zwar Reihen mit einem geraden Index von „links“ nach „rechts“ und Reihen mit einem ungeraden Index von „rechts“ nach „links“. Beginnend mit dem Wert 1 soll den jeweiligen Elementen dabei ein jeweils um 1 erhöhter Wert zugewiesen werden.

Beispiel für eine 4*5-Matrix:

```
 1  2  3  4  5
10  9  8  7  6
11 12 13 14 15
20 19 18 17 16
```

Achtung: Der der Prozedur übergebene Parameter kann eine beliebige Referenz auf ein beliebiges 2-dimensionales Array mit int-Werten sein. Beachten und berücksichtigen Sie alle möglichen Sonderfälle!

Aufgabe 66:

Implementieren Sie eine Funktion **`getOddValues`**. Die Funktion bekommt eine Referenz auf ein Array mit `int`-Werten als Parameter übergeben. Sie soll eine Referenz auf ein innerhalb der Funktion neu erzeugtes Array mit `int`-Werten zurückliefern, das genau die ungeraden Werte des Parameter-Arrays in umgekehrter Reihenfolge enthält.

Schreiben Sie weiterhin ein kleines Testprogramm, das aus drei Teilen besteht:

- Definition und Anlegen eines Arrays `values` (z.B. `int[] values = {-3, 4, 8, -11, 0, 7};`)
- Aufruf der Funktion `getOddValues` mit dem Array `values` als Parameter
- Ausgabe aller Werte des zurückgelieferten Arrays auf die Konsole

Beispiel:

```
übergebenes Array = { -3, 4, 8, -11, 0, 7}

geliefertes Array = { 7, -11, -3}
```

Aufgabe 67:

Bei einer Variante des aus der Vorlesung bekannten Nim-Spiels gibt es n Mulden mit jeweils n Körnern ($n > 2$). Zwei Spieler nehmen abwechselnd Körner aus einer der Mulden weg. Wie viele sie nehmen, spielt keine Rolle; es muss mindestens ein Korn sein und es dürfen bei einem Zug nur Körner einer einzigen Mulde genommen werden. Derjenige Spieler, der den letzten Zug macht, also die letzten Körner wegnimmt, gewinnt, der andere Spieler verliert.

Aufgabe: Implementieren Sie diese Variante des Nim-Spiels in Java, so dass zwei menschliche Spieler gegeneinander an einer Konsole antreten können. Eingaben eines Spielers soll das Programm kontrollieren und bei nicht regelkonformen

Eingaben soll es den Spieler (so früh wie möglich) zu einer korrekten Eingabe auffordern. Den Wert N können Sie im Programm als Konstante festlegen (`final int n = 5;`)

Orientieren Sie sich was Eingaben und Ausgaben angeht an folgendem beispielhaften Ablauf des Programms (n = 5; Benutzereingaben in <>):

```
Muldenindex: 0 1 2 3 4
Koerner:     5 5 5 5 5

Spieler 1 ist am Zug!
Index der Mulde: <1>
Anzahl der Koerner: <5>
Muldenindex: 0 1 2 3 4
Koerner:     5 0 5 5 5

Spieler 2 ist am Zug!
Index der Mulde: <1>
Fehlerhafte Eingabe! Index der Mulde: <4>
Anzahl der Koerner: <6>
Fehlerhafte Eingabe! Anzahl der Koerner: <2>
Muldenindex: 0 1 2 3 4
Koerner:     5 0 5 5 3

...

Muldenindex: 0 1 2 3 4
Koerner:     0 0 0 0 3

Spieler 2 ist am Zug!
Index der Mulde: <4>
Anzahl der Koerner: <3>
Muldenindex: 0 1 2 3 4
Koerner:     0 0 0 0 0

Spieler 2 hat gewonnen!
Spieler 1 hat verloren!
```

Aufgabe 68:

Spielidee des Spieles *Platzda*: Auf einem Spielplatz liegt ein Baumstamm, der zum Spielen und Balancieren einlädt. Alle Kinder möchten darauf spielen, aber leider ist der Stamm nicht groß genug. Die ersten drei Kinder steigen auf den Stamm und spielen dort. Sobald das nächste Kind von links auf den Stamm steigen möchte, schiebt es alle Kinder etwas weiter nach rechts. Das Kind, das sich ganz rechts auf dem Stamm befindet, fällt hinunter. Die Spieler des Spiels Platzda sollen bestimmen, welches Kind als nächstes vom Baumstamm fällt.

Spielregeln und -ablauf: Anfangs wird gefragt, wie groß der Baumstamm ist, d.h. wie viele Kinder N gleichzeitig auf dem Stamm stehen können. Die Kinder werden durch Namen (**`String`**) repräsentiert. Anfangs stehen N Kinder mit dem Namen „**`niemand`**" auf dem Baumstamm. Dann spielen 2 Spieler gegeneinander. Sie sind abwechselnd an der Reihe. Es besteht Zugzwang. Jeder Spielzug besteht aus zwei Teilen. Im ersten Teil muss der Spieler den Namen des Kindes eingeben, dass als nächstes rechts vom Baumstamm fällt. Im zweiten Teil muss er dann einen beliebigen Namen eines Kindes eingeben, das als nächstes links auf den

Baumstamm klettert. Das Spiel ist unmittelbar beendet, wenn ein Spieler im ersten Teil einen falschen Namen eingibt.

Implementieren Sie ein imperatives Java-Programm, mit dem zwei menschliche Spieler gegeneinander an einer Konsole das Spiel *Platzda* spielen können. Orientieren Sie sich, auch was die Ausgaben angeht, an folgendem Beispielablauf meiner Musterlösung (Benutzereingaben in Klammern <>):

```
Baumstammgroesse (> 2): 3
Spieler 1! Wer fällt vom Stamm? <niemand>
Spieler 1! Richtig! Neuer Name: <otto>
Spieler 2! Wer fällt vom Stamm? <niemand>
Spieler 2! Richtig! Neuer Name: <karl>
Spieler 1! Wer fällt vom Stamm? <niemand>
Spieler 1! Richtig! Neuer Name: <maria>
Spieler 2! Wer fällt vom Stamm? <otto>
Spieler 2! Richtig! Neuer Name: <kai>
Spieler 1! Wer fällt vom Stamm? <karl>
Spieler 1! Richtig! Neuer Name: <sarah>
Spieler 2! Wer fällt vom Stamm? <kai>
Falsch! maria faellt vom Stamm!
Spieler 2 hat verloren!
```

Aufgabe 69:

Zelluläre Automaten bestehen aus einer Menge an Zellen, die jeweils zwei mögliche Zustände annehmen können: tot oder lebendig. Bei dieser Aufgabe geht es um eindimensionale zelluläre Automaten, d.h. die Zellen sind in einem ein-dimensionalen Array angeordnet und haben jeweils zwei Nachbarzellen. Nur die beiden äußeren Zellen haben jeweils nur eine Nachbarzelle.

Eine Generation von Zellen zu einem Zeitpunkt A wird durch eine Zustandsübergangsfunktion zu einer Nachfolge-Generation zum Zeitpunkt A+1 überführt. Der Zustand jeder einzelnen Zelle in der Nachfolge-Generation wird dabei auf der Grundlage des Zustands der Zelle sowie der Zustände ihrer Nachbarzellen in der aktuellen Generation überführt.

Die dieser Aufgabe zugrundeliegende Zustandsübergangsfunktion lässt sich folgendermaßen beschreiben:

- Wenn eine Zelle keinen lebendigen Nachbarn hat, ist ihr neuer Zustand tot.
- Wenn eine Zelle genau einen lebendigen Nachbarn hat, bleibt ihr Zustand erhalten.
- Wenn eine Zelle zwei lebendige Nachbarn hat, ändert sie ihren Zustand von tot auf lebendig bzw. von lebendig auf tot.

Eine Simulation eines zellulären Automaten besteht aus einer gegebenen Start-Generation von Zellen. Ausgehend von dieser wird die Zustandsübergangsfunktion eine gewisse Anzahl mal ausgeführt. Bei der Ausgabe einer Generation auf die Konsole wird eine tote Zelle durch einen Unterstrich „_“ und eine lebendige Zelle durch ein „#“-Zeichen repräsentiert.

Aufgabe: Gegeben sei das folgende Java-Programm.

```
public class Zellen {
     public static void main(String[] args) throws Exception {
```

```
        boolean[] generation = { false, true, true, true, false,
              true, true, false, true, false, true, false, true,
              false, true, false, false, true, false, false };
        final int anzahlSimulationen = 9;
        printGeneration(generation, 0);
        for (int s = 1; s <= anzahlSimulationen; s++) {
              generation = ueberfuehreGeneration(generation);
              printGeneration(generation, s);
        }
    }
}
```

Die Start-Generation ist dabei durch ein boolesches Array vorgegeben, die Anzahl an Simulationsschritten durch eine int-Konstante. Implementieren Sie die fehlenden Funktionen **printGeneration** und **ueberfuehreGeneration** gemäß der obigen Angaben. **printGeneration** gibt die als Parameter übergebene Generation auf den Bildschirm aus (s.u.), **ueberfuehreGeneration** wendet die Zustandsübergangsfunktion auf die als Parameter übergebene Generation an und liefert die Nachfolge-Generation. Bei der Ausführung des obigen Programms sollte folgende Ausgabe auf der Konsole erzeugt werden:

```
Generation 0: _###_##_#_#_#_#__#__
Generation 1: _#_#####_#_#_#______
Generation 2: __##___##_#_#_______
Generation 3: __##___###_#________
Generation 4: __##___#_##_________
Generation 5: __##____###_________
Generation 6: __##____#_#_________
Generation 7: __##_____#__________
Generation 8: __##________________
Generation 9: __##________________
```

Achtung: Beachten Sie, dass die gegebene Start-Generation und die Anzahl an Simulationsschritten natürlich nur Beispiele sind. Ihr Programm sollte auch bei geänderten Werten noch ordnungsgemäß funktionieren.

Aufgabe 70:

Die „Verflixte Eins" ist ein beliebtes Würfelspiel. Es wird mit N (>=2) Spielern gespielt, die der Reihe nach am Spielzug sind. Für das Spiel wird ein normaler Würfel (1-6) und Schreibzeug zum Notieren der Punkte benötigt. Ein Spieler startet seinen Zug und beginnt zu würfeln. Er kann innerhalb seines Zuges prinzipiell beliebig oft würfeln und seine gewürfelten Augenzahlen werden immer addiert. Sollte er jedoch eine Eins würfeln, so sind alle Punkte seines aktuellen Zuges verloren und der nächste Spieler ist dran. Für das Würfeln einer zwei, drei und vier bekommt der Spieler die entsprechende Augenzahl. Bei einer fünf oder sechs bekommt er dagegen die 10-fache Punktzahl – also 50 bzw. 60 Punkte. Das Spiel endet, sobald ein Spieler 1000 oder mehr Punkte erreicht hat. (Quelle: http://ddi.uni-

wuppertal.de/material/materialsammlung/oberstufe/oom/dieverflixteeins/-ab_01_beschreibung.pdf)

Ein Ausschnitt aus einem Spiel: Martin und Paul spielen das Würfelspiel „Die verflixte Eins“. Martin startet seinen Zug und würfelt eine 4. Das bringt ihm 4 Punkte. Er würfelt nochmal und sichert sich durch eine 5 weitere 50 Punkte. Die 54 Punkte reichen ihm und er beendet seinen Zug. Er gibt den Würfel an Paul weiter, der momentan mit 214 Punkten in Führung liegt. Paul startet seinen Zug. Er würfelt eine 6 und bekommt dafür 60 Punkte. Beim zweiten Wurf hat Paul Pech: Er würfelt eine 1. Seine 60 Punkte werden gelöscht und er muss seinen Zug mit 0 Punkten beenden. Nun ist wieder Martin an der Reihe. Er würfelt eine 5 und später eine 6 und erhöht damit seinen Punktestand um 110 Punkte (usw.).

Aufgabe: Implementieren Sie das Spiel „Verflixte Eins“ (auf imperative Art und Weise) in Java, so dass N (>=2) menschliche Spieler gegeneinander an einer Konsole antreten können. Den Wert N können Sie im Programm als Konstante festlegen (bspw. `final int N = 3;`). Ausschließlich durch Änderung des Konstantenwertes muss das Spiel aber auch mit einer anderen Anzahl an Spielern spielbar sein.

Orientieren Sie sich was Eingaben und Ausgaben angeht an folgendem beispielhaften Ablauf des Programms (N = 3; Benutzereingaben in <>):

```
Aktueller Spielstand:
Spieler 1: 0
Spieler 2: 0
Spieler 3: 0

Spieler 1 ist am Zug
gewuerfelte Zahl: 2
Bisherige Rundenpunkte: 2
Wollen Sie nochmal wuerfeln (j/n): <j>
gewuerfelte Zahl: 1
Verflixt, leider nix!
Aktueller Spielstand:
Spieler 1: 0
Spieler 2: 0
Spieler 3: 0


Spieler 2 ist am Zug
gewuerfelte Zahl: 6
Bisherige Rundenpunkte: 60
Wollen Sie nochmal wuerfeln (j/n): <j>
gewuerfelte Zahl: 3
Bisherige Rundenpunkte: 63
Wollen Sie nochmal wuerfeln (j/n): <j>
gewuerfelte Zahl: 5
Bisherige Rundenpunkte: 113
Wollen Sie nochmal wuerfeln (j/n): <n>
Aktueller Spielstand:
Spieler 1: 0
Spieler 2: 113
Spieler 3: 0

...

Spieler 3 ist am Zug
gewuerfelte Zahl: 2
```

```
Bisherige Rundenpunkte: 2
Wollen Sie nochmal wuerfeln (j/n): <j>
gewuerfelte Zahl: 5
Bisherige Rundenpunkte: 52
Wollen Sie nochmal wuerfeln (j/n): <j>
gewuerfelte Zahl: 6
Bisherige Rundenpunkte: 112
Wollen Sie nochmal wuerfeln (j/n): <n>
Aktueller Spielstand:
Spieler 1: 345
Spieler 2: 657
Spieler 3: 1012
Spieler 3 hat gewonnen!
```

Aufgabe 71:

Implementieren Sie in Java die folgende Funktion entsprechend der gegebenen javadoc-Beschreibung:

```
/*
* Rotiert die Elemente des übergebenen Arrays um die
* angegebene (positive) Distanz nach rechts. Nach Aufruf
* dieser Methode ist das Element, das zuvor an Index i
* war, nun am Index (i + distance) % array.length, und
 * zwar für alle Werte i zwischen 0 und array.length - 1.
 *
 * Beispiel: Nehmen wir das Array
 * array = [3, 7, -2, 8, 9]. Nach Aufruf von
 * rotate(array, 1) (oder auch rotate(array, 6)), hat
 * das Array den Inhalt [9, 3, 7, -2, 8].
 *
 * @param array - das zu rotierende Array (!= null)
 *
 * @param distance - die Distanz, um die die Elemente des
 * Arrays rotiert werden sollen; Achtung: kann ein
 * beliebiger nicht negativer int-Wert sein
 */
public static void rotate(int[] array, int distance)
```

Achtung: Sie dürfen bei der Implementierung der Funktion keine Methoden der JDK-Klassenbibliothek benutzen!

Aufgabe 72:

Bei dieser Aufgabe sollen Sie ein Programm entwickeln, durch das ein menschlicher Spieler am Computer das Knobelspiel *Lampen* spielen kann.

Regeln:

Das Spiel besteht aus einem quadratischen Spielfeld der Größe n ($2 < n < 10$), das aus einzelnen Zellen besteht, die Lampen repräsentieren. Lampen können sich im Zustand *an* oder *aus* befinden. Anfangs sind alle Lampen im Zustand *aus*. In jedem Spielzug wählt der Spieler eine Zelle aus, deren Zustand umgeschaltet werden soll (von *aus* in *an* oder umgekehrt). Gleichzeitig werden aber auch die Nachbarlampen oberhalb, unterhalb, links und rechts von der entsprechenden Lampe umgeschaltet (insofern sie existieren). Ziel (und Ende) des Knobelspiels ist es, den Zustand zu erreichen, dass alle Lampen angeschaltet sind.

Aufgabe:

Schreiben Sie ein Java-Programm, mit dessen Hilfe ein menschlicher Spieler das Spiel *Lampen* spielen kann.

Hinweise:

Der generelle Spielablauf lässt sich folgendermaßen skizzieren:

- Abfrage der Spielfeldgröße
- Initialisierung des Spielfeldes
- Ausgabe des Spielfeldes
- Solange das Spiel nicht beendet ist, tue folgendes
    - Korrekten Spielzug einlesen
    - Spielzug ausführen
    - Ausgabe des Spielfeldes

Beispiel für einen Programmablauf (Benutzereingaben stehen in Klammern (<>), Lampen im Zustand *aus* werden durch ein ‚.' , Lampen im Zustand *an* durch ein ‚+' gekennzeichnet):

```
Feldgroesse (2 < n < 10): <5>

 01234
0.....
1.....
2.....
3.....
4.....

Reihe der Lampe, die umgeschaltet werden soll (0 <= r < 5): <3>
Spalte der Lampe, die umgeschaltet werden soll (0 <= r < 5): <2>

 01234
0.....
1.....
2..+..
3.+++.
4..+..

Reihe der Lampe, die umgeschaltet werden soll (0 <= r < 5): <0>
Spalte der Lampe, die umgeschaltet werden soll (0 <= r < 5): <0>

 01234
0++...
1+....
2..+..
3.+++.
4..+..

Reihe der Lampe, die umgeschaltet werden soll (0 <= r < 5): <3>
Spalte der Lampe, die umgeschaltet werden soll (0 <= r < 5): <1>

 01234
0++...
1+....
```

```
2.++..
3+..+.
4.++..

...
```

Aufgabe 73:

GnomeSort-Algorithmus (aus Wikipedia):

Man stelle sich einen Gartenzwerg (*garden gnome*) vor, welcher vor *n* Blumentöpfen steht, die unterschiedliche Größen haben dürfen. Die Blumentöpfe sind in einer von links nach rechts verlaufenden Reihe aufgestellt. Anfangs steht der Gartenzwerg vor dem Blumentopf ganz links und möchte die Blumentöpfe von links nach rechts der Größe nach aufsteigend sortieren.

Der Gartenzwerg wiederholt ständig folgende Schritte: Immer, wenn er sich ganz links befindet, macht er einen Schritt nach rechts. Ansonsten vergleicht er die beiden Blumentöpfe, vor denen er gerade steht (genauer: er vergleicht den, vor dem er gerade steht, mit dem links davon). Stellt er fest, dass sie in der richtigen Reihenfolge sind, so macht er einen Schritt nach rechts. Stellt er hingegen fest, dass die Reihenfolge nicht stimmt, so vertauscht er die beiden Blumentöpfe und macht einen Schritt nach links. Fertig ist er, wenn er rechts neben dem ganz rechts stehenden Blumentopf ankommt.

Implementieren Sie eine Methode

```
public static void gnomeSort(int[] values)
```

die ein beliebiges als Parameter übergebenes Array gemäß dem GnomeSort-Algorithmus implementiert.

Aufgabe 74:

Countingsort ist ein stabiles Sortierverfahren, das eine gegebene Folge von n Zahlen aus einem beschränkten Intervall mit k Elementen mit linearem Zeitaufwand (O(n+k)) sortiert. Der Algorithmus arbeitet nicht vergleichsbasiert, sondern zählt die Zahlen der Eingabe – er arbeitet also adressbasiert. Im Vergleich zum vergleichenden Sortieren mit der bestmöglichen Komplexität O(n*log n) ergibt sich ein Vorteil, wenn die Intervalllänge k sehr klein gegenüber der Anzahl der zu sortierenden Elemente n ist. Gut geeignetes Beispiel: Sortieren aller Einwohner Bayerns (n = 12,85 Mio) nach ihrem Alter (in vollendeten Jahren: k = 0..150).

Der Algorithmus funktioniert folgendermaßen, wobei wir voraussetzen, dass es sich bei den Zahlen des Arrays um natürliche Zahlen >= 0 handelt.

1. In einem ersten Durchlauf durch das zu sortierende Array a wird der größte (max) Wert ermittelt.
2. Es wird ein Hilfsarray h der Größe max + 1 angelegt.
3. Das Array a wird ein zweites Mal durchlaufen. Für jeden gefundenen Wert w wird dabei h[w] um 1 erhöht, d.h. h[w] merkt sich, wie oft der Wert w im Array a vorkommt.
4. Anschließend werden nun das Hilfarray h und das zu sortierende Array a von vorne nach hinten durchlaufen (und das zu sortierende Array a dabei

überschrieben). Ist h[i] dabei größer als 0 wird h[i] mal nacheinander der Wert i in das Array a eingetragen.

Beispiel:

- Array a = {1, 3, 3, 1, 0, 1}
- Max = 3
- Nach (3) ist h = {1, 3, 0, 2}, d.h. der Wert 0 kommt einmal, der Wert 1 dreimal, der Wert 2 keinmal und der Wert 3 zweimal vor.
- In (4) wird nun einmal die 0, dann dreimal die 1, dann 0-mal die 2 und dann zweimal die 3 in a eingetragen, so dass anschließend gilt: a = {0, 1, 1, 1, 3, 3}, d.h. a ist sortiert.

Aufgabe:

Implementieren Sie in Java eine Methode

```
public static void countingsort(int[] array)
```

die das als Parameter übergebene Array gemäß dem oben beschriebenen Countingsort-Algorithmus sortiert. Sie können dabei voraussetzen, dass das Array keine negativen Werte enthält.

Aufgabe 75:

BogoSort-Algorithmus (aus Wikipedia):

BogoSort ist ein (extrem schlechter) Sortieralgorithmus, der folgendermaßen funktioniert: Gegeben ein Array, dessen Werte der Größe nach sortiert werden sollen. In jedem Durchlauf werden zufällig zwei Indizes des Arrays bestimmt und die entsprechenden Elemente vertauscht. Dies wird sooft wiederholt, bis das Array sortiert ist.

Aufgabe 1: Implementieren Sie eine Methode

```
public static boolean isSorted(int[] values)
```

die für ein beliebiges als Parameter übergebenes Array überprüft, ob das Array aufsteigend sortiert ist, und entsprechend true oder false liefert.

Aufgabe 2: Implementieren Sie eine Methode

```
public static void bogoSort(int[] values)
```

die ein beliebiges als Parameter übergebenes Array gemäß dem BogoSort-Algorithmus implementiert. Dabei kann die Methode isSorted aus Teilaufgabe 1 genutzt werden.

Aufgabe 76:

Ein *Schaltjahr* (englisch: leap year) ist ein Jahr, das im Unterschied zu einem *Gemeinjahr* einen zusätzlichen Schalttag, den 29. Februar, enthält. Die Schaltjahrregelung besteht aus folgenden drei Regeln:

- Die durch 4 ganzzahlig teilbaren Jahre sind Schaltjahre.

- Säkularjahre, also die Jahre, die ein Jahrhundert abschließen (z. B. 1800, 1900, 2100 und 2200) sind keine Schaltjahre.
- Schließlich sind die durch 400 ganzzahlig teilbaren Säkularjahre doch Schaltjahre. Damit sind z. B. 1600, 2000 und 2400 jeweils wieder Schaltjahre.

Teilaufgabe 1:

Implementieren Sie zunächst eine Funktion

```
static boolean isLeapYear(int year)
```

Die Funktion soll überprüfen, ob das als Parameter übergebene Jahr ein Schaltjahr ist und entsprechend *true* oder *false* zurückliefern. Die Funktion soll Seiteneffekt-frei sein, d.h. insbesondere keine Ausgaben auf die Konsole produzieren.

Teilaufgabe 2:

Implementieren Sie weiterhin eine Funktion

```
static int[] getLeapYears(int from, int to)
```

Die Funktion soll ein Array erzeugen und zurückliefern, das genau die Schaltjahre enthält, die zwischen den als Parameter übergebenen Jahren *from* und *to* liegen. Dabei soll die in Teilaufgabe 1 implementierte Funktion genutzt werden. Die Funktion soll ebenfalls Seiteneffekt-frei sein. Die Größe des Arrays soll der entsprechenden Anzahl an Schaltjahren entsprechen, insbesondere also nicht größer sein. Die Nutzung von Klassen der JDK-Klassenbibliothek ist nicht erlaubt.

Teilaufgabe 3:

Schreiben Sie ein Java-Programm, das durch Aufruf der Funktion *getLeapYears* aus Teilaufgabe 2 die Schaltjahre zwischen 1860 und 2030 berechnet und auf die Konsole ausgibt.

Aufgabe 77:

Implementieren Sie eine Methode

```
static void insertionSort(int[] numbers)
```

Die Funktion soll die Werte des im Parameter übergebenen Arrays der Größe nach sortieren und dabei den im Folgenden beschriebenen InsertionSort-Algorithmus anwenden. Beachten Sie bitte mögliche Randfälle bez. des übergebenen Parameters. Die Funktion soll nur sortieren, aber keine Ausgaben auf die Konsole tätigen.

Der InsertionSort-Algorithmus funktioniert folgendermaßen:

Gegeben sei ein Array mit n Elementen, das in aufsteigender Reihenfolge sortiert werden soll. Die Elemente werden ab Index 1 der Reihe nach von links nach rechts betrachtet:

- Merke dir das Element bei Index i.
- Suche links von Index i den so genannten Einfügeindex des Elementes. Die Suche starte dabei bei Index i-1 und läuft von rechts nach links. Es wird jeweils verglichen, ob das Element am Suchindex kleiner oder gleich dem gemerkten Element ist. Ist das nicht der Fall, wird das Element des Suchindex um eine Position nach rechts verschoben. Wird ein solches Element bei Index j gefunden, lautet der Einfügeindex j+1. Wird überhaupt kein solches Element gefunden, lautet der Einfügeindex 0.
- Füge das gemerkte Element beim Einfügeindex in das Array ein.

Das Array besteht also aus einem sortierten vorderen Teil, der in jedem Durchlauf um ein Element wächst, und einem unsortierten hinteren Teil, der entsprechend schrumpft. Betrachtet wird in jedem Durchlauf das erste Element des unsortierten Teils, das in den bereits sortierten Teil einsortiert wird.

Beispiel:

…

```
2 5 7 9 | 3 6 1 6       elem = 3
2 5 7 9   9 6 1 6
2 5 7 7   9 6 1 6
2 5 5 7   9 6 1 6
2 3 5 7   9 6 1 6

2 3 5 7 9 | 6 1 6       elem = 6
2 3 5 7 9   9 1 6
2 3 5 7 7   9 1 6
2 3 5 6 7   9 1 6
```

…

Aufgabe 78:

In Las Vegas veranstaltet Casinobesitzer Al Capone Junior ein neues Gewinnspiel. Eine Runde des Spiels läuft so ab: Jeder Teilnehmer zahlt einen Einsatz von 25 Dollar. Den Betrag setzt er auf eine „Glückszahl“, eine ganze Zahl im Bereich von 1 bis 1000. Nachdem alle Teilnehmer gesetzt haben, wählt Al zehn Zahlen, ebenfalls im Bereich von 1 bis 1000. Anschließend werden die Gewinne ausgezahlt: Für jeden Teilnehmer wird diejenige von Als Zahlen bestimmt, die am nächsten an der Glückszahl des Teilnehmers liegt. Der Abstand dieser Zahl zu seiner Glückszahl ist der Gewinn des Teilnehmers.

Ein Beispiel: Bei einer Runde des Gewinnspiels waren unter anderem fünf alte Bekannte von Al dabei. Sie setzten auf diese Glückszahlen: Bugsy: 1, Bonnie: 15, Clyde: 100, Mickey: 200, Lucky: 300. Al wählte danach diese Zahlen: 1, 35, 117, 321, 448, 500, 678, 780, 802, 999. Damit erhielt Bugsy keinen Gewinn, Bonnie erhielt 14, Clyde 17, Mickey 83 und Lucky 21 Dollar. Die jeweils 25 Dollar Einsatz gehen natürlich an Al, der damit in dieser Runde 10 Dollar Verlust gemacht hätte (5 * 25 – (14 + 17 + 83 + 21)).

Aufgabe: Schreiben Sie ein Java-Programm, das genau eine Spielrunde implementiert. Die Anzahl an Spielern sei eine feste aber prinzipiell beliebige Zahl >= 1 (→ Konstante).

- Zunächst werden alle Spieler aufgefordert, über die Konsole ihre jeweilige Glückszahl zwischen 1 und 1000 einzugeben. Die Glückszahlen werden in einem Array gespeichert.
- Anschließend werden 10 Zufallszahlen zwischen 1 und 1000 ermittelt und auf die Konsole ausgegeben (Hinweis: Es ist durchaus möglich, dass Zufallszahlen mehrfach vorkommen). Auch die Zufallszahlen werden in einem Array gespeichert.
- Danach wird für jeden Spieler gemäß den Spielregeln der Gewinn errechnet, mit dem Einsatz von 25 Dollar verrechnet und der tatsächliche Gewinn bzw. im Verlustfall „Lost“ auf den Bildschirm ausgegeben.

Beispiel für einen möglichen Programmablauf für 3 Spieler (Eingaben in <>):

```
Player 1! Your number (1<=n<=1000): <500>
Player 2! Your number (1<=n<=1000): <200>
Player 3! Your number (1<=n<=1000): <700>
Lucky number 1 = 122
Lucky number 2 = 881
Lucky number 3 = 25
Lucky number 4 = 19
Lucky number 5 = 284
Lucky number 6 = 702
```

```
Lucky number 7 = 923
Lucky number 8 = 311
Lucky number 9 = 116
Lucky number 10 = 939
Player 1: Benefit = 164
Player 2: Benefit = 53
Player 3: Lost
```

Aufgabe 79:

Spiel 51 ist ein Spiel für 2 Personen, das mit einem Paket Spielkarten gespielt wird. Die Spielkarten besitzen die Werte FROM bis TO. FROM und TO seien konstante Werte mit aktuell FROM = -1 und TO = 6, sollen sich aber prinzipiell vor einem Spielbeginn ändern lassen können. Zu jedem Spielkartenwert gibt es 4 Karten (also 4 Karten mit dem Wert -1, 4 Karten mit dem Wert 0, …, 4 Karten mit dem Wert 6). Alle Karten liegen offen aus.

Die beiden Spieler ziehen abwechselnd eine der offen liegenden Karten und legen diese auf einen Stapel. Beginnend bei einem Spielstand von 0 wird dabei der Wert der Karte zum aktuellen Spielstand addiert. Der Spieler, der als erstes den Spielstand 51 erreicht, hat verloren und das Spiel ist beendet. Sollte es zu einem Zeitpunkt keine offen liegenden Karten mehr geben, werden alle Karten des Stapels wieder offen ausgelegt.

Schreiben Sie ein Java-Programm, mit dem 2 Spieler gegeneinander das Spiel 51 an einer Konsole spielen können. Bei fehlerhaften Eingaben soll das Programm den Spieler darauf hinweisen und zu einer gültigen Eingabe auffordern. Orientieren Sie sich, was die Ein- und Ausgaben angeht, an dem folgenden Beispielablauf eines Spiels.

Beispiel für einen möglichen Programmablauf (Eingaben in <>):

```
Karten:
Nummern: -1 0 1 2 3 4 5 6
Anzahl:   4 4 4 4 4 4 4 4
Spielstand = 0
Spieler 1 ist am Zug!
Welche Karte möchtest du ziehen? <6>

Karten:
Nummern: -1 0 1 2 3 4 5 6
Anzahl:   4 4 4 4 4 4 4 3
Spielstand = 6
Spieler 2 ist am Zug!
Welche Karte möchtest du ziehen? <6>

...

Karten:
Nummern: -1 0 1 2 3 4 5 6
Anzahl:   4 4 3 4 3 3 1 0
Spielstand = 47
```

```
Spieler 1 ist am Zug!
Welche Karte möchtest du ziehen? <4>
Spielstand = 51
Spieler 1 hat verloren
```

Aufgabe 80:

Bei dieser Aufgabe sollen Sie ein Programm entwickeln, durch das ein menschlicher Spieler am Computer das Knobelspiel *Lampen* (aber nur 1-dimensional! Siehe auch Aufgabe 72) spielen kann.

Regeln:

Das Spiel besteht aus einem eindimensionalen Spielfeld der Größe *n* (n > 2), das aus einzelnen Zellen besteht, die Lampen repräsentieren. Lampen können sich im Zustand *an* oder *aus* befinden. Anfangs sind alle Lampen im Zustand *aus*. In jedem Spielzug wählt der Spieler eine Zelle aus, deren Zustand umgeschaltet werden soll (von *aus* in *an* oder umgekehrt). Gleichzeitig werden aber auch die Nachbarlampen links und rechts von der entsprechenden Lampe umgeschaltet (insofern sie existieren). Ziel (und Ende) des Knobelspiels ist es, den Zustand zu erreichen, dass alle Lampen angeschaltet sind.

Aufgabe:

Schreiben Sie ein Java-Programm, mit dessen Hilfe ein menschlicher Spieler das Spiel *Lampen* spielen kann.

Hinweise:

Der generelle Spielablauf lässt sich folgendermaßen skizzieren:

- Abfrage der Spielfeldgröße
- Initialisierung des Spielfeldes
- Ausgabe des Spielfeldes
- Solange das Spiel nicht beendet ist, tue folgendes
  - Korrekten Spielzug einlesen
  - Spielzug ausführen
  - Ausgabe des Spielfeldes

Beispiel für einen Programmablauf (Benutzereingaben stehen in Klammern (<>), Lampen im Zustand *aus* werden durch ein ‚.‘, Lampen im Zustand *an* durch ein ‚x‘ gekennzeichnet):

```
Groesse (>2): 6
......
Index: 2
.xxx..
Index: 3
.x..x.
```

```
Index: 0
x...x.
Index: 6
Fehlerhafte Eingabe! Index: 5
x....x
...
```