TU Ilmenau, Fakultät IA
FG Telematik/Rechnernetze
Prof. Dr.-Ing. G. Schäfer
http://www.tu-ilmenau.de/telematik/aup

TECHNISCHE UNIVERSITÄT
ILMENAU

Algorithmen und Programmierung

Zusatzaufgaben Java Tutorium[1]

Aufgabe 1 (Zeitformatierung)

Gegeben ist eine (lange) Zeitdauer in Sekunden
Gesucht ist ein Java Programm zur Umrechnung in Tage, Stunden, Minuten, Sekunden
Beispiel:
Eingabe: 284712
Ausgabe: 3 Tage, 7 Stunden, 5 Minuten und 12 Sekunden

Aufgabe 2 (Zahlenreihe)

Entwickeln Sie ein Java Programm, das für eine natürliche Zahl n alle geraden Zahlen kleiner gleich $2n$ ausgibt.

Aufgabe 3 (Primzahltest)

Entwickeln Sie ein Java Programm, das für eine natürliche Zahl $n \geq 2$ feststellt, ob sie eine Primzahl ist oder nicht. Prüfen Sie dazu die potenziellen echten Teiler t von n. n ist eine Primzahl, wenn es kein t mit $1 < t < n$ gibt, das n ohne Rest teilt (Modulus-Operator %); andernfalls ist n keine Primzahl.

Aufgabe 4 (Perfekte Zahlen)

Eine natürliche Zahl n nennt man eine „perfekte Zahl“ , wenn sie gleich der Summe aller ihrer echten Teiler (einschließlich 1 aber ohne n) ist. Die Summe der echten Teiler von n soll mit $T(n)$ bezeichnet werden. Perfekte Zahlen sind z.B.

$$T(6) = 3 + 2 + 1 = 6$$

$$T(28) = 14 + 7 + 4 + 2 + 1 = 28$$

Schreiben Sie einen Algorithmus, der möglichst viele perfekte Zahlen findet und ausgibt. Ihr Algorithmus muss nicht von alleine anhalten, sondern kann endlos weiterrechnen.

Aufgabe 5 (Befreundete Zahlen)

Verwandt mit den perfekten Zahlen sind „befreundete Zahlen“. Zwei Zahlen a und b sind befreundet, wenn $T(a) = b$ und $T(b) = a$. Dabei bezeichnet $T(n)$ wieder die Summe aller echten Teiler von n. Das kleinste Paar befreundeter Zahlen ist 220, 284:

$$T(220) = 1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284$$

$$T(284) = 1 + 2 + 4 + 71 + 142 = 220$$

Schreiben Sie einen Algorithmus zur Suche nach befreundeten Zahlen.

[1] Stand: 2. November 2018

Aufgabe 6 (Berechnung von π)

Eine einfache (und recht langsame) Methode die Zahl π zu berechnen, liegt darin die folgende Berechnungsvorschrift anzuwenden:

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \dots$$

Schreiben Sie ein Java-Programm, das in einer Schleife die obige Summe, und somit π, berechnet. Brechen Sie die Berechnung ab, sobald sich ihr Zwischenergebnis pro Schleifendurchlauf nur noch um weniger als $0,00000001$ ändert. Geben Sie ihr Ergebnis und die Anzahl notwendiger Iterationen an.

Aufgabe 7 (Mittelwert)

Entwickeln Sie ein Java Programm, das den Mittelwert der Zahlen in einem Feld (Array) berechnet.

Aufgabe 8 (Maximum)

Entwickeln Sie ein Java Programm, das die größte Zahl in einem Feld (Array) findet und ausgibt.

Aufgabe 9 (Sortierung)

Entwickeln Sie ein Java Programm, das die Zahlen in einem Feld der Länge n sortiert ausgibt.

Aufgabe 10 (Permutationen)

Jede (beliebige) Umsortierung einer Zahlenfolge bezeichnet man als Permutation. Beispiel: $(4, 2, 3, 1)$ ist eine Permutation von $(1, 2, 3, 4)$. $(2, 3, 2, 4)$ ist jedoch keine.
Entwickeln Sie ein Java Programm, das überprüft, ob die Zahlen in einem gegebenen Feld der Länge n eine Permutation von $(1, \dots, n)$ sind.

Aufgabe 11 (Zelluläre Automaten)

Conway's Game of Life spielt auf einem zweidimensionalen Feld, dessen Einträge (Zellen) entweder mit `0` oder `1` belegt sind. Zellen mit `1` *leben*, alle anderen sind *tot*. Zellen die sich senkrecht, waagerecht oder schräg berühren, gelten als benachbart. Durch eine gedachte Anordnung auf einem Torus (der Körper in Form eines "Donut") berühren sich auch Zellen die an gegenüberliegenden Rändern liegen. Rundenweise wird nun für jede Zelle des Feldes ihr künftiger Zustand entschieden:

- Eine tote Zelle mit *genau* 3 lebenden Nachbarn beginnt zu leben.
- Eine lebende Zelle mit *weniger als* 2 oder *mehr als* 3 lebenden Nachbarn stirbt.
- Alle restlichen Zellen behalten ihren aktuellen Zustand.

(a) Schreiben Sie ein Java-Programm, das ausgehend von einem initialisierten Feld (`int[][]`) beliebiger Größe rundenweise die Regeln des *Game of Life* anwendet.

Hinweis: Achten Sie darauf, dass Sie die neuen Zustände der Zellen nicht direkt in das Feld der aktuellen Runde schreiben können, da Sie sonst noch benötigte Informationen überschreiben. Stattdessen müssen die Zustände der kommenden Runde zunächst in einem separaten Feld zwischengespeichert werden.

Hinweis: Sollten sie die modulo-Operation einsetzen wollen, beachten Sie, dass diese von Java für negative Werte nicht mathematisch korrekt implementiert wird. So gilt zwar $-1 \mod 11 \equiv 10$, allerdings rechnet Java `-1 % 11 = -1`.

(b) Visualisieren Sie Ihr Spielfeld.

(c) Auf der Website zur Vorlesung findet sich eine Datei mit einem Testspielfeld. Bei korrekter Implementierung sollten nach 54 Runden alle Zellen auf dem Spielfeld tot sein.

Hinweis: Zum Einlesen des Spielfeldes können Sie die Klasse aup.FileUtils verwenden:

```
import aup.*;
...
String path = FileUtils.getFilePath();
int[][] triangle = FileUtils.readIntMatrix(path);
```

Aufgabe 12 (Wiener werfen)

Es gibt viele Methoden einen Näherungswert für die Zahl π zu berechnen. Ein offensichtliches Verfahren ist das Werfen von Wiener Würstchen auf den Küchenboden. Dazu gilt es zunächst parallele Linien in gleichem Abstand l auf den Boden zu zeichnen, wobei l gleich der Länge der Würstchen ist. Anschließend werden die Wiener auf den Küchenboden geworfen. Eine Näherung für die Zahl π ergibt sich durch folgende Formel

$$\pi = \frac{2 \cdot N}{S}.$$

Dabei ist N die Gesamtanzahl der Würstchen und S die Anzahl der Würstchen, welche so gelandet sind, dass sie eine der Linien auf dem Küchenboden kreuzen.
Schreiben Sie ein Programm in Java, welches N und l als Eingaben erhält und durch simuliertes Würstchenwerfen in einer Küche der Größe $100 \cdot l$ eine Näherung für π berechnet. Achten Sie dabei darauf, dass sowohl die Position, als auch die Lage (Drehung) der Würstchen zufällig und gleichverteilt bestimmt wird.

Aufgabe 13 (Post'sches Korrespondenzproblem in Java)

Das Post'sches Korrespondenzproblem (siehe auch Kapitel 6, Folie 17) ist im Allgemeinen nicht entscheidbar. Schränkt man die Suche allerdings auf Korrespondenzen mit einer maximalen Länge k ein, wird das Problem lösbar.
Entwerfen Sie ein Java-Programm, welches das Post'sche Korrespondenzproblem mit der Beschränkung $k = 16$ löst. Die Eingaben α, β des Problems sollen dabei durch zwei String-Arrays (`String[]`) repräsentiert werden (welche Sie z.B. fest in Ihr Programm encodieren dürfen).

(a) Ihr Programm soll zunächst überprüfen, ob α und β die gleiche Länge haben.

(b) Zudem soll überprüft werden, ob alle Worte aus α und β über dem Alphabet $A = \{0, 1, \ldots 9, a, b, \ldots, z\}$ (Dezimalziffern und Kleinbuchstaben) gebildet wurden und mindestens ein Zeichen lang sind.

(c) Sind die ersten beiden Bedingungen erfüllt, so soll nach einer Korrespondenz gesucht werden. Sofern eine solche existiert, soll sowohl die Korrespondenz (Folge von Indizes aus α bzw. β) als auch das resultierende zusammengesetzte Wort ausgegeben werden.

Hinweis: Zur Lösung des Problems eignet sich der Ansatz des *Backtrackings*.

Aufgabe 14 (Sudoku)

Schreiben Sie ein Programm, welches mit Hilfe von Backtracking ein Sudoku-Rätsel löst. Nutzen Sie zur Repräsentation des Spielfelds ein zweidimensionales Array `int[][]` und gehen Sie davon aus, dass nicht ausgefüllte Felder mit `0` gekennzeichnet sind.

Hinweis: Zum Testen Ihres Programms finden Sie auf unserer Webseite zur Vorlesung eine Beispieldatei für ein konkretes Sudoku-Rätsel. Dieses können Sie beispielweise mit unserer Bibliothek einlesen:

```
import aup.*;
...
String path = FileUtils.getFilePath();
int[][] a = FileUtils.readIntMatrix(path);
```

Aufgabe 15 (Emulator für Registermaschinen)

Schreiben Sie ein Programm, welches eine Registermaschine (wie in der Vorlesung kenngengelernt) emuliert. Das auszuführende Registermaschinenprogramm könnte beispielsweise im Java Quelltext definiert werden. Schöner wäre es allerdings, das Programm aus einer Textdatei einzulesen. Für letztere Variante sollten Sie zunächst versuchen, geeignete Möglichkeiten zum Einlesen der Datei zu recherchieren (Stichworte: Textdatei einlesen, Strings an „Trennzeichen“ aufspalten (z.B. Leerzeichen, siehe Methode `split` eines Strings)).

Aufgabe 16 (Zusammenhangskomponenten in Graphen)

In der Vorlesung wurde Ihnen in Kapitel 6 ab 95 ein Problem aus der Graphentheorie vorgestellt. Dabei haben Sie den Begriff einer „Zusammenhangskomponente“ in einem ungerichteten Graphen kennengelernt. Schreiben Sie ein Programm, welches alle Zusammenhangskomponenten (d.h. Mengen von Knoten) in einem gegebenen Graphen findet. Zur Repräsentation des Graphen können Sie sich z.B. an der Aufgabe 5 des 6. Übungsblatts orientieren. Beachten Sie, dass wir hier ungerichtete Graphen betrachten, d.h. dass die Richtung der Kanten keine Rolle spielt.

Hinweis: Um *eine* Zusammenhangskomponente zu finden, kann man von einem beliebigen Knoten v aus starten und alle Knoten ermitteln, welche von v aus erreichbar sind. Dies sind natürlich zunächst alle direkten Nachbarn von v. Aufbauend auf dieser Idee kann man dann z.B. rekursiv alle Knoten suchen, welche von einem Nachbarn von v (und somit auch von v aus) erreichbar sind. Dabei muss man allerdings darauf achten, eine Endlosrekursion zu vermeiden (mögliche Kreise im Graphen).

Aufgabe 17 (Ein Problem aus der Graphentheorie)

In der Vorlesung wurden Ihnen in Kapitel 6 ab Folie 95 ein Problem aus der Graphentheorie sowie ein *konstruktiver Induktionsbeweis* vorgestellt. Überführen Sie den Beweis in ein Programm zum Finden der Paare und der kanten-disjunkten Pfade.

Hinweis: Als Unterproblem müssen Sie eine Zusammenhangskomponente des Graphen finden, siehe vorherige Aufgabe.