

Aufgabensammlung für die Vorlesung Algorithmen, Datenstrukturen und Programmierung - Programmieraufgaben

Lehrstuhl Management der IT-Sicherheit

20. Mai 2008

Diese Aufgabensammlung wird in der Vorlesung Algorithmen, Datenstrukturen und Programmierung verwendet. Verbesserungsvorschläge sind stets willkommen! Eine Übersicht der Aufgaben finden Sie ab Seite 2.

Aufgabenübersicht

Inhaltsverzeichnis

1	Zahlen raten	5
2	Ein- und Ausgabe in Dateien	5
3	Matrixmultiplikation	6
4	Fibonacci-Zahlen	6
5	Postschalter	6
6	Buchverwaltung	7
7	Vergleich grundlegender Sortierverfahren	7
8	Schnelle Exponentiation	8
9	Mergesort	8
10	Summenberechnung	8
11	Syntaxanalyse	9
12	Traversierung von Bäumen	9
13	Sudoku	9
14	Populationswachstum	10

Anmerkung: Die folgende Klasse `OOUtil.java` stellt häufig benötigte Methoden zur Verfügung und wird verwendet, um den Bearbeitern gerade zu Beginn die Konzentration auf das Wesentliche zu ermöglichen:

```
1  import java.io.BufferedReader;
2  import java.io.IOException;
3  import java.io.InputStreamReader;
4
5  public class OOUtil {
6      private final float version = 1.1f;
7      public String toString()      { return "OOUtil v" + version; }
8
9      public int readInt() {
10         return readInt("Bitte geben Sie eine ganze Zahl ein: ");
11     }
12
13     public int readInt(String msg) {
14         try {
15             if (msg != null) System.out.print(msg);
16             BufferedReader input = new BufferedReader(new InputStreamReader
17                 (System.in));
18             String number = input.readLine();
19             return Integer.parseInt(number);
20         } catch (NumberFormatException e) {
21             return readInt(msg);
22         } catch (IOException e) {
23             e.printStackTrace();
24         }
25         return 0;
26     }
27
28     public float readFloat() {
29         return readFloat("Bitte geben Sie eine Fließkommazahl ein: ");
30     }
31
32     public float readFloat(String msg) {
33         try {
34             if (msg != null) System.out.print(msg);
35             BufferedReader input = new BufferedReader(new InputStreamReader
36                 (System.in));
37             String number = input.readLine();
38             return Float.parseFloat(number);
39         } catch (NumberFormatException e) {
40             return readFloat(msg);
41         } catch (IOException e) {
42             e.printStackTrace();
43         }
44         return 0;
45     }
46
47     public double readDouble() {
48         return readDouble("Bitte geben Sie eine Fließkommazahl ein: ");
49     }
50
51     public double readDouble(String msg) {
52         try {
53             if (msg != null) System.out.print(msg);
54             BufferedReader input = new BufferedReader(new InputStreamReader
```

```

        (System.in));
53     String number = input.readLine();
54     return Double.parseDouble(number);
55 } catch (NumberFormatException e) {
56     return readDouble(msg);
57 } catch (IOException e) {
58     e.printStackTrace();
59 }
60 return 0.0;
61 }
62
63 public String readString() {
64     return readString("Bitte geben Sie einen String ein: ");
65 }
66
67 public String readString(String msg) {
68     try {
69         if (msg != null) System.out.print(msg);
70         BufferedReader input = new BufferedReader(new InputStreamReader
71             (System.in));
72         return input.readLine();
73     } catch (IOException e) {
74         e.printStackTrace();
75     }
76     return "";
77 }
78
79 public char readChar() {
80     return readChar("Bitte geben Sie einen Character ein: ");
81 }
82
83 public char readChar(String msg) {
84     if (msg != null) System.out.print(msg);
85     BufferedReader input = new BufferedReader(new InputStreamReader(
86         System.in));
87     try {
88         char c = (char)input.read();
89         if(c!='\n' && c!='\r' && c!='\t') return c;
90         else return readChar(msg);
91     } catch (IOException e) {
92         e.printStackTrace();
93     }
94     return 'a';
95 }
96
97 public int getGgT(int a, int b) {
98     if (a < 0) a = Math.abs(a); // a = a * -1;
99     if (b < 0) b = Math.abs(b);
100    while (b != 0) {
101        int h = b;
102        b = a % b;
103        a = h;
104    }
105    return a;
106 }

```

Aufgabe 1 Zahlen raten

Damit Sie das in der theoretischen Übung vorgestellte Spiel „Zahlen raten“ künftig auch ohne Ihren Banknachbarn spielen können, sollen Sie in dieser ersten Aufgabe ein Programm schreiben, das Ihren Spielpartner ersetzt. Dabei nehmen Sie als Benutzer zunächst die Rolle des Ratenden ein.

Programmablauf:

Der Nutzer soll zu Beginn des Programms einmalig die Möglichkeit haben eine *Obergrenze* für die zu erratende Zahl einzugeben.

Eine *Rate-Session* beginnt mit der Eingabe des ersten Rateversuchs und endet mit dem Erraten der Zahl. Dabei erhält der Nutzer nach jedem Versuch ein *Feedback* (z.B. „geratende Zahl zu groß/ zu klein / Treffer“).

Nach jeder Rate-Session wird dem Nutzer angezeigt wie viele *Versuche* er benötigt hat. Anschließend hat er die Möglichkeit eine weitere Runde (mit derselben Obergrenze) zu spielen oder das Programm zu beenden.

Ihr Programm sollte die *Durchführung einer einzelnen Rate-Session als Funktion* realisieren. Diese Funktion erhält als Parameter die Obergrenze für die zu ratende Zahl und gibt die Anzahl der benötigten Versuche zurück. Die zu ratende Ganzzahl soll als Zufallszahl innerhalb der Funktion erzeugt werden.

Erweiterungen*:

- Ergänzen Sie Ihr Programm um eine Statistikfunktion, die ausgibt, wie viele Versuche Sie im Durchschnitt benötigt haben. Vergleichen Sie Ihr Ergebnis mit dem Ihres Nachbarn!
- Falls Ihnen selbst raten zu langweilig wird, ergänzen Sie Ihr Programm um eine Komponente, bei der Sie sich die Zahl ausdenken und der Computer versucht die Zahl zu erraten.
- Lassen Sie Ihren Computer gegen sich selbst spielen.
- Legen Sie eine Benutzerverwaltung und eine High-Score-Liste an.

Denkanregungen:

- Prüfen Sie folgende Behauptung:
Bei einer Obergrenze von 5.000 kann die Zahl bei Verwendung einer geeigneten Strategie stets mit höchstens 13 Versuchen erraten werden.
- Um wie viel erhöht sich die Zahl der maximalen Versuche bei einer Verdopplung der Obergrenze?
- Angenommen der Spielpartner würde als Antwort auf Ihren Rateversuch nur die Aussagen „Treffer“ oder „kein Treffer“ machen. Wie hoch wäre dann die Wahrscheinlichkeit bei 13 Versuchen und einer Obergrenze 5.000 die richtige Zahl zu erraten?

Aufgabe 2 Ein- und Ausgabe in Dateien

Im Rahmen dieser Aufgabe werden Sie Ihre Kenntnisse im Umgang mit Dateien auffrischen. Hierzu sollen Sie ein Programm schreiben, das auf verschiedene Arten Schreib- und Leseoperationen auf von Ihnen selbst erstellten Dateien durchführt.

Mindestfunktionalitäten Ihres Programms:

- Öffnen einer Textdatei.
- Zeichenweises Auslesen des Dateiinhalts.
- Zeilenweises Auslesen des Dateiinhalts.
- Schreiben und Einfügen von Zeichen und Zeichenketten in eine Datei.
- Schreiben einer Struktur (Inhalt beliebig; in Anlehnung an die vorhergehende Aufgabe z.B. Anzahl Versuche, Obergrenze, Spielername) in eine Datei. Anschließend wieder auslesen.

Erweiterungen*:

- Erstellen Sie für das Programm aus der vorhergehenden Aufgabe eine Statistik-Datei in der zur jeweiligen Obergrenze die durchschnittliche Anzahl an Versuchen und der Spielername gespeichert sind.
- Schreiben Sie eine Auswertungsfunktion, die die Inhalte der generierten Datei übersichtlich und nach Größe sortiert darstellt.

Aufgabe 3 Matrixmultiplikation

In dieser Aufgabe soll der Umgang mit dynamisch angelegten Feldern und die Adressierung in zweidimensionalen Arrays wiederholt werden.

Schreiben Sie ein Programm, das eine Matrix A mit der Zeilenzahl ra und der Spaltenzahl ca und eine Matrix B mit den entsprechenden Größen rb und cb einliest und das Matrix-Produkt $C = A * B$ berechnet.

Welcher Beziehung müssen ra, ca, rb, cb genügen, damit dies möglich ist?

Welche Zeilen- bzw. Spaltenzahl hat C?

Hinweis: Konsultieren Sie ggf. Ihre Unterlagen aus der theoretischen Übung.

Geben Sie C auf dem Bildschirm aus. Alle Matrix-Elemente seien vom Datentyp double. Allokieren Sie nur den unbedingt nötigen Speicherplatz.

Ergänzung: Lesen Sie die Matrizen aus einer Datei aus und fügen Sie das Ergebnis am Ende der Datei an.

Aufgabe 4 Fibonacci-Zahlen

Die folgende Aufgabe soll den Umgang mit einer rekursiven Funktion zeigen und deutlich machen, dass die Rekursion unter Umständen zwar leicht verständlich, aber im Hinblick auf die Performance sehr ungünstig sein kann.

Die Fibonacci-Zahlen haben als praktischen Hintergrund die Vermehrung von Kaninchen: Das erste Kaninchenpaar wird geboren im Monat 0. Nach einem Monat ist dieses Paar erwachsen und wieder einen Monat später gebiert es ein neues Kaninchenpaar usw. Was ist die Zahl der Kaninchen nach N Monaten? Vereinfachend sei angenommen, dass kein Kaninchen stirbt.

Zur genauen Analyse der Entwicklung der Population unterscheiden wir zwischen jungen und erwachsenen Kaninchenpaaren. Ein gerade geborenes Paar ist jung und wird nach einem Zeitschritt erwachsen. Ein erwachsenes Paar gebiert ein junges Paar nach einem Zeitschritt.

Es sei J_N die Zahl der jungen und E_N die Zahl der erwachsenen Paare nach N Zeitschritten. Anfangs (zur Zeit $N = 0$) gibt es nur ein junges Paar ($J_0 = 1, E_0 = 0$). Nach einem Monat ist das junge Paar erwachsen geworden ($J_1 = 0, E_1 = 1$). Nach zwei Monaten gebiert das nun erwachsene Paar ein junges Paar ($J_2 = 1, E_2 = 1$) und dann wieder nach dem nächsten Monat. Im gleichen Monat wird das junge Paar erwachsen ($J_3 = 1, E_3 = 2$).

Allgemein gilt: Die Zahl der neugeborenen Paare J_{N+1} ist gleich der Zahl der erwachsenen Paare E_N aus dem vorigen Zeitschritt und die Population der erwachsenen Paare E_{N+1} nimmt zu um die Zahl der jungen Paare J_N aus dem vorigen Zeitschritt.

Somit $J_{N+1} = E_N, E_{N+1} = E_N + J_N$.

Die Anfangswerte sind $J_0 = 1$ und $E_0 = 0$. Aus der linken Gleichung folgt $J_N = E_{N-1}$. Setzt man dies in die rechte Gleichung ein und ersetzt E_{N+1} durch F_N , so erhält man die rekursive Beziehung, die die Fibonacci-Zahl F_N definiert:

$F_N = F_{N-1} + F_{N-2}$ für $N \geq 2$, wobei $F_0 = F_1 = 1$.

Aufgabe:

Implementieren Sie den Algorithmus, der die Anzahl der Kaninchen zum Zeitpunkt N ($=F_N$) rekursiv berechnet. Versuchen Sie dann einen anderen (nicht rekursiven) Weg der Implementierung zu finden. Messen Sie bei beiden Vorgehensweisen die Zeit, die für die Berechnung verbraucht wird.

Hinweis: Bei der rekursiven Berechnung der Fibonacci-Zahlen steigt der Aufwand exponentiell, bei der gesuchten alternativen Berechnung nur linear.

Aufgabe 5 Postschalter

In dieser Aufgabe soll der Umgang mit Warteschlangen und Stacks geübt werden.

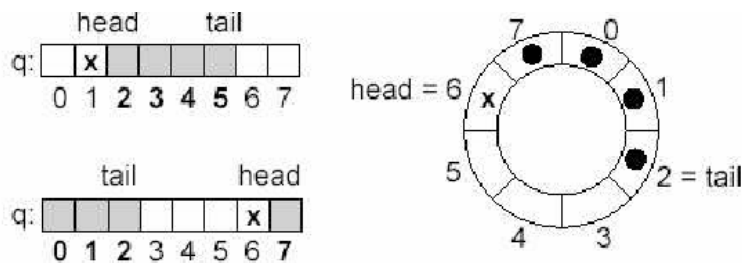
Schreiben Sie ein Programm, das den Schalter einer Postfiliale simuliert. Die Zeit wird in diskrete Zeitintervalle unterteilt. Der Ablauf an Ihrem Schalter lässt sich folgendermaßen charakterisieren:

- Pro Zeitintervall kommt mit einer Wahrscheinlichkeit von 0,4 ein neuer Kunde in Ihre Filiale.

- Jeder Kunde, der in Ihre Filiale kommt, möchte entweder genau ein Paket oder genau einen Brief aufgeben. Ein Kunde hat mit einer Wahrscheinlichkeit von 0,8 einen Brief bei sich und mit einer Wahrscheinlichkeit von 0,2 möchte er ein Paket aufgeben.
- Vor Ihrem Schalter bildet sich in der Regel eine Warteschlange.
- Falls weniger als 5 Kunden anstehen, stellt sich der Kunde an, ansonsten verlässt der Kunde Ihre Filiale wieder.
- Am Ende eines Zeitintervalls wird ein Kunde mit einer Wahrscheinlichkeit von 0,5 bedient.
- Briefe und Pakete werden hinter dem Schalter auf einen Briefstapel und auf einen Paketstapel gelegt.
- Der Briefstapel kann 80 Briefe aufnehmen.
- Der Paketstapel kann 20 Pakete aufnehmen.
- Falls entweder der Brief- oder der Paketstapel voll ist, schließt Ihr Postschalter.

Hinweise zur Implementierung:

- Die Warteschlange soll als zyklisches Array implementiert werden. Es gilt dabei das FIFO-Prinzip. Folgende Abbildungen veranschaulichen das Prinzip eines zyklischen Arrays.



- Die Stapel für Briefe und Pakete sollen als Stack implementiert werden. Es gilt das LIFO-Prinzip.

Zusatzaufgabe:

- Ermitteln Sie für verschiedene Längen der Warteschlange wie viele Kunden die Filiale wieder verlassen ohne bedient worden zu sein. Sie können zudem die Wahrscheinlichkeiten variieren.

Aufgabe 6 Buchverwaltung

Diese Aufgabe behandelt grundlegende Operationen auf doppelt verketteten Listen.

Für einen kleinen Internet-Buchladen sollen Sie ein Programm entwerfen, das die verschiedenen Bücher in einer doppelt verketteten Liste verwaltet.

Ein Knoten buch enthält als Information den Titel eines Buches, den Namen des Autors und Zeiger auf den vorhergehenden und den nachfolgenden Knoten.

Ein Zeiger head soll auf den ersten Knoten, ein Zeiger tail auf den letzten Knoten verweisen.

Schreiben Sie ein Programm, das beliebig viele Titel und die zugehörigen Autoren vom Bildschirm einliest und dabei schrittweise eine doppelt verkettete Liste aufbaut. Ordnen Sie dabei jeden neu eingelesenen Knoten so in die Liste ein, dass sich bezüglich des Autorennamens eine aufsteigende Ordnung ergibt. Innerhalb der einzelnen Titel eines Autors braucht keine bestimmte Ordnung eingehalten werden.

Der Benutzer soll anschließend die Möglichkeit zur Eingabe eines Buchtitels haben. Daraufhin wird ihm der zugehörige Autor ausgegeben. Außerdem werden die Titel aller Bücher desselben Autors dargestellt. Verwenden Sie dabei die Verkettung in beiden Richtungen.

Zusatzaufgabe:

Schaffen sie eine Möglichkeit zum Entfernen aller Bücher eines bestimmten Autors aus der Liste.

Aufgabe 7 Vergleich grundlegender Sortierverfahren

In der Vorlesung haben Sie grundlegende Sortierverfahren kennen gelernt. In dieser Aufgabe werden Sie einige davon implementieren und deren Laufzeitverhalten anhand verschiedener Testdatensätze analysieren.

Implementieren Sie für folgende Sortierverfahren Funktionen, die ein Array mit long-Werten aufsteigend sortieren:

- Bubble Sort
- Selection Sort
- Quicksort

In der VUR finden Sie das Archiv testzahlen.zip. Es enthält 3 Dateien, die im Binärmodus jeweils mit denselben 25.000 long-Werten gefüllt wurden (alle Werte positiv und < 25.000).

- testzahlen1.dat – enthält die unsortierten Werte
- testzahlen2.dat – enthält die Werte aufsteigend sortiert
- testzahlen3.dat – enthält die Werte absteigend sortiert

Schreiben Sie ein Programm, das die Dateien mit den Sortierdaten in (dynamische) Arrays einliest und diese anschließend mit Hilfe einer der oben implementierten Funktionen sortiert. Der zu verwendende Sortieralgorithmus wird zu Programmbeginn durch den Benutzer vorgegeben.

Ermitteln Sie für die einzelnen Sortiervorgänge folgende Daten:

- Anzahl der Operationen (sinnvolle Unterteilung wünschenswert, z.B. Vergleiche, Vertauschungen, Rekursive Aufrufe, etc.)
- Laufzeit der Sortierfunktion

Vergleichen Sie die ermittelten Zeiten und Werte für die einzelnen Dateien und Algorithmen (z.B. Kreuztabelle anlegen). Versuchen Sie Ihre Algorithmen zu optimieren. Wie ändert sich die Laufzeit, wenn Sie als Datentyp int verwenden?

Aufgabe 8 Schnelle Exponentiation

In der theoretischen Übung haben Sie den Pseudocode für einen Algorithmus zur schnellen Exponentiation kennen gelernt.

Schreiben Sie ein Programm, das x^y für x und y vom Typ int berechnet. Versuchen Sie diese Funktion so effizient wie möglich zu implementieren (unnötige Berechnungen vermeiden). Vergleichen Sie die Laufzeit mit der herkömmlichen Methode (Führen Sie hierzu eine grosse Zahl von Berechnungen durch).

Aufgabe 9 Mergesort

In dieser Aufgabe soll der Umgang mit dem Sortierverfahren Mergesort eingeübt werden. Dieses Verfahren erfordert es nicht die ganzen zu sortierenden Daten in den Hauptspeicher zu laden, wodurch es möglich ist sehr große Datenmengen zu sortieren.

Schreiben Sie ein Programm, das den in der Vorlesung behandelten Mergesort-Algorithmus in seiner Grundform umsetzt. Als „Bänder“ sollen drei Dateien verwendet werden, d.h. eine Datei als Eingabeband, eine als Band 1 und eine als Band 2.

Als Testdaten können Sie die drei Dateien des Archivs testzahlen.zip verwenden, die schon für

Aufgabe 7 zur Verfügung gestellt wurden.

Ermitteln Sie für jede der drei Testdateien die Anzahl der Operationen und die Laufzeit. Vergleichen Sie die Werte mit den Daten für Bubble Sort, Selection Sort und Quick Sort aus Aufgabe 7. Welche Schlüsse können Sie daraus ziehen?

Zusatzaufgabe:

Implementieren Sie das ebenfalls in der Vorlesung besprochene Balanced Merge. Wie ändert sich die Anzahl der Operationen und die Laufzeit im Vergleich zur Grundform?

Aufgabe 10 Summenberechnung

Diese Aufgabe soll deutlich machen, dass der verwendete Algorithmus unter Umständen den technischen Eigenschaften der verwendeten Programmiersprache angepasst werden muss.

Entwickeln Sie ein Programm, das die Summe $1 - 1/2 + 1/3 - 1/4 + \dots + 1/9999 - 1/10000$ auf die folgenden vier Arten berechnet:

- Addition der Terme strikt von links nach rechts,

- Addition der Terme strikt von rechts nach links,
- getrennte Addition der positiven und negativen Terme, von links nach rechts, und
- getrennte Addition der positiven und negativen Terme, von rechts nach links.

Führen Sie das Programm aus. Warum ergeben die Berechnungsmethoden unterschiedliche Resultate? Welche ist vorzuziehen und warum?

Aufgabe 11 Syntaxanalyse

Diese Aufgabe soll das Vorgehen bei der Syntaxanalyse vermitteln.

Schreiben Sie einen Parser, der den folgenden String (und auch beliebige andere Strings, die genauso aufgebaut sind) von der Stringdarstellung eines binären Baums (A(B(E(F**)(G*(H**)))(D**))(C*(I(K(J**)(L**))*))) in eine interne Repräsentation aus dynamisch generierten Knoten überführt. Die Syntax für die Stringdarstellung in EBNF ist

Node ::= '(' Char Node Node ')' | '*'

Char ::= 'A' | 'B' | 'C' | ... | 'Z'

Aufgabe 12 Traversierung von Bäumen

Diese Aufgabe soll nochmals die verschiedenen Möglichkeiten der Traversierung von Bäumen aufzeigen und gleichzeitig die Entrekursivierung vertiefen.

Schreiben Sie Funktionen, die den in der vorigen Aufgabe aufgebauten binären Baum traversieren. Es soll jeweils

- eine rekursive und
- eine iterative

Version für

- die Preorder-,
- die Inorder- und
- die Postorder-Strategie

implementiert werden.

Testen Sie die Funktionen (und Ihren Parser aus der vorhergehenden Aufgabe) auch noch mit den folgenden „Bäumen“:

- (P(M(S(A**)(A**))*)(L*(E*(R(T**)(E(E**)*))))
- (N(S*(I(E(O**)*)))(S(S*(P*(A**)))(S**))
- (T(U(N**)*)(K(R(O*(K*(N**)))(E*(R**))))
- (T(U(N**)*)(K(R(O*(K*(N**))*)(E*(R**))))
- (T(U(N**)*)(K(R(O*(K*(N**)))*)(E*(R**))))

Aufgabe 13 Sudoku

Sudoku ist ein in den letzten Jahren populär gewordenes Knobelenspiel. Ein Spiel besteht aus einem Gitterfeld mit 3 x 3 Blöcken, die jeweils in 3 x 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. 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. (Quelle: Wikipedia)

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

Gehen Sie für die folgenden Aufgaben davon aus, dass eine Klasse `Sudoku` existiert, die ein Sudokufeld in Form eines zweidimensionalen Arrays enthält, das mit einer Anfangsbelegung versehen ist und mit folgender Codezeile angelegt wurde:

```
private byte[][] sfeld = new byte[9][9];
```

Konvention für den Zugriff auf die Array-Elemente: Der erste Index bezeichnet die jeweilige Zeile, der zweite Index bezeichnet die jeweilige Spalte.

- a) (10 Min.) Schreiben Sie eine Methode `pruefeGueltigkeit`, die prüft, ob die aktuelle Belegung des Sudokus gültig ist (d.h. in keiner der Zeilen, Spalten oder Blöcke kommt eine Zahl doppelt vor - leere Felder sind hier erlaubt) und einen Wahrheitswert zurückliefert.
- b) (5 Min.) Schreiben Sie eine Methode `pruefeVollstaendigkeit`, die prüft, ob ein Sudoku vollständig belegt ist und einen Wahrheitswert zurückliefert.
- c) (5 Min.) Schreiben Sie eine Methode `getNextEmpty`, die die Koordinaten des nächsten freien Feldes (Suche von links nach rechts und von oben nach unten) als zweielementiges Array zurückliefert.
- d) (14 Min.) Schreiben Sie eine Methode `solve`, die mit Hilfe von Backtracking eine gültige Lösung für das Sudoku ermittelt, falls eine solche existiert. Die Lösung kann direkt in das Sudoku eingetragen werden. Falls keine Lösung möglich ist, soll eine `NoSolutionException` ausgelöst werden (die entsprechende Klasse muss nicht implementiert werden). Hinweise: *Sie können das Problem rekursiv oder iterativ lösen. Die rekursive Variante erscheint den Aufgabenstellern jedoch deutlich einfacher. Nutzen Sie, falls nötig, die Methoden aus den vorherigen Teilaufgaben.*

Aufgabe 14 Populationswachstum

Die Entwicklung der Größe einer Population kann mit der Gleichung $x_{i+1} = ax_i(1 - x_i)$ beschrieben werden, wobei x_i Werte zwischen 0 und 1 annehmen kann und die relative Größe der Population zum Zeitpunkt i beschreibt. Für $1 < a < 2$ konvergiert die Funktion gegen einen Grenzwert.

- a) (9 Min.) Schreiben Sie eine rekursive Funktion `popRek` mit $a = 1.05$ und $\varepsilon = 0.001$, die als Parameter den Startwert x_0 übergeben bekommt und den Wert von x_i zurückliefert, sobald sich x_i zum ersten Mal um weniger als ε von x_{i-1} unterscheidet.
- b) (8 Min.) Schreiben Sie mit den Angaben aus Teilaufgabe a) eine iterative Variante der Funktion, die zusätzlich ausgibt, nach wie vielen Schritten ε erstmals unterschritten wurde.