Übungsbeispiele Algorithmen, Datenstrukturen und Programmieren 1

Sommersemester 2010

Harald Schilly & Hermann Schichl
harald.schilly@univie.ac.at & hermann.schichl@univie.ac.at

Einleitung. Die folgenden Beispiele sind für die Programmiersprache Java. Sie sollen alleine oder zu zweit bearbeitet werden. Wichtig ist das weitgehend eigenständige Arbeiten und das tiefergehende Verständnis der einzelnen Beispiele. Die Beispiele werden von den Übungsleitern und Tutoren kontrolliert. Die Zahl der Sterne im Rechteck ist ein Hinweis für den Schwierigkeitsgrad. Die mit einem ⊕ gekennzeichneten Beispiele haben einen höheren Schwierigkeitsgrad und gehen über den Rahmen des Praktikums hinaus.

Die Ziele der Übung sind, sich mit dem Implementieren von Prozeduren vertraut zu machen, Entwicklungsumgebungen kennenzulernen, Datenstrukturen zur Repräsentation von Information im Speicher verwenden zu können, Programm-Code ausreichend verständlich zu dokumentieren und Techniken zur Verarbeitung von Daten kennenzulernen. Weitere Informationen, Hinweise und Literatur befinden sich am Schluss nach den Beispielen und auf der Webseite.

Der erste Teil bis einschließlich Beispiel “Strings” muss bis spätestens 26. April (Gruppe Schilly) bzw. 28. April (Gruppe Schichl) abgeben werden.

Der zweite Teil, beginnend mit “API”, der Beispiele beschäftigt sich ausführlich mit Objektorientierung, Standardalgorithmen und zusätzlich benötigten Fähigkeiten wie Code-Dokumentation, Performance-Tests und dem Testen von Methoden und Algorithmen. Die Beispiele sind wieder mit Schwierigkeitsgraden (erkennbar durch die Anzahl der Sterne im Rechteck) versehen, die auch gleichzeitig die Anzahl der Punkte für ein vollständig abgegebenes Beispiel wiederspiegeln. Alternativ dazu kann man auch die schwierigeren mit ⊕ gekennzeichneten Zusatzbeispiele, die am Ende des Übungsblatts angeführt sind, machen. Diese zählen jeweils 6 Punkte und setzen meist Programme aus dem zweiten Teil voraus.

Die restlichen Beispiele sind bis spätestens 21. Juni (Gruppe Schilly) bzw. 23. Juni (Gruppe Schichl) abzugeben. Am 28. Juni (Gruppe Schilly) bzw. 30. Juni (Gruppe Schichl) gibt es ein Abschlussgespräch und die Benotung.

Abmeldungen sind bis spätestens 31. März möglich, siehe §8(2) der “Satzung der Universität Wien”[1].

1. (Algorithmen, Struktogramme, ⋆) Erstelle mit Hilfe von EasyCode, Structorizer[2] oder händisch auf Papier ein Struktogramm, welches den Einkauf eines neuen Computers oder Fernsehers beschreibt. Teste die im Geschäft vorhandenen Geräte, ob sie alle Anforderungen genügen, es genug Geld zum Kaufen gibt, das günstigste gewählt wird, etc.

[1] http://www.univie.ac.at/satzung/studienrecht.html
[2] http://structorizer.fisch.lu/

2. (Algorithmen, Struktogramme II, ⋆) Erstelle mit Hilfe von EasyCode, Structorizer oder händisch auf Papier ein Struktogramm, zur Berechnung der Lösungen x_1 und x_2 einer quadratischen Gleichung $a \cdot x^2 + b \cdot x + c = 0$ für die Eingabe der Parameter a, b und c. Berücksichtige alle Sonderfälle:
   - $a = 0$
   - $a = 0$ und $b = 0$
   - komplexe Lösungen
   - Doppellösung
   - ...

   Anmerkungen für Bsp. 1-2:
   - Fang mit einem einfachen Algorithmus an (wenige Schritte).
   - Stell sicher, dass die Bedingungen Endlichkeit, Eindeutigkeit, Interpretierbarkeit für den Algorithmus erfüllt sind.
   - Verfeinere den Algorithmus durch Hinzufügen zusätzlicher Schritte und Fälle.

3. (Prozeduren I, ⋆) Definiere Variablen, die Werte für Jahre, Monate, Tage, Stunden, Minuten und Sekunden beinhalten und weise ihnen beliebige Anfangswerte zu. Das Programm soll die Gesamtzahl der Sekunden (`long`-Typ) dieser Zeitspanne berechnen und ausgeben. Dabei nehmen wir vereinfacht an, dass ein Monat immer 30 Tage und ein Jahr 360 Tage hat.
   Beispiel: 1 Jahr, 3 Monate, 10 Tage, 6 Stunden, 42 Minuten und 15 Sekunden = 39768135

4. (Prozeduren II, ⋆) Kehre das vorhergehende Programm so um, dass aus der Gesamtzahl der Sekunden wieder die ursprünglichen Werte von Jahren, Monaten, Tagen, Stunden, Minuten und Sekunden berechnet und ausgeben werden. Teste, ob die Werte übereinstimmen, wenn das vorhergehende und dieses Programm verknüpft werden.
   Beispiel: 602 = 10 Minuten, 2 Sekunden.

5. (Prozeduren III, ⋆) Speichere in den Variablen a und b eine untere und obere Grenze und in einer weiteren Variablen s die Schrittweite. Teste dass $0 < a < b$ ist. Die Zahlen sollen vom Typ `double` sein. Berechne das Produkt von allen Zahlen von a bis b mit der Schrittweite s:
   $$\prod_i r_i \text{ mit } r_i = a + i \cdot s,\ r_i \leq b,\ i \in \{0, 1, 2, \dots\}$$

6. (Prozeduren IV, ⋆) Berechnen die Gesamtkosten für eine Auto-Reise u. a. mit Hilfe einer `for`-Schleife. Speichere in Variablen die Kosten für einen Liter Benzin, die zu fahrende Strecke, Benzinverbrauch pro 100 km, die Anzahl der Tage am Urlaubsort, tägliche Kosten für Übernachtung und Verpflegung und die Heimfahrt. Dann implementiere den Ablauf von der Hinfahrt über jeden einzelnen Tag in einer `for`-Schleife und die Rückfahrt. Je nach Wert der Variablen sollen die Gesamtkosten berechnet und ausgegeben werden.

7. (Prozeduren V, ⋆) Implementiere den in Aufgabe 2 erstellten Algorithmus zur Lösung einer quadratischen Gleichung und gib das Ergebnis aus. Es genügt die reelen Lösungen ausgeben zu können, vergiss nicht die Behandlung aller sonstigen Sonderfälle.

8. (Prozeduren VI, ⋆) Addiere in einer Schleife gleichverteilte Zufallszahlen zwischen 0 und 10 bis die Summe den Wert eine festen Konstante (z. B. 100) überschreitet. Wiederhole dies 10 mal und gib jeweils die Summe und die Anzahl der jeweiligen Schleifen-

durchläufe aus.
Hinweis: gleichverteilte Zufallszahlen bekommt man mittels `Math.random()`.

9. (Prozeduren VII, ⋆) Eine beliebige Funktion $f : \mathbb{R} \to \mathbb{R}$ soll in einer separaten Methode `double f(double x)` angegeben werden. Gib eine Auswertungstabelle für diese Funktion aus. Es soll der Anfangswert, der Endwert und die Schrittweite in Variablen vorgegeben werden. Gib die Tabelle zeilenweise als formatierten `String` mit der Funktion `System.out.printf()` auf die Art aus, dass die Zahlen am Komma untereinander ausgerichtet sind und es eine vertikale Trennlinie gibt.

10. (Prozeduren VIII, ⋆) Modelliere mittels `Math.random()` und `Math.floor()` einen virtuellen Würfel mit den Seiten 1 bis 6. Würfle 3000 mal mit jeweils drei Würfeln.
    a) Wie hoch ist die ermittelte Wahrscheinlichkeit, dass alle drei Würfel die gleiche Zahl zeigen?
    b) Wie hoch ist die ermittelte Wahrscheinlichkeit, dass bei zwei aufeinanderfolgenden Würfen die Zahl 1 sichtbar ist?

    Gib die Werte aus.
    Hinweis: Die Wahrscheinlichkeit ist der Quotient aus “erwünschten Ereignissen” / “allen Ereignissen”.

11. (Debuggen I, ⋆⋆) Finde in den folgenden Codeteilen den jeweils einzigen Fehler damit das Programm dann lauffähig ist.
    Bonus: Überlege, was der Code jeweils “Mathematisches” macht!

    a)
    ```
    int f1 = 144, f2 = 84, f;
    while (f2 != 0) {
      f3 = f2;
      f2 = f1 % f2;
      f1 = f3;
    }
    System.out.println(f1);
    ```

    b)
    ```
    for (int i = 1, j = 1; i < 100; i += j, j += i) {
      System.out.printf("%s, %s, " i, j);
    }
    System.out.println("...");
    ```

12. (Debuggen II, ⋆⋆) Finde in den folgenden Codeteilen den jeweils einzigen Fehler. Das Programm lässt sich zwar kompilieren und ausführen, jedoch wurde jeweils eine wichtige Kleinigkeit übersehen!
    Bonus: Überlege, was der Code jeweils “Mathematisches” macht!

    a)
    ```
    double[] c = {0.2, −1, 0, 3, 2, 0.7 };
    double v = 0, x = 3.1415;
    int e = c.length − 1;
    for (int i = 0; i < e; i−−) {
      v =  (v + c[i]) ∗ x;
    }
    v += c[e];
    System.out.println("v: " + v);
    ```

b)

```
1 double sum = 0;
2 final int ITER = 1000000;
3 for (int i = 0; i<ITER; i++) {
4     double x = 2 * Math.random() - 1;
5     double y = 2 * Math.random() - 1;
6     double r = Math.sqrt(x^2 + x^2);
7     if (r < 1) sum++;
8 }
9 System.out.println(4 * sum/ITER);
```

13. (Kommandozeile, ⋆) In diesem Beispiel lesen wir die auf der Kommandozeile an das Programm übergebene Parameter aus. Diese stehen in der Variablen `args`, die in der `public static void main(String[] args)` Funktion angegeben wird. Die Aufgabe besteht darin, alle Parameter in umgekehrter Reihenfolge, getrennt durch Leerzeichen, in einen String zusammenzufassen und auszugeben.

14. (Eingabe, Bedingungen, ⋆⋆⋆) Erstelle eine Applikation, die Integer-Zahlen mit Werten im Bereich von 0 bis 20 als Parameterliste von der Kommandozeile einliest. Fehlerhafte Eingaben werden mit einer Fehlermeldung quittiert und ignoriert. Die eingelesenen Zahlen sollen in einem einfachen Balkendiagramm mittels Textausgabe (`System.out.println()` folgendermaßen dargestellt werden: Pro Zahl wird in einer Spalte (!) der Wert der Zahl als Sternchen-Balken (∗) und der Wert selbst dargestellt.

    Beispiel für die Ausgabe bei Eingabe von `3 10 5 3 8`

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

3 10  5  3  8
```

15. (Strings, ⋆) In diesem Beispiel beschäftigen wir uns mit den Eigenheiten von Zeichenketten (Strings). Gehe zuerst am Papier theoretisch durch, welchen Wert die einzelnen `testStrings`–Tests haben (also ob sie `true` oder `false` ausgeben). Dann führe den Code aus und vergleiche jeden einzelnen Test. Was fällt auf? Erkläre das Verhalten schriftlich!

```
1     String s1 = "abc";
2     String s2 = "abc";
3     testStrings(s1, s2);
4     s3 = new String(s2);
5     testStrings(s1, s3);
6     String s4 = "a";
7     s4 += "bc";
8     testStrings(s1, s4);
9     s1 = s1.intern();
10    s4 = s4.intern();
```

```
11      testStrings(s1, s4);
```

wobei `testStrings()` folgendermaßen definiert ist:

```
1  static void testStrings(final String a, final String b) {
2      System.out.println("equals() -> " + (a.equals(b)));
3      System.out.println("  ==     -> " + (a == b));
4  }
```

Teil 2

16. (API, ★) Lerne die Java Plattform Dokumentation und insbesondere die Java-API kennen (siehe Literaturangaben). Wo ist sie zu finden, wie ist sie aufgebaut, welche Arten von Dokumentationen gibt es? Beantworte folgende Fragen:
    a) Was ist ein “Java HotSpot Compiler” und was macht er?
    b) Welche Besonderheit hat das Paket `java.lang` im Unterschied zu allen anderen Paketen der API?
    c) Später lernen wir, dass Java eine objektorientierte Sprache ist, wobei (fast) alles von einem ganz allgemeinen Objekt abgeleitet wird. Finde dieses Objekt namens “`Object`” in der API und notiere alle aufgelisteten Methoden.
    d) Suche nun die Klasse `java.lang.Math` und erkläre, was die Methoden `hypot` und `toDegrees` machen.
    e) Finde die Klasse `ArrayList` und erkläre anhand der Beschreibung, was sie leistet, und beschreibe einen möglichen Anwendungsfall.

17. (Rekursion I, Benchmark, ★★) Berechne die Folgenglieder einer Rekursion sowohl rekursiv als auch iterativ. Es soll möglich sein bei beliebigen Anfangswerten beginnen zu können (z. B. $a_0 = 2$, $a_1 = 6$). Die rekursive Formel ist:

$$f(n) = f(n-1) - f(n-2) + n$$

    mit Anfangswerten $f(0) = a_0$ und $f(1) = a_1$.
    Programmiere die Folge sowohl *iterativ* als auch *rekursiv*.
    Teste diese rekursive Methode im Vergleich zum iterativen Ansatz in einem Benchmark (um auf aussagekräftige Zahlen zu kommen, sollte dieselbe Berechnung genügend oft wiederholt werden; siehe Glossar).
    Hinweis: Verwende den Typ `long` und berechne die ersten 40 Zahlen.

18. (Rekursion II, Benchmark, ★★) Verbessere die rekursive Berechnung der Folgenglieder im vorhergehenden Beispiel der Art, dass Zwischenergebnisse in einem Array zwischengespeichert werden. Ist das Zwischenergebnis bekannt, gib es zurück – wenn nicht, führe die Berechnung aus und speichere das Ergebnis. Der Effekt ist, dass unnötige Rekursionen eingespart werden. Mache erneut ein aussagekräftiges Benchmark (siehe Glossar).

19. (Klassen, Bruch I, ★★★) Programmiere eine Klasse `Bruch`, welche das Rechnen mit Bruchzahlen im mathematischen Sinn ermöglicht. Eine Bruchzahl ist definiert als ein Paar ganzer Zahlen, genannt “Zähler” und “Nenner”. Diese sollen private Felder der Klasse sein. Die Bruchzahl ist dann definiert durch $\frac{\texttt{Zähler}}{\texttt{Nenner}}$.

    Wie schon in der Vorlesung besprochen, soll es mehrere Konstruktoren geben:

- `public Bruch(int z, int n)` – Hauptkonstruktor, *z/n*
- `public Bruch(int z)` – ganzzahliger Bruch, *z*/1
- `public Bruch()` – Bruch mit dem Wert 1
- `public Bruch(Bruch b)` – Generiere ein *neues*, unabhängiges Bruch-Objekt aus einem existierenden Bruch Objekt ("*Copy-Constructor*").

Damit das Rechnen mit Bruchzahlen möglich ist, muss es Methoden geben, welche ein anderes Objekt der Klasse `Bruch` als Argument haben und ein *neues* Objekt generieren. Insgesamt sollen folgende Methoden programmiert werden:

- `public Bruch add(Bruch other)` – Addition
- `public Bruch add(int number)` – Addition einer Ganzzahl
- `public Bruch sub(Bruch other)` – Subtraktion
- `public Bruch mul(Bruch other)` – Multiplikation
- `public Bruch mul(double scalar)` – Multiplikation mit Skalar (approximiere die Fließkommazahl mit einem Bruch)
- `public Bruch div(Bruch other)` – Division
- `public double value()` – gibt den Wert des Bruchs als `double` zurück
- `public String toString()` – generiert eine für den Menschen lesbare Ausgabe (z. B. "3/4"), welche dann z. B. in `System.out.println()` verwendet wird.

Darüber hinaus soll die Klasse den Bruch "kürzen" können. "Kürzen" ist das gleichzeitige Dividieren des Zählers und Nenners durch deren größten gemeinsamen Teiler. Zuerst implementiere eine entsprechende Methode `void kuerzen()`. Dann eine für alle Brüche global, einheitlich konfigurierbare, private, boolesche und statische Variable `private static boolean automatischKuerzen` mit statischen "Settern" und "Gettern". Ist diese "Flag"-Variable `True`, so soll nach jeder Operation automatisch die Kürzen-Routine auf das entsprechende Ergebnis angewendet werden.

Anschließend implementiere folgende Rechnung und gib das richtige Ergebnis aus:

$$5 \cdot \left(\frac{2}{5} + \frac{6}{10} \right) / \frac{7}{2} - 1 = \frac{3}{7} = 0.\overline{42857142}$$

20. (Bruch II, Tests, ★★) Schreibe für die Klasse `Bruch` aus dem vorhergehenden Beispiel Tests. Das heißt, eine weitere Klasse `BruchTest` instanziert einige fix vorgegebene Brüche, ruft die Methoden für die Berechnungen auf, und überprüft ob jedes Ergebnis korrekt ist, indem mit dem bekannten Ergebnis verglichen wird.
Hinweis: Verwende das Schlüsselwort `assert` und erweitere die Klasse `Bruch` um einem Methode `public boolean equals(Bruch other)`, welche auf Äquivalenz testet (Achtung: auch unterschiedliche nicht gekürzte Brüche können äquivalent sein, da sie denselben Wert haben).

21. (Javadoc, Sprachelemente, Code Conventions, ★) Lies die notwendigen Kapitel der Dokumentation für "javadoc" durch. Formatiere den Code der Bruch-Klasse entsprechend der "Java Code Conventions" (siehe Literaturangaben). Insbesondere korrigiere Zeilenumbrüche, Einrückungen und beachte die korrekte Groß-/Kleinschreibung. Lerne dabei die unterschiedlichen Sprachelemente zu identifizieren! Schreibe für den Code passende Beschreibungen für die Klasse, Methoden (und Parameter) und Felder, trage den eigenen Namen als Autor ein und mach einen Querverweis (Schlüsselwort "`@link`") innerhalb des Beschreibungstextes der Klassenbeschreibung `Bruch` auf die Methode `add(Bruch other)`. Generiere anschließend die "javadoc"-Dokumentation.

22. (Klassen, Matrix I, ★★★) Schreibe eine Klasse `MyMatrix`, welche die wichtigsten Matrix-Operationen implementiert. Die Klasse soll allgemeine $m \times n$ Matrizen mit `double` Einträgen ermöglichen. Dokumentiere die Klasse und alle Methoden passend für "javadoc". Folgende Methoden soll `MyMatrix` beherrschen:

```
1  // Constructor für eine n x n Matrix
2  public MyMatrix(int n)
3  // Constructor für eine rows x cols Matrix
4  public MyMatrix(int rows, int cols)
5  // Zufallsmatrix, statische Methode, die eine n x n Matrix generiert
6  public static MyMatrix random(int n)
7  // Größe der Matrix, [rows, cols]
8  public int[] size()
9  // teste, ob die Matrix quadratisch ist
10 public boolean isSquare()
11 // gib Element an Position (r,c) zurück
12 public double get(int r, int c)
13 // setze Element (r,c) auf den Wert d
14 public void set(int r, int c, double d)
15 // addiere eine Matrix a
16 public MyMatrix add(MyMatrix a)
17 // multipliziere Matrix mit Skalar a
18 public MyMatrix mult(double a)
19 // multipliziere Matrix mit einer Matrix a
20 public MyMatrix mult(MyMatrix a)
21 // transponiere die MyMatrix und gib ein neues MyMatrix Objekt zurück
22 public MyMatrix transpose()
```

23. (Collections, ★★) Oft wird nicht nur ein Objekt benötigt, sondern eine ganze "Sammlung" von ähnlichen Objekten, welche wiederum in einem "Sammel"-Objekt gespeichert werden. Das ist eine Weiterentwicklung von Arrays (`Klasse[]`) und kann in den unterschiedlichsten Situationen nützlich sein. Erstelle ein kurzes Programm, welches folgendes bewerkstelligt:
    - Speichere die aufeinanderfolgenden ganzen Zahlen von 10 bis 100 in einem `int[]`–Array.
    - Anschließend dieselben Zahlen in einer `ArrayList<Integer>`. Dabei gibt der Klassenname in den spitzen Klammern an, welchen Typ diese `ArrayList` beinhaltet (Stichwort "*Generics*").
    - Dann diese Zahlen als assoziiertes Paar mit den Schlüsselwörtern ("1-te", "2-te", "3-te", ...) in einer `HashMap<String, Integer>`. (Der String stellt die Ordnungszahl, der Integer die Zahl selbst dar!)
    - Anschließend gib alle "Sammel"-Objekte mittels `System.out.println(Variable)` aus und berechne jeweils die Summe aller Zahlen.

    Tipp: Für die Ausgabe des Arrays ist `Arrays.toString(Variable)` nützlich.
    Weiterführende Informationen: `http://java.sun.com/docs/books/tutorial/collections/TOC.html` und in der API hier: `http://java.sun.com/javase/6/docs/technotes/guides/collections/index.html`.
    Bemerkung: Seit Java mit der Version 1.5 sind `int` und `Integer` äquivalent – das wird "autoboxing" genannt.

24. (Klassen, Interfaces & Objekte - Koffer I, ★★★) In diesem Beispiel werden wir eine Koffer-Klasse programmieren und mit Gegenständen (Objekte aus Klassen) füllen. Dabei soll folgendes implementiert werden:

- Ein *Interface* “Beschreibung”, welches die Methoden `String getName()` für den Namen des Objekts und `int getGewicht()` für das Gewicht des Objekts in Gramm vereinbart. Dieses Interface soll von den nachfolgenden abstrakten Klassen `Gegenstand` und `Koffer` implementiert werden.
- Die möglichen Gegenstände sind ebenfalls Klassen, die von einer gemeinsamen abstrakten Oberklasse `Gegenstand` abgeleitet werden – es soll mindestens 5 verschiedene geben. Vergiss nicht auf einen Pullover! Außerdem soll jeder Gegenstand in einem Feld `boolean zerbrechlich` angeben, ob der Gegenstand zerbrechlich ist.
- Eine Klasse `Koffer`, welche `Beschreibung` implementiert. Der Name des Koffers ist der Name des Eigentümers. Die Methode `int getGewicht()` berechnet das Gesamtgewicht (Eigengewicht + Summe aller Gegenstände) und eine Methode `boolean isZerbrechlich()` sagt, ob ein zerbrechlicher Gegenstand drinnen ist. Die Gegenstände sollen in einem privaten Feld `inhalt` als `ArrayList<Gegenstand>` gespeichert werden. Weiters soll sich ein Koffer selbst repräsentieren können, das heißt, überschreibe die `public toString()` Methode die von der höchsten `Object` Klasse abgeleitet wurde und gib das Gewicht und die Liste der Gegenstände zurück. Beispielausgabe für `System.out.println(koffer1)`: “`Susi's Koffer - 5,200 kg - { Hose, Schuhe, Fotoapparat, ... }`”.
- Dokumentiere das Interface, alle Klassen und Methoden für “javadoc”.

Anschließend instanziere 5 verschiedene Koffer, gib deren Beschreibungen aus und berechne deren Gesamtgewicht!

25. (Statistik I, ★★★) Aus einer Textdatei werden Wörter eingelesen. Sie sind durch mindestens ein Leerzeichen, Trennzeichen oder Zeilenumbrüche getrennt. Beispiel:

    ```
    Das ist ein Beispiel für
    einen Text wie er in einer
    Textdatei stehen könnte.
    ```

    Erstelle für die eingelesenen Wörter Statistiken. Diese soll folgende Kennzahlen ausgeben und sie in separaten Methoden, angewendet auf alle eingelesenen Wörter, berechnen:

    - Anzahl der Wörter
    - Anzahl aller Zeichen
    - Mittelwert der Wortlängen
    - Minimum und Maximum der Wortlängen
    - Standardabweichung und Varianz der Wortlängen
    - eine Liste der 10 häufigsten Wörter,
    - eine Liste der 10 häufigsten Wörter für jede vorkommende Wortlänge (also eine separate Zählung für jede Länge),

    Pass auf, dass das Programm nicht über Sonderzeichen stolpert. Verwende Sortiermethoden der Java API, sinnvolle Datentypen und achte auf effizienten Code. Dokumentiere jede Methode passend für “javadoc”.

    Hinweis: Für das Einlesen wird die Klasse `java.util.Scanner`, und für die Statistiken die Klasse `java.util.HashMap` oder eine Matrix nützlich sein!

26. (Sortieren I, ★★) Erstelle eine Klasse, welche Wörter aus einer Textdatei nach einem beliebigen Verfahren sortiert. Z. B. funktioniert das “Bubble-Sort”-Verfahren so, dass

durch fortlaufendes Vertauschen von benachbarten Einträgen die Liste sortiert wird. Es wird die Liste dabei so lange abgearbeitet, bis keine Vertauschungen mehr notwendig sind. Die Wörter sollen so wie im vorhergehenden Beispiel eingelesen und in einer neuen Textdatei ausgegeben werden.

- Der Algorithmus muss selbst implementiert werden. Die Verwendung fertiger Routinen wie zum Beispiel der `sort()`-Methode der Klasse `java.util.Arrays` ist nur für Vergleichszwecke zulässig.
- Das Einlesen und der Sortiervorgang sollen in separaten Methoden gekapselt werden. Die Methoden sollen geeignete Parameter und Rückgabewerte enthalten. Es dürfen keine Daten mittels globaler Variablen übergeben werden.
- Messe die durchschnittliche, minimale und maximale Geschwindigkeit des Programms für jeweils unterschiedlich große Mengen an zu sortierenden Zahlen (Benchmark, siehe Glossar). Vergleiche mit Kollegen, mache Vermutungen über die Komplexität, implementiere eventuell eine zweite Sortiermethode!

27. (Statistik II, ★★) Erstelle für den Text aus "Statistik I" ein Histogramm der Buchstabenhäufigkeiten, wobei alle Zeichen in Kleinbuchstaben (30 Stück) umgewandelt werden. Stelle es wie im Beispiel mit dem Balkendiagramm als Text in der Kommandozeile dar. Achte auf eine sinnvolle vertikale Skalierung.
Hinweis: Es kann der einfache Diagramm-Code aus einem früheren Beispiel recyclet werden. Beachte worauf es ankommt, wenn alter Code in einem neuen Projekt benützt werden soll.

Weiterführende Beispiele

Die folgenden Beispiele sind zur tieferen Auseinandersetzung mit dem Stoff gedacht.

1. (Statistik IV, ⊕) Ziel der folgenden Übung ist, Formeln aus der Literatur der Informatik in einem (relativ einfachen) Programm zu implementieren. Basierend auf den Techniken aus dem Programm zu Statistik III, erstelle eines, das die Entropie eines gegebenen Textes nach Shannon berechnet. Dies wird in *"Prediction and Entropy of Printed English"*, (1950) auf Seite 51 in Formel (1) erklärt:

   Die Entropie ist

   $$F_N = -\sum_{i,j} \Pr(b_i, j) \log_2(\Pr(j|b_i))$$

   wobei b_i alle n-gramme sind, (b_i, j) bedeutet, dass Buchstabe j an das n-gramm b_i angehängt wird und $\Pr(j|b_i)$ ist die bedingte Wahrscheinlichkeit, dass ein j auf b_i folgt – das ist $\Pr(b_i, j)/\Pr(b_i)$.

   Gibt es Unterschiede zwischen Deutsch, Englisch, Französisch, ... ?

   Tipp: Verwandle alle Zeichen in Großbuchstaben und arbeite nur mit diesen! Dann generiere eine Liste b_i von *allen* n-grammen (das sind Buchstaben-Arrays der Länge n) – ein guter Wert für n ist 3, teste später auch 4 und 5 (siehe Formel (2)). Anschließend iteriere über alle n-gramme, durchsuche die Texte und ermittle die beiden Wahrscheinlichkeiten für die Formel.

   Literatur: `http://languagelog.ldc.upenn.edu/myl/Shannon1950.pdf`

2. (Rekursion III, Matrix III, ⊕) Implementiere als zusätzliche, schnellere Multiplikationsroutine der `MyMatrix`-Klasse für quadratische Matrizen den "Strassen-Algorithmus"[3].

[3] `http://de.wikipedia.org/wiki/Strassen-Algorithmus`

Mache Benchmarks um die Implementation dieser Methode mit der naiven Methode vergleichen zu können und sie zu “tunen”. Es wird nämlich notwendig sein, für kleine Submatritzen auf die herkömmliche Multiplikationsmethode zurückzugreifen (“cutoff”). Welcher “Cutoff”-Wert erweist sich als günstig, ab welcher Größe ist Strassen schneller als die “naive” Methode? Für sehr große Matrizen muss eventuell der maximal zur Verfügung gestellte Heap-Speicher vergrößert werden (`-Xmx<Zahl>m` Parameter der JVM). Teste für Größen zwischen 50×50 bis 1000×1000 ob die beiden Multiplikationsmethoden identische Ergebnisse liefern.

3. (Matrix II, Test, ⊕) Schreibe eine `MySparseMatrix`-Klasse in einem sogenannten “sparse”-Format. Das bedeutet, dass nur von 0 verschiedene Elemente gemeinsam mit ihrer jeweiligen Position gespeichert werden. Implementiere die Methoden aus Matrix I für diese sparse-Matrizen. Dieses Format ist sinnvoll, um Matrizen mit sehr wenigen von 0 verschiedenen Einträgen platzsparend abzuspeichern. Verwende zum Beispiel die `java.util.HashMap` Klasse um die Assoziation von der Position mit dem entsprechenden Wert zu speichern. Definiere und verwende ein `Interface` um beide Matrix-Klassen auf ein und dieselbe Art austauschbar verwenden zu können. Schreibe Tests in einer `MatrixTest` Klasse für alle Methoden des Interfaces. Funktionieren beide Matrix-Klassen richtig?

4. (Rekursion IV, Matrix IV, Mathematik, ⊕) Ergänze die Klasse `MyMatrix`, mit einer zusätzlichen Funktion `double det(MyMatrix d)`, die durch Verwendung des Laplaceschen Entwicklungssatzes die Determinante der Matrix berechnet. Die Methode `double det()` soll eine rekursive Funktion verwenden um die Determinante der aktuellen Matrix zu bestimmen.
   Der Laplacesche Entwicklungssatz lautet:
   Für $A \in \mathbb{R}^{n \times n}$ und $i \in 1 \ldots n$ beliebig fix gewählt, gilt:

$$\det(A) = \sum_{j=1}^{n} (-1)^{i+j} a_{ij} \det(A_{ij}).$$

   A_{ij} bezeichnet eine $(n-1) \times (n-1)$ Untermatrix von A, die durch das Streichen der i-ten Zeile und j-ten Splate entsteht. Der Ausdruck a_{ij} symbolisiert den (i,j)-ten Eintrag von A.
   Die Untermatrix A_{ij} soll man durch die Methode `MyMatrix minor(int i, int j)` erhalten können.
   Schreibe Tests um den Code zu überprüfen.

5. (Simulation, ⊕) Programmiere Conway’s Game of Life. Starte mit einer zufälligen $n \times m$-Matrix, die Einträge 0 (“tot”) und 1 (“lebendig”) hat. Visualisiere die Matrix mittels einer einfachen Textausgabe auf der Konsole. In jedem Schritt wird für jede Zelle folgendes gemacht:
   - hat eine lebende Zelle 2 lebende Nachbarn, bleibt sie am Leben,
   - jede Zelle mit 3 lebenden Nachbarn bleibt oder wird lebendig,
   - jede andere Zelle stirbt aufgrund von Einsamkeit oder Überbevölkerung.

   Iteriere mit einer kurzen Pause damit sich die Entwicklung verfolgen lässt.

   Das fertige Programm soll der Art erweitert werden, dass gezielte Anfangsbedingungen vorgegeben werden können und parametrisiere die Entscheidungen über das Schicksal jeder Zelle (also bei wievielen Nachbarn überlebt die Zelle, bei wievielen wird eine geboren). Beobachte die Verhaltensänderungen und mach eine Statistik über die Lebenserwartung.

Tipps

- Fehlermeldung `java.lang.NoClassDefFoundError` beim Starten von der Kommandozeile: Die kompilierte JAVA Klasse – oder besser gesagt das Wurzelverzeichnis des kompilierten Programms – ist nicht im durchsuchten Klassenpfad. Die Lösung ist, den `classpath` zu setzen, zum Beispiel:

  `java -cp <pfad> <KlassenName> [Argumente]`

  oder in einer Verzeichnishierarchie für Pakete das Wurzelverzeichnis:

  `java -cp <Wurzelverzeichnis> paket/hierachie/<KlassenName> [Parameter]`

  für eine Klasse im Paket `paket.hierachie.<KlassenName>` dessen Wurzelverzeichnis in
  `<Wurzelverzeichnis>` liegt.

- Programmiere wenn möglich so, dass
  - kein Code doppelt vorkommt,
  - alle Variablen und nicht mehr weiter abgeleitete Methoden das Schlüsselwort `final` haben,
  - alle Methoden möglichst eingeschränkten Zugriff haben, sprich, alles, worauf man im Paket Zugriff haben soll, auf "default" (d. h. keine Angaben), alles lokale auf `private` und nur wenige, ausgewählte Methoden und Klassen auf `public` setzen,
  - möglichst wenig neue Objekte generiert werden, vor allem nicht innerhalb von Schleifen, und
  - übergebene Parameter aus nicht vertrauenswürdigen Quellen überprüft werden.

Literatur

- Guido Krüger, Thomas Stark: "Handbuch der Java-Programmierung", 5. Auflage, kostenloser Download bei `http://javabuch.de`. Dies ist eine öfters überarbeitete und gut durchdachte Einführung in die Java Sprache. Hervorzuheben sind die Kapitel: 1, 2.2, 2.3 (2.3.3), 4, 5, 6, 11, 11.4, 13.2, 15 für die Grundlagen, 7, 8, 9 für Objektorientierte Programmierung (OOP), des weiteren 10.4.1, 17.2 und 21 sowie 51.1, 51.2 und 51.5.

- "Sun JDK 5/6 Dokumentation", online unter `http://java.sun.com/j2se/1.6.0/docs/`. Vollständige Dokumentation der J2SE (Standard Edition) inklusive der API. Im Zweifelsfall ist das die beste Quelle für alles was Java betrifft. Trotz der etwas sperrigen Sprache ist es wichtig, sich in der API zurechtzufinden.

- "Java Code Conventions", online unter `http://java.sun.com/docs/codeconv/`. Da der Großteil der Zeit aus der Bearbeitung von bereits geschriebenem Codes besteht, ist es wichtig, einen konsistenten Stil beizubehalten. Dies erleichtert das Erkennen bestimmter Elemente wie Klassen, Felder, Variablen, Strukturen und verringert die Einarbeitungszeit beim Lesen fremden Codes. (siehe Kapitel 7 und 9)

- "Documentation Comments with the Javadoc Tool", online unter `http://java.sun.com/j2se/javadoc/writingdoccomments/index.html`. Fast ebenso wichtig wie ein funktionierendes Programm ist eine Dokumentation der Klassen, Methoden und Felder. Diese Information wird mittels `javadoc` extrahiert und in HTML-Dokumenten (oder PDF, etc.) gesammelt oder kann beispielsweise von IDEs in der Kontexthilfe angezeigt werden. Dies erleichtert das Verständnis von Code später und für andere.

Glossar

- *Schreibtischtest*: Notiere auf einem Papier die einzelnen Schritte mit den Zwischenergebnissen, um den Ablauf des Programms genau nachvollziehen zu können. Jeder einzelne Schritt soll ähnlich wie er im Computer stattfinden könnte, nachvollzogen werden können.

- *Benchmark*: Das ist ein Test, um die Leistungsfähigkeit eines Programms zu bestimmen. Hierfür misst man die Zeit, die es zum Ausführen braucht. Dabei ist es oft nützlich, die Größe des Problems zu ändern, um eine Aussage über die Skalierbarkeit zu erhalten. Die Zeit misst man am besten über Differenzen in der Systemzeit: `System.currentTimeMillis()` in Millisekunden (entspricht 1/1000 Sekunde!) oder `System.nanoTime()` in Nanosekunden.

  Aufgrund der Eigenschaft der virtuellen Java-Maschine, Code zuerst nur zu interpretieren und im Laufe der Zeit die Teile in effizienten Maschinencode zu übersetzen, welche häufig ausgeführt werden, ist es schwierig, Benchmarks zu machen. Daher ist es ratsam, mehrere Wiederholungen des Befehls zu machen, um auf aussagekräftige Werte zu kommen. Auch kann dies auf die Art verfeinert werden, dass in zwei verschachtelten Schleifen das Minimum über einen in der inneren Schleife berechneten Mittelwert berechnet wird.

  Neben der Zeit, kann auch der verbrauchte Arbeitsspeicher interessant sein.

  Genaueres Monitoring ist mittels Tools wie `jvisualvm` möglich. Siehe z. B. `http://java.sun.com/javase/6/docs/technotes/guides/visualvm/profiler.html` über CPU und Memory Profiling von Applikationen.

- *Rekursion*: So nennt sich eine Technik, wenn sich eine Funktion selbst erneut aufruft. Beachte stets, dass es immer einen passenden Abbruch gibt, um unbegrenzt tiefe Rekursionen zu vermeiden. In einigen Beispielen kann es günstig sein Zwischenergebnisse zu speichern, um wiederholtes Berechnen derselben Sub-Rekursion zu vermeiden. Das nennt sich “Dynamic Programming”.

- *Test*: Ein Test ist ein zusätzlicher Teil des Programms, welcher überprüft, ob Methoden oder Funktionen das Richtige berechnen. Beispielsweise kann eine Funktion `int = func1(int a)`, die zur übergebenen Variable a den Wert 10 addiert, dadurch kontrolliert werden, dass sie mit einen bestimmten Wert (z. B. 3) aufgerufen wird und die Ausgabe mit dem erwarteten Wert 13 verglichen wird.

  Dafür gibt es auch Frameworks wie `JUnit`, welche von allen gängigen Entwicklungsumgebungen unterstützt werden.