Übungsbeispiele Programmierpraktikum

Sommersemester 2011

Harald Schilly & Ferenc Domes
harald.schilly@univie.ac.at & ferenc.domes@univie.ac.at

Einleitung. Die folgenden Beispiele sollen mit der Programmiersprache Java gelöst werden. Es soll alleine oder zu zweit gearbeitet werden. Wichtig ist das weitgehend eigenständige Arbeiten und das tiefergehende Verständnis der einzelnen Beispiele. Generell herrscht Anwesenheitspflicht, davon ausgenommen sind diejenigen, welche bereits vor dem Ende die restlichen Beispiele abgeben.

Die Beispiele werden von den Übungsleitern und Tutoren kontrolliert. Die Zahl der Sterne im Rechteck ist ein Hinweis für den Schwierigkeitsgrad. 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 13./15. April 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. Achtung: Zweiergruppen müssen mindestens ein ⊕-Beispiel durchnehmen!

Die restlichen Beispiele sind bis spätestens 15. bzw. 17. Juni abzugeben. Am 22. bzw. am 24. Juni gibt es ein Abschlussgespräch und die Benotung.

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

[1] `http://www.univie.ac.at/satzung/studienrecht.html`

Anmerkungen für Bsp. 1-2: Ein Algorithmus ist eine Liste von Anweisungen mit Kontrollstrukturen, welche Daten (Variablen) auslesen, ändern oder neue erzeugen. Bevor tatsächlich programmiert wird, schadet es nicht, sich mit diesen Konzepten vertraut zu machen. Diese ersten beiden Beispiele dienen dazu, sich mit den Anweisungen `if` und `while` sowie dem Konzept von Variablen auseinanderzusetzen. Die Beispiele ab Nummer 3. betten dies dann in eine konkrete Programmiersprache (Java) mittels einer Syntax ein.

- Fang mit einem einfachen Algorithmus an (wenige Schritte, nur eine Schleife, ...).
- Stell sicher, dass die Bedingungen Endlichkeit und Eindeutigkeit erfüllt sind.
- Verfeinere den Algorithmus durch Hinzufügen zusätzlicher Schritte und Fälle.

1. (Algorithmen, Struktogramme I, ⋆) Erstelle mit Hilfe von Structorizer[2], dem visuellen Programmiertool Scratch[3] 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
   - ...

2. (Algorithmen, Struktogramme II, ⋆) Das folgende Beispiel darf auch nur in Ansätzen gelöst werden, formuliere zumindest eine Idee.
   Finde mittels einer einfachen Richtungssuche (oder Ähnlichem) eine numerische Näherung einer Nullstelle einer Funktion $f(x)$. Der Algorithmus soll bei einem Startwert x_0 beginnen, dann mittels Auswertung der Funktion f an der Stelle einer vorläufigen Näherung $\hat{x}$ einen neuen besseren Wert für $\hat{x}$ finden. Ist die Näherung gut genug (z. B. $||f(x)|| < 10^{-5}$), soll der Algorithmus abbrechen. Teste mit "Bleistift und Papier" den Ablauf der Anweisungen.
   Tipp: Eine Nullstelle befindet sich normalerweise zwischen zwei Punkten x_1 und x_2 wo $f(x_1)$ und $f(x_2)$ unterschiedliche Vorzeichen haben!

3. (Kontrollstrukturen 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 immer 360 Tage hat.
   Beispiel: 1 Jahr, 3 Monate, 10 Tage, 6 Stunden, 42 Minuten und 15 Sekunden = 39768135

4. (Kontrollstrukturen 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.

[2] `http://structorizer.fisch.lu/`
[3] `http://scratch.mit.edu/`

5. (Kontrollstrukturen III, ⋆) Berechne die Summe aller Zahlen in einer 10×10 Multiplikationstabelle mittels zwei verschachtelten `for`-Schleifen.

$$\sum_{a,b} ab \text{ mit } a, b \in \{1, \ldots, 10\}$$

6. (Kontrollstrukturen IV, ⋆) Implementiere den in Aufgabe 1 oder 2 erstellten Algorithmus zur Lösung einer quadratischen Gleichung oder Annäherung an eine Nullstelle und gib das Ergebnis aus.
   Für die quadratische Gleichung: Es genügt die reellen Lösungen zu behandeln, aber vergiss nicht die Sonderfälle (z. B. "Komplexe Lösung" als Rückmeldung geben) – `NaN` als "Lösung" ist nicht erlaubt.
   Für die Nullstellenapproximation: Die Funktion $f(x)$ soll in einer separaten Methode `static double f(double x)` angegeben werden. Zum Testen eignet sich $f(x) = sin(x)$ mit einem Startwert von $x_0 = 3$ gut.

7. (Kontrollstrukturen V, ⋆) Addiere in einer Schleife gleichverteilte Fließkomma-Zufallszahlen zwischen -10 bis $+10$, bis der Absolutbetrag der Summe den Wert einer festen Konstante K (z. B. 30) überschreitet. Wiederhole dies 10-mal und gib jeweils die Anzahl der Schleifendurchläufe aus.
   Hinweis: gleichverteilte Zufallszahlen bekommt man mittels `Math.random()`.
   Frage: Könnte dieser Algorithmus Probleme machen?

8. (ASCII I, ⋆) Eine beliebige Funktion $f : \mathbb{R} \to \mathbb{R}$ soll in einer separaten Methode `static 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, zwei Nachkommastellen haben und es eine vertikale Trennlinie gibt. Beispiel:

```
     x    |     f(x)
----------|------------
     2.00 |     2.91
     4.50 |     1.21
     6.00 |     0.72
     8.50 |   -11.92
    10.00 |  -108.03
```

9. (Kontrollstrukturen VI, ⋆⋆) Modelliere mittels `Math.random()` und `Math.floor()` einen virtuellen Würfel mit den Seiten 1 bis einschließlich 6. Würfle 12345-mal mit jeweils vier Würfeln. Wie hoch ist die ermittelte Wahrscheinlichkeit, dass ...
   a) ... alle vier Würfel die gleiche Zahl zeigen?
   b) ... drei Würfel die gleiche Zahl zeigen?
   c) ... bei zwei aufeinanderfolgenden Würfen die Zahl 6 sichtbar ist?
   d) ... die Summe der Augenzahlen aller Würfel eines Wurfes höher ist, als die Summe der Augen des vorhergehenden?

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

10. (Kontrollstrukturen VII, ★★) Hier soll die *Fläche* unter einer Kurve im Intervall $[a, b]$ einer Funktion f berechnet werden. Die Funktion $f(x)$ soll in einer separaten Methode `static double f(double x)` angegeben werden. Benütze die zusammengesetzte Trapezregel[4] und `Math.abs()`. Vergleiche die Güte der berechneten Annäherung für drei verschiedene Schrittweiten im selben Intervall.
Beispiel: $f(x) = sin(x)$ im Interval $[0, 3.14]$ ist ungefähr 2 und im Interval $[0, 6.28]$ ungefähr 4.

11. (Codeanalyse I, ★★) Versuche die folgenden Codeteilen zu verstehen, führe den Code ggf. aus. Beschreibe in ein paar Sätzen und mit Hilfe eines Struktogramms (siehe 1. & 2.) was der Code macht.

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

12. (Codeanalyse II, ★★) Wie oben, Erklärung und Struktogramm.

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

13. (Kommandozeile, ★) In diesem Beispiel lesen wir die auf der Kommandozeile an das Programm übergebene numerische Parameter aus. Diese stehen in der Variablen `args`, die in der `public static void main(String[] args)` Funktion angegeben wird. Die Werte dürfen nicht im Programm selbst in einer Variablen zugewiesen werden! (In Eclipse: "Runtime Configuration, Arguments") Die Aufgabe besteht darin die Summe, das Produkt und den Durchschnittswert aller übergebenen Zahlen zu bestimmen. Es sind Ganzzahlen und Fließkommazahlen erlaubt, jedoch keine Buchstaben (Fehlermeldung). Beispiel:

```
$ java Kommandozeile 4 5.5 6
Summe: 15.5
Produkt: 132
Durchschnitt: 5.16666667
```

14. (ASCII II, ★★★) Das folgende Programm soll eine mathematische Funktion in der Konsole zeichnen. Im Programm selbst soll eine statische Funktion $f(x)$ mit der Signatur `static double f(double x)` definiert werden, die z. B. `return x*Math.sin(x)/2;` berechnet. Weiters gibt es Variablen a und b für die linke und rechte Grenze der Darstellung. Skaliere die Differenz von $b - a$ auf eine sinnvolle Breite von 80–100 Zeichen und ebenso in die vertikale Richtung von etwa 40–50 Zeichen ($\max(f(x)) - \min(f(x)) \; \forall x \in [a, b]$). Die

[4] `http://de.wikipedia.org/wiki/Trapezregel`

Funktion selbst soll als “Kette” von zusammenhängenden ∗-Zeichen dargestellt werden. Beispiel:

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

Es ist außerdem günstig Grenzen, sowie Minima und Maxima vertikaler Werte anzugeben.

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  public static void main(String... args) {
2      String s1 = "xyz";
3      String s2 = "xyz";
4      testStrings(s1, s2);
5      s2 = new String(s2);
6      testStrings(s1, s2);
7      s2 = "x";
8      s2 += "yz";
9      testStrings(s1, s2);
10     s1 = s1.intern();
11     s2 = s2.intern();
12     testStrings(s1, s2);
13 }
```

`testStrings()` ist folgendermaßen definiert:

```
1  static void testStrings(final String a, final String b) {
2      System.out.printf("<%s>.equals(<%s>) -> %s\n", a, b, a.equals(b));
3      System.out.printf("<%s>   ==   <%s>  -> %s\n\n", a, b, 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 mit beliebigen Anfangswerten zu beginnen (z. B. $a_0 = 2$, $a_1 = 2$). Die rekursive Formel ist:

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

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 derart, 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, Komplexe Zahlen I, ★★★) Programmiere eine Klasse `Complex`, welche das Rechnen mit komplexen Zahlen im mathematischen Sinn ermöglicht. Eine komplexe Zahl ist definiert als ein Paar Fließkommazahlen, genannt “Realteil” und “Imaginärteil”. Diese sollen private Felder der Klasse sein.

Die Klasse soll mehrere Konstruktoren haben:

- `public Complex(double r, double i)` – Hauptkonstruktor, $r + i \cdot \mathrm{i}$
- `public Complex(double r)` – reelle Zahl, $r + 0 \cdot \mathrm{i}$
- `public Complex()` – Zahl mit dem Wert $0 + 0 \cdot \mathrm{i}$
- `public Complex(Complex c)` – Generiere ein *neues*, unabhängiges Complex-Objekt aus einem existierenden Complex Objekt (“*Copy-Constructor*”).

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

- `public Complex add(Complex other)` – Addition
- `public Complex add(double real)` – Addition einer reellen Zahl
- `public Complex sub(Complex other)` – Subtraktion
- `public Complex sub(double real)` – Subtraktion einer reellen Zahl
- `public Complex mul(Complex other)` – Multiplikation
- `public Complex mul(double real)` – Multiplikation mit reeller Zahl
- `public Complex div(Complex other)` – Division
- `public Complex div(doble real)` – Division mit reeller Zahl

- `public double abs()` – gibt den Absolutbetrag zurück (`Math.hypot()`)
- `public double arg()` – Argument der komplexen Zahl (Winkel im "mathematischen" Sinn)
- `public String toString()` – generiert eine für den Menschen lesbare Ausgabe (z. B. "3 + 5i"), welche dann z. B. in `System.out.println()` automatisch für Objekte dieser Klasse verwendet werden.

Anschließend implementiere folgende Rechnung und gib das richtige Ergebnis aus:

$$((2+16i)/(1-4i))+5*(-6-6i) = -33.6470588\ldots - 28.5882352\ldots i$$

20. (Komplexe Zahlen II, Tests, ★★) Schreibe für die Klasse `Complex` aus dem vorhergehenden Beispiel Tests. Das heißt, eine weitere Klasse `ComplexTest` instanziert einige fix vorgegebene komplexe Zahlen, ruft die Methoden für die Berechnungen auf, und überprüft ob jedes Ergebnis korrekt ist, indem mit dem bekannten Ergebnis verglichen wird. *Alle* Methoden sollen einzeln getestet werden!
Hinweis: Verwende das Schlüsselwort `assert` und erweitere die Klasse `Complex` um eine Methode `public boolean equals(Complex other)`, welche auf Äquivalenz testet (Achtung: auch unterschiedliche Zahlen können approximativ denselben Wert haben).

21. (Javadoc, Sprachelemente, Code Conventions, ★) Lies die notwendigen Kapitel der Dokumentation für "javadoc" durch. Formatiere den Code der `Complex`-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 `Complex` auf die Methode `add(Complex other)`.
Generiere anschließend die "javadoc"-Dokumentation als Sammlung von HTML Dateien.

22. (Klassen, Matrix I, ★★★) Schreibe eine Klasse `ComplexMatrix`, welche die wichtigsten Matrix-Operationen implementiert. Die Klasse soll allgemeine $m \times n$ Matrizen mit `Complex` Einträgen ermöglichen. Dokumentiere die Klasse und alle Methoden passend für "javadoc" und schreibe ein Beispielprogramm, das jede Methode aufruft und auf der Konsole ausgibt.
Folgende Methoden soll `ComplexMatrix` beherrschen:

```
1  // Constructor für eine n x n Matrix
2  public ComplexMatrix(int n)
3  // Constructor für eine rows x cols Matrix
4  public ComplexMatrix(int rows, int cols)
5  // Zufallsmatrix, n x n Matrix mit zufälligen Einträgen
6  public static ComplexMatrix random(int n)
7  // Größe der Matrix, [rows, cols]
8  public int[] size()
9  // gib Element an Position (r,c) zurück
10 public double get(int r, int c)
11 // setze Element (r,c) auf den Wert v
12 public void set(int r, int c, Complex v)
13 // addiere eine Matrix a
14 public ComplexMatrix add(ComplexMatrix a)
15 // multipliziere Matrix mit Skalar a
```

```
16 public ComplexMatrix mult(double a)
17 // multipliziere Matrix mit einer Matrix a
18 public ComplexMatrix mult(ComplexMatrix a)
19 // Darstellung
20 public String toString()
```

Tipp: `toString()` benützt die `toString()` Methode der `Complex`-Klasse und stellt die Matrix in tabellarischer Form z. B. mittels `String.format(...)` dar.

23. (Collections, ★★) Oft wird nicht nur ein Objekt benötigt, sondern eine ganze "Sammlung" von ähnlichen Objekten (Instanzen derselben Klasse, bzw. mit demselben Interface), 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 20 zufällig gewählte ganze Zahlen von 0 bis 1000 in
        - einem `int[]`–Array.
        - in einer `ArrayList<Integer>`.
          (Dabei gibt der Klassenname in den spitzen Klammern an, welchen Typ diese `ArrayList` beinhaltet; Stichwort "*Generics*").
        - als assoziiertes Paar mit den Schlüsselwörtern ("1-te", "2-te", "3-te", ...) in einer `HashMap<String, Integer>`.
          (Der String stellt die Ordnungszahl, das assoziierte Integer-Objekt die Zahl selbst dar!)
    - Anschließend gib alle "Sammel"-Objekte mittels `System.out.println()` aus und berechne jeweils
        - die Summe aller Zahlen, und
        - die minimale und maximale Zahl.

      Die Werte sollten für jede der Datenstrukturen jeweils übereinstimmen.

    Tipp: Für die Ausgabe des Arrays ist `Arrays.toString(Variable)` nützlich.
    Weiterführende Informationen hier[5] und in der API hier[6].
    Bemerkung: Seit Java mit der Version 1.5 sind `int` und `Integer` äquivalent – das wird "autoboxing" genannt.

24. (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, jeweils angewendet auf eine `ArrayList<String>` aller eingelesenen Wörter, berechnen:
    - Anzahl der Wörter
    - Anzahl aller Zeichen
    - Mittelwert der Wortlängen

[5] `http://download.oracle.com/javase/tutorial/collections/TOC.html`
[6] `http://download.oracle.com/javase/6/docs/technotes/guides/collections/index.html`

- Minimum und Maximum 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 (2D-Array) nützlich sein!

25. (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 eigenständig 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.
    - 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!
    - Einen Extrapunkt ★ gibt es, wenn die Sortiermethode ganz allgemein für das Interface `Comparable` programmiert wird. Hierzu soll die Sortierroutine auf dieser Liste operieren: `List<Comparable>` und die Methode `compareTo(...)` verwenden.
      Teste die dadurch gewonnene Flexibilität, indem nun derselbe Code außer der Liste von `Strings` auch eine Liste von zufälligen `Integer`-Objekten sortieren kann.

26. (Statistik II, ASCII III, ★★) Erstelle für den Text aus "Statistik I" ein Histogramm der Buchstabenhäufigkeiten, wobei alle Zeichen in Großbuchstaben umgewandelt werden sollen. Stelle es als Balkendiagramm in der Kommandozeile dar. Achte auf eine sinnvolle vertikale Skalierung. Beispiel:

```
   *
   *
   *
   *
   *     *
   * *   *
   * *   *
 * * *   *
 * * * * *
 * * * * *
 A B C D E...
```

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 `ComplexMatrix`-Klasse für quadratische Matrizen den "Strassen-Algorithmus"[7]. Mache Benchmarks um die Implementation dieser Methode mit der naiven Methode vergleichen zu können und um 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 `ComplexSparseMatrix`-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 gemeinsames `Interface` "`IComplexMatrix`" um beide Matrix-Klassen auf ein und dieselbe Art austauschbar verwenden zu können. Schreibe (modifiziere) Tests in einer `MatrixTest` Klasse für alle Methoden des Interfaces und teste beide Matriximplementationen. Funktionieren beide Matrix-Klassen richtig?

4. (Spiel, ⊕) Programmiere das Spiel "Schiffe Versenken". Ablauf:

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

a) Der Computer denkt sich ein Spielfeld aus. Es soll die Größe 10x10 haben und darauf für den Spieler (noch unbekannt) ein Schiff der Länge 4, 3 Schiffe der Länge 3, 2 mit Länge 2 und 4 mit Länge 1 horizontal oder vertikal zufällig platzieren. Jedes Feld ist maximal von einem Schiff belegt!

b) Der Hauptteil des Spiels besteht darin, das Feld mittels ASCII auf der Konsole auszugeben. Darauf sind bereits getätigte Schüsse zu sehen, wobei "X" für "Treffer" und "." für "verfehlt" steht.

c) Der Spieler gibt einen neuen Schuss ab, indem er z. B. "B5" in die Konsole eingibt.

d) Spielziel ist, alle Schiffe mit so wenig Schüssen wie möglich zu versenken!

Hinweis: Eingabe von der Konsole mittels

```
1 Scanner scanner = new Scanner(System.in);
2 ...
3 System.out.print("Ihr nächster Schuss: ");
4 String eingabe = scanner.nextLine();
```

5. (Simulation, ⊕) Programmiere Conway's Game of Life[8]. 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 derart 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,

[8] `http://en.wikipedia.org/wiki/Conway's_Game_of_Life`

- 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.
- "Oracle Java JDK 5/6 Dokumentation", online unter[9]. 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[10]. Da der Großteil der Zeit aus der Bearbeitung von bereits geschriebener 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[11]. 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

- *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

[9] `http://download.oracle.com/javase/6/docs/`
[10] `http://www.oracle.com/technetwork/java/codeconv-138413.html`
[11] `http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html`

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[12] 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.

[12] `http://download.oracle.com/javase/6/docs/technotes/guides/visualvm/index.html`