Übungsbeispiele Algorithmen, Datenstrukturen und Programmieren 1

Sommersemester 2009 – Teil I

Thomas Leitner & Harald Schilly

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 $\oplus$ 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.

1. (Algorithmen, Struktogramme, $\boxed{\star}$) Erstelle mit Hilfe von EasyCode oder händisch auf Papier ein Struktogramm, welches den Einkauf in einem Supermarkt beschreibt. Es soll auf unterschiedliche Situationen reagieren können, ein Einkaufswagen benützt werden, teste, ob alles Notwendige vorhanden ist, etc.
   Anmerkungen für Bsp. 1-3:
   - 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.
   - Teste erneut die obigen Bedingungen.

2. (Algorithmen, Struktogramme II, $\boxed{\star}$) Erstelle mit Hilfe von EasyCode oder händisch auf Papier ein Struktogramm, zur
   a) Berechnung des passenden Retourgeldes nach dem Kauf eines Kaffees in einem Automaten.
   b) Berechnung der Lösungen x_1 und x_2 eines quadratischen Gleichungssystems $a \cdot x^2 + b \cdot x + c = 0$ für die Eingabe der Parameter a, b und c – inklusive aller Sonderfälle.

3. (Prozeduren I, $\boxed{\star}$) Definiere Variablen, um die Werte von Jahren, Monaten, Tagen, Stunden, Minuten und Sekunden zu halten und weise ihnen beliebige Anfangswerte zu. Das Programm

soll die Gesamtzahl der Sekunden (`long`-Typ) berechnen und ausgeben. Dabei nehmen wir vereinfacht an, dass ein Monat immer 30 und ein Jahr 360 Tage hat.
Beispiel: 1 Jahr, 3 Monate, 10 Tage, 6 Stunden, 42 Minuten und 15 Sekunden = 39768135

4. (Prozeduren II, $\boxed{\star}$) Erweitere das vorhergehende Programm so, 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, $\boxed{\star}$) Speichere in den Variablen `a` und `b` die untere und obere Grenze und in einer weiteren Variablen `s` die Schrittweite. Die Zahlen sollen vom Typ `double` sein. Berechne nun die Summe von allen Zahlen von `a` bis `b` mit der Schrittweite `s`:
$\sum \{r_i \mid r_i = a + is,\, r_i \leq b,\, i \in \{0, 1, 2, \ldots\}\}$

6. (Prozeduren IV, $\boxed{\star}$) Hier berechnen wir die Gesamtkosten für eine Auto-Reise. 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 Schleife, und die Rückfahrt. Je nach Wert der Variablen sollen die Gesamtkosten berechnet und ausgegeben werden.

7. (Prozeduren V, $\boxed{\star}$) Implementiere den in Aufgabe 2 erstellten Algorithmus zur Lösung eines quadratischen Gleichungssystems und gib das Ergebnis aus.

8. (Prozeduren VI, $\boxed{\star}$) Addiere in einer Schleife gleichverteilte Zufallszahlen zwischen 0 und 1, solange bis die Summe den Wert eine festen Konstante (z. B. 10) überschreitet. Wiederhole dies 10 mal und gib jeweils die Summe und die Anzahl der Schleifendurchläufe aus.
Hinweis: gleichverteilte Zufallszahlen bekommt man mittels `Math.random()`.

9. (Prozeduren VII, $\boxed{\star}$) Berechne eine Auswertungstabelle für eine Funktion. Die Funktion f soll in einer separaten Methode `double f(double x)` angegeben werden. Es soll der Anfangswert, der Endwert und die Schrittweite in passenden Variablen angegeben 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, $\boxed{\star}$) Programmiere mittels `Math.random()` und `Math.round()` einen virtuellen Würfel mit den Seiten 0 bis 5. Würfle 1000 mal mit jeweils zwei Würfeln. Wie hoch ist die ermittelte Wahrscheinlichkeit, ein gleiches Paar zu erhalten? Wie hoch ist die ermittelte Wahrscheinlichkeit, zwei mal hintereinaner ein gleiches Paar zu erhalten?

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

a)

```
int f1 = 144, f2 = 84; t;
while (f2 != 0) {
  t  = f2;
  f2 = f1 % f2;
  f1 = t;
}
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! Optional: 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)

```
double x = 0, dx = 0.5, d = 1;
for (int ev = 0, r = 1; r <= 1e4; ev++) {
  if (f(x) + d * f(x + dx) < 0) {
    d *= -1;
    dx /= 2;
  }
  if (Math.abs(f(x)) < 1 / r) {
    System.out.printf("[%5s] x = %25s | f(x) = %25s \n", ev, x, f(x));
     r *= 2;
  }
  x += d * dx;
}
```

mit

```
final static double f(final double x) {
  return 2 * Math.sin(1 - Math.log(x)) * x + 0.1;
}
```

13. (Kommandozeile, ★) In diesem Beispiel lesen wir auf der Kommandozeile an das Programm übergebene Parameter aus. Die Aufgabe besteht darin, alle Parameter in umgekehrter Reihenfolge, getrennt durch ein Leerzeichen, in einen String zusammenzufassen und auszugeben.

14. (Eingabe, Bedingungen, $\boxed{\star\star\star}$) Erstelle eine Applikation, die max. 15 Integer-Zahlen mit Werten von 0 bis 20 von der Tastatur oder (wahlweise) als Parameterliste von der Kommandozeile einliest. Fehlerhafte Eingaben werden mit einer Fehlermeldung quittiert. 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. Der Benutzer muss allerdings nicht alle 15 Zahlen angeben, denn die Eingabe soll mit `x` vorzeitig abgebrochen werden können.
Hilfestellung für Lesen, Umwandlung und Testen der Eingabe: `java.util.Scanner`-Klasse. Gibt es Probleme oder unklares Verhalten, erstelle eine Struktogramm und mach einen Schreibtischtest für zwei gültige Zahlen.

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

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

 3 10  5  3  8
```

15. (Strings, $\boxed{\star}$) 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 (`true` oder `false`). Dann implementiere den Code und gib jeden einzelnen Test aus. Was fällt auf? Erkläre das Verhalten!

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

wobei `testStrings` folgendermaßen definiert ist:

```
1  void testStrings(final String s1, final String s2) {
2      System.out.println("s1.equals(s2) -> " + (s1.equals(s2)));
3      System.out.println("s1 == s2      -> " + (s1 == s2));
4  }
```

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/1.5[oder6].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.

- *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.