Übungsbeispiele Algorithmen, Datenstrukturen und Programmieren 1

Sommersemester 2009 – Teil II

Thomas Leitner & Harald Schilly

Einleitung. Dieser zweite Teil 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. Es gibt für alle verpflichtenden Beispiele dieser Übungsangabe zusammen 30 Punkte. Alternativ dazu kann man auch die schwierigeren 5 Zusatzbeispiele, die am Ende des Übungsblatts angeführt sind, machen. Diese haben jeweils 6 Punkte und somit zusammen ebenfalls 30 Punkte.

16. (Rekursion I, Benchmark, $\boxed{\star\star}$) Es sollen die berühmten Fibonacci Zahlen für zwei beliebige positive Anfangswerte *iterativ* und *rekursiv* berechnet werden. Die Formel ist: $f(n+1) = f(n) + f(n-1)$ mit Anfangswerten $f(0) = f_0$ und $f(1) = f_1$. 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 Teil 1). Hinweis: Verwende den Typ `long` (oder `BigInteger`) und berechne die ersten 40 Zahlen.

17. (Rekursion II, Benchmark, $\boxed{\star\star}$) Verbessere die rekursive Berechnung der Fibonacci Zahlen im vorhergehenden Beispiel so, dass Zwischenergebnisse für einen bestimmten Methodenaufruf in einem Array zwischengespeichert werden. Ist das Zwischenergebnis bekannt, gib es zurück – wenn nicht, führe die Berechnung aus und speichere das Ergebnis. Dadurch werden unnötige Rekursionen eingespart. Mache erneut ein aussagekräftiges Benchmark (siehe Glossar Teil 1).

18. (Klassen, Bruch I, $\boxed{\star\star\star}$) 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 lokale, 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 generiert. 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:

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

19. (Bruch II, Tests, $\boxed{\star\star}$) Schreibe für die Klasse `Bruch` aus dem vorhergehenden Beispiel Tests. Das heißt, eine weitere Klasse `BruchTest` instanziert fest vorgegebene Brüche und testet jede Methode auf deren korrekte Funktionsweise, 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.

20. (API, $\boxed{\star}$) Lerne die Java Plattform Dokumentation und insbesondere die Java-API kennen (siehe Literaturangaben Teil 1). Wo ist sie zu finden, wie ist sie aufgebaut, welche Dokumentationen gibt es? Beantworte folgende Fragen:
    a) Was ist ein "Java HotSpot Compiler" und was macht er?
    b) Wo lassen sich bekannte Fehler ("Bugs") für die momentan verwendete Java Version finden? Nenne einen.
    c) Welche Besonderheit hat das Paket `java.lang` im Unterschied zu allen anderen Paketen der API?
    d) Java ist eine objektorientierte Sprache, wobei (fast) alles von einem ganz allgemeinen Objekt abgeleitet wird. Finde dieses Objekt in der API und notiere alle Methoden.

e) Lies die Klassen im Paket `java.lang` durch. Anschließend, betrachte die Klasse `java.lang.Math` näher und erkläre, was die Methoden `hypot` und `toDegrees` machen.

f) Finde die Klasse `ArrayList` und erkläre anhand der Beschreibung, was sie leistet und beschreibe einen möglichen Anwendungsfall.

21. (Javadoc, Sprachelemente, Code Conventions, ⋆) Lies die notwendigen Kapitel der Dokumentation für "javadoc" durch. Formatiere den völlig unleserlichen Code entsprechend der "Java Code Conventions" (siehe Literaturangaben, Teil 1). Lerne dabei, die unterschiedlichen Sprachelemente zu identifizieren! Dann 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 ("link") von dem Beschreibungstext der Methode `RUN` auf `f`. Generiere anschließend die "javadoc"-Dokumentation. Den folgenden Code gibt es auch zum Download auf der Webseite.

```
1              public class
2  dirtycode {public String naME = "Name";
3              public static void main(
4  String[]    args) {
5  dirtycode   DC =
6              new dirtycode();
7  DC.         RUN();}
8              void
9  RUN()
10 {   int     A_NUM = 3;
11 System.
12     out.println( "f(" +
13         A_NUM
14 + ")=" +
15 f(A_NUM));
16 System.out.   println("char="+      GetChar(3));
17
18      System.out.println(
19
20             "fill="+fill(3.14159265));}
21 int f(int X) {
22  for(int I = 0; I < 10;I++) {
23    X+=I; if (X % 2 == 1
24    || X > 8) X += 2 * X;
25         }
26          return X;
27            }
28         char
29       GetChar(int X) {return naME.charAt(X);}
30    double fill(double
31   o) {   while(o<1000) o *= 2;
32 return o;}}
```

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  // Default constructor
2  public MyMatrix()
3  // Constructor für eine n x n Matrix
4  public MyMatrix(int n)
5  // Constructor für eine rows x cols Matrix
6  public MyMatrix(int rows, int cols)
7  // Zufallsmatrix, statische Methode, die eine n x n Matrix generiert
8  public static MyMatrix random(int n)
9  // Größe der Matrix, [rows, cols]
10 public int[] size()
11 // teste, ob die Matrix quadratisch ist
12 public boolean isSquare()
13 // gib Element an Position (r,c) zurück
14 public double get(int r, int c)
15 // setze Element (r,c) auf den Wert d
16 public void set(int r, int c, double d)
17 // addiere eine Matrix a
18 public MyMatrix add(MyMatrix a)
19 // multipliziere Matrix mit Skalar a
20 public MyMatrix mult(double a)
21 // multipliziere Matrix mit einer Matrix a
22 public MyMatrix mult(MyMatrix a)
23 // extrahiere Unter-Matrix ab Position (a,b) mit gegebener Höhe und Breite
24 public MyMatrix submatrix(int a, int b, int height, int width)
```

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 (`Typ[]`) und kann in den unterschiedlichsten Situationen nützlich sein. Erstelle ein kurzes Programm, welches folgendes bewerkstelligt:
    - Zehn Zahlen von 21 bis 30 in einem `int[]`–Array speichert.
    - Dieselben zehn Zahlen in einer `ArrayList<Integer>`. Dabei gibt der Klassenname in den spitzen Klammern an, welchen Typ diese `ArrayList` beinhaltet ("*Generics*").
    - Diese Zahlen als assoziiertes Paar mit der entsprechenden Ordnungszahl (1-te, 2-te, 3-te, ...) in einer `HashMap<Integer, Integer>` speichert. (Die erste Zahl soll die Ordnungszahl, die zweite die Zahl selbst sein!)
    - Anschließend gib alle "Sammel"-Objekte mittels `System.out.println(Variable)` aus und berechne jeweils die Summe aller Zahlen.

    Tipp: Für das Array 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 - Pizza I, ★★★) In diesem Beispiel werden wir eine Pizza-Klasse konstruieren und mit Zutaten (Objekte aus Klassen) belegen. Dabei soll folgendes implementiert werden:
    - Ein *Interface* "Beschreibung", welches die Methoden `String getName()` für den Namen des Objekts und `int getPreis()` für den Wert des Objekts in Cent erzwingt. Dieses

Interface soll von den nachfolgenden abstrakten Klassen `Zutat` und `Pizza` implementiert werden.

- Die möglichen Zutaten sind ebenfalls Klassen, die von einer gemeinsamen abstrakten Oberklasse `Zutat` abgeleitet werden – es soll mindestens 5 verschiedene geben. Vergiss nicht auf Gewürze wie Oregano! Jede Zutat soll außerdem mit einem Feld `boolean vegetarisch` angeben, ob die Zutat vegetarisch ist.
- Eine abstrakte Klasse `Pizza`, welche `Beschreibung` implementiert. Die Methode `int getPreis()` berechnet den Gesamtpreis der Pizza (Boden + Arbeitszeit + Summe der Zutaten) und eine Methode `boolean istVegetarisch()` welche testet, ob die Pizza vegetarisch ist. Die Zutaten sollen in einem privaten Feld `zutaten` als `ArrayList<Zutat>` gespeichert werden. Weiters soll sich eine Pizza selbst repräsentieren können, das heißt, überschreibe die `public toString()` Methode der höchsten `Object` Klasse und gib den Preis und die Liste der Zutaten zurück.
  Beispielausgabe für `System.out.println(pizza)`: "`Pizza Salami - 4,10 Euro - { Tomaten, Salami, Oregano }`".
- Programmiere 3 fest vorgegebene Pizzaklassen, die in ihrem jeweiligen Konstruktor die Zutaten festlegen.
- Um eine beliebige Pizza zu belegen, programmiere eine Klasse `WunschPizza`, welche eine Methode `void belegen(Zutat z)` hat und eine höhere Arbeitszeit beansprucht. Die Zutaten sollen dann im Hauptprogramm bestimmt werden, indem diese Pizza zuerst instanziert wird und dann der Pizza neu erzeugte Zutaten hinzugefügt werden.
- Dokumentiere das Interface, alle Klassen und Methoden für "javadoc".

Anschließend instanziere 5 Pizzen, darunter eine beliebig belegte, gib ihre Beschreibungen aus und berechne deren Gesamtpreis!

25. (Klassen, Objekte & Exception - Pizza II, ★★) Wir eröffnen ein Pizzageschäft und konstruieren uns nun eine sogenannte "Fabrik" ("Factory") für Pizzen. Das soll eine Klasse sein, welche in einer statischen Methode `Pizza buildPizza(String pizza)` Pizza-Objekte nur nach deren Namen (ein String) erzeugen und zurückgeben kann. Beispielsweise soll `Pizza hw = PizzaFabrik.buildPizza("Hawaii")` eine Pizza "Hawaii" erzeugen können.
Ist der Fabrik die angegebene Pizza *nicht* bekannt, so wirf eine selbstdefinierte `Unbekannte-Pizza`-Exception zurück.

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

```
5 70 -4 8 8
6 45 -51 4 3 -4
4.6 2 -5.552 21
```

Erstelle für diese Zahlenliste eine einfache Statistik. Diese soll folgende Kennzahlen ausgeben und sie in separaten Methoden, angewendet auf alle eingelesenen Zahlen, berechnen:

- Anzahl
- Mittelwert
- Minimum, Maximum

- Quantile, Median
- Standardabweichung, Varianz

Pass auf, dass das Programm nicht über irrtümlich verstecke Zeichen 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` nützlich sein!

27. (Sortieren I, ★★) Erstelle eine Klasse, welche Integer-Zahlen aus einer Textdatei nach einem beliebigen Verfahren sortiert. Z. B. funktioniert das "Bubble-Sort"-Verfahren so, dass durch fortlaufendes Vertauschen von benachbarten Zahlen die Liste sortiert wird. Es wird die Liste dabei so lange abgearbeitet, bis keine Vertauschungen mehr notwendig sind. Die Zahlen 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 Teil 1). Vergleiche mit Kollegen, mache Vermutungen über die Komplexität, implementiere eventuell eine zweite Sortiermethode!

28. (Statistik II, ★★) Erstelle für die Zahlenliste aus "Statistik I" ein Histogramm, das die Häufigkeiten der Zahlen in gleichmäßig verteilten Intervallen zählt und darstellt. Stelle es wie im Beispiel mit dem Balkendiagramm als Text in der Kommandozeile dar. Achte auf eine sinnvolle vertikale Skalierung, und die Anzahl der Balken soll einstellbar sein.
Hinweis: Es kann der einfache Diagramm-Code aus einem früheren Beispiel recycelt werden. Beachte worauf es ankommt, wenn alter Code in einem neuen Projekt benützt werden soll.

29. (Statistik III, ★★★) Schreibe ein Analyseprogramm für Texte. Eine gute Quelle ist das Projekt Gutenberg (`http://www.gutenberg.org/catalog/`). Lade reine Textdateien herunter, welche von dem Programm nach folgenden Kriterien analysiert werden sollen:
    - Anzahl, Länge (minimal, durchschnittlich, maximal) der Wörter,
    - eine Liste der 10 häufigsten Wörter,
    - eine Liste der 10 häufigsten Wörter für jede vorkommende Wortlänge (also eine separate Zählung für jede Länge),
    - eine Statistik die angibt, welcher Buchstabe besonders häufig auf einen anderen innerhalb eines Wortes folgt.

    Beachte, dass Sonderzeichen ignoriert werden sollen und fange etwaige Sonderfälle ab.
    Die Klasse `java.util.Scanner` wird abermals nützlich sein und für die Statistiken die Klasse `java.util.HashMap` oder eine Matrix.

Weiterführende Beispiele

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

1. (Matrix II, Test, $\oplus$) 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?

2. (Rekursion III, Matrix III, $\oplus$) Implementiere als zusätzliche, schnellere Multiplikationsroutine der `MyMatrix`-Klasse für quadratische Matrizen den "Strassen-Algorithmus". (`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. (Rekursion IV, Matrix IV, Mathematik, $\oplus$) 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 $i \in 1 \dots n$ 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()` erhalten können. Schreibe einige Tests um den Code zu überprüfen.

4. (Threads & Parallelisierung, Logging, $\oplus$) Hier soll eine einfache Simulation eines Pizzalokales programmiert werden.

   - Erzeuge in parallelen Threads (einer pro Pizza) 6 Pizzen, wobei jede eine eindeutige ID-Nummer (0, 1, ...) bekommen soll.

- Diese sollen auf eine Ablagefläche mit einem Platz für maximal zwei Pizzen abgelegt werden. Ist kein Platz mehr vorhanden, müssen die jeweiligen Threads für das Pizza-Erzeugen warten.
- Es gibt zwei Öfen, welche jeweils eine (und nur eine, und nicht beide gleichzeitig dieselbe) Pizza von der Ablagefläche nehmen, backen (2 Sekunden reichen) und ausgeben.
- Ausgegeben werden sie, indem sie nach dem Backen in einem Datenobjekt unbegrenzter Größe gesammelt werden, wobei auch der Gesamtpreis aller Pizzen berechnet wird.
- Protokolliere alle Ereignisse der Simulation im "`Logger`" als Informationseintrag. Vermerke darin wer gerade aktiv ist und die ID-Nummer des jeweiligen Threads (`Thread.currentThread().getId()`).

Hinweis: Verwende die `Thread.sleep()` Methode, um den jeweiligen Thread für eine kurze Zeit (wenige Sekunden) zu unterbrechen. Verwende außerdem die Klassen `java.util.concurrent.ArrayBlockingQueue` für die begrenzte Ablagefläche, `java.util.concurrent.ConcurrentLinkedQueue` für die Ausgabe und eine statische, global verfügbare Instanz von `java.util.concurrent.atomic.AtomicInteger` über alle Pizza-Erzeugen Threads hinweg, um den Pizzen eindeutige ID-Nummern zu verpassen!
Tipp: Zum Loggen instanziere ein globales Logger-Objekt: `Logger log = Logger.getAnonymousLogger()` und erzeuge Einträge mittels `log.info("Event")`.

5. (Statistik IV, $\oplus$) Basierend auf den Techniken aus dem Programm zu Statistik III, erstelle eines, welches die Sprache eines kurzen, neuen Textes erraten kann. Lese hierfür Texte von mindestens 4 verschiedenen Sprachen ein und teste den eingegebenen Text gegen diese. Eine gute Strategie könnte sein, für jede Sprache eine Statistik der häufigsten Wörter anzulegen. Die Wörter der unbekannten Sprache können dann gegen diese Statistik getestet werden. Übereinstimmungen mit häufig vorkommenden Wörtern sind ein starkes Indiz für diese Sprache. Neben der Häufigkeit der Wörter könnten aber auch noch andere Parameter relevant sein!

   Tipp: Es kann praktisch sein, für jede Sprache ein separates Verzeichnis anzulegen um dort die Texte zu sammeln. Diese werden beim Start automatisch geladen und verarbeitet. Die Namen der Unterverzeichnisse legen die jeweilige Sprache fest. Achte auf die Skalierbarkeit und Performance, das heißt, speichere z. B. von den eingelesenen Texten nur die 10000 häufigsten Wörter und nicht den gesamten Text, um einen Speicherüberlauf zu verhindern.