Prof. Andreas Goerdt
M. Pippig, M. Riedel, U. Schönke

TU Chemnitz
Wintersemester 2003/2004

Algorithmen und Programmierung

6. Übung – Lösungsvorschläge

1. Aufgabe:

   Herangehensweise:

   Wir überlegen uns zunächst, wie groß b maximal sein kann. Bei der kleinstmöglichen Basis $a = 2$ wäre b maximal gleich $\log_2 n$. Bei größeren a wäre b auf jeden Fall kleiner. Wir suchen unser b also im Intervall von 2 bis $\log_2 n$. Für jedes dieser b müssen wir nun überprüfen, ob es ein a gibt mit $a^b = n$. Dafür wenden wir den Algorithmus zur binären Suche aus der Vorlesung an. Als obere Grenze des Suchintervalls können wir dabei $n/2+1$ wählen, da a auf keinen Fall größer als $n/2$ sein kann.

   Aus diesen Überlegungen ergibt sich folgendes Programm:

```
import Prog1Tools.IOTools;
public class PotenzSuchen {

  public static void main(String[] args) {

    long a, b, n, maxB, i, nTemp, test;
    long untereGrenze, obereGrenze, mitte;
        // Grenzen des Suchintervalls,
        // mitte enthaelt Haelfte der Elementanzahl

    // Initialisierung
    n = IOTools.readLong("n = ");
    a = 2;
    b = 2;
    maxB = 0;
    nTemp = n;
    while (nTemp / 2 > 0) {
      nTemp = nTemp / 2;
      maxB = maxB + 1;
    }

    // alle moegl. b's testen
    while (b <= maxB) {
      /*****************************************************/
      /* binaere Suche                                     */
```

```
untereGrenze = 2;
obereGrenze  = n / 2 + 1;

// binaere Suche ueber die a's
while (untereGrenze + 1 < obereGrenze) {

  // Haelfte der Elementanzahl
  mitte = (obereGrenze – untereGrenze) / 2;

  // (untereGrenze + mitte)^b berechnen
  test = untereGrenze + mitte;
  i = 1;
  while (i < b) {
    // Bereichsueberschreitung ausschliessen
    if ((java.lang.Long.MAX_VALUE/(untereGrenze + mitte)) < test){
      test = java.lang.Long.MAX_VALUE;
      i=b;
    }
    else {
      // keine Bereichsueberschreitung
      test = test * (untereGrenze + mitte);
      i = i + 1;
    }
  }

  // Test, ob links o. rechts weiter
  if (test <= n) {
    untereGrenze = untereGrenze + mitte;
  }
  else {
    obereGrenze = obereGrenze – mitte;
  }
}

// Potenz a hoch b berechnen
a = untereGrenze;
test = a;
i = 1;
while (i < b) {
  test = test * a;
  i = i + 1;
}
```

```
            // tatsaechlich Potenz gefunden?
            if (test == n) {
               System.out.println("gefunden: a = " + a + ", b = " + b);
            }
            /* Binaere Suche zu Ende                                    */
            /************************************************************/

            // naechstes b testen
            b = b + 1;
         }
         return;
      }
   }
```

2. Aufgabe:

   a) Ausgabe:
   ```
   i = 3, j = 9
   i = 3, j = 6
   ```
   Grund:

   Ausdrücke werden so abgearbeitet, dass Operationen mit höherer Priorität vor Operationen mit niedrigerer Priorität ausgeführt werden. Operationen mit gleicher Priorität werden von links nach rechts abgearbeitet, es sei denn, es ist mit Klammern eine Reihenfolge festgelegt.

   Im ersten Fall `((i=3)*i)` wird zunächst `(i=3)` ausgewertet und damit `i` der Wert 3 zugewiesen. Danach wird `i` ausgewertet (hat jetzt den Wert 3) und die Multiplikation ausgeführt. `j` erhält folglich den Wert 9.

   Im zweiten Fall `(i*(i=3))` wird erst `i` ausgewertet (hat den Wert 2) und danach `(i=3)` ausgeführt. Das Ergebnis der Multiplikation ist also 6.

   b) Ausgabe:
   ```
   a = 12
   b = 12
   ```
   Die erste Berechnung `(a += (a=3))` ist identisch zu `(a = a + (a=3))`. Zunächst wird `a` ausgewertet (hat den Wert 9) und danach die Zuweisung `(a=3)` ausgeführt. Das Ergebnis der Addition ist 12.

   Die zweite Berechnung `(b = b + (b=3))` wird analog ausgeführt und liefert das selbe Ergebnis (12), da `b` den gleichen Anfangswert (9) hat.

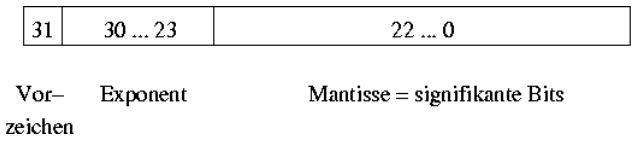
   c) Ausgabe:
   ```
   true
   ```
   Der Ausdruck `(a || b == b && c)` enthält die Operatoren `||` (Priorität 3), `==` (Priorität 8) und `&&` (Priorität 4). Zuerst wird also `(b == b)` ausgewertet, was

`true` ergibt. Danach wird `(true && c)` ausgewertet, was auch `true` ergibt, da `c = true` ist. Abschließend wird `(a || true)` ausgewertet, was unabhängig vom Wert von a immer `true` ergibt.

3. Aufgabe:

a) Die exakte Darstellung direkt im Speicher ist von Prozessor zu Prozessor unterschiedlich. Laut *Java Language Specification, 2nd edition* hält sich Java in Bezug auf `float` und `double` jedoch stets an den Standard IEEE 754.

Zahlen vom Typ `float` werden nach diesem Standard folgendermaßen dargestellt:



Das erste Bit (von links) gibt das Vorzeichen an (0: positive Zahl, 1: negative Zahl). Die nächsten (von links) 8 Bits geben den Exponenten, die letzten 23 Bits die Mantisse an. Die Zahlen werden dabei normalisiert dargestellt, dass heißt im Binärsystem als $1,\ldots \cdot 2^a$.

Um auf diese Binärdarstellung zu kommen, wandeln wir den Betrag der gegebenen Dezimalzahlen wie gewohnt in das Binärsystem um und verschieben das Komma so, dass vor dem Komma genau eine 1 steht. Die Anzahl der Stellen, um die wir das Komma verschoben haben, ist der Wert a.[1] Zu a wird der Wert 127 (als „Bias“ bezeichnet) hinzuaddiert, das Ergebnis in Binärdarstellung bildet dann den Exponenten-Teil der internen Darstellung der Ausgangszahl.

Die ersten 23 Stellen nach dem Komma (in der normalisierten Darstellung) bilden die Mantisse der internen Darstellung. Achtung: Falls die 24. Stelle nach dem Komma gleich 1 ist, wird aufgerundet, also zu der aus den 23 Stellen nach dem Komma bestehenden Bitfolge 1 addiert.

Die Darstellung der vorgegebenen Zahlen nach IEEE 754 ist also wie folgt:

x1: 01110001010010011111001011001010
x2: 00111111000000000000000000000000
x3: 00111101110011001100110011001101
x4: 10111111000000000000000000000000
x5: 00001101101000100100001001100000

Die Bitfolge zur Speicherung dieser Darstellungen z. B. in einem Register kann jedoch völlig anders aussehen und ist vom Prozessor abhängig.

[1] Bei Zahlen >1 wird das Komma nach links verschoben, also ist a positiv. Bei Zahlen <1 wird das Komma nach rechts verschoben, also ist a negativ.

b) Das Programm liefert folgende Ausgabe:

```
Infinity
Infinity
1.6E308
```

Der 1. Ausdruck `(4.0 * d * 0.5)` wird, da keine Klammerung vorhanden ist, von links nach rechts ausgewertet. `(4.0 * d)` ergibt jedoch `3.2E308`, was außerhalb des darstellbaren Bereiches liegt.

Der 2. Ausdruck `(0.5 * (4.0 * d))` wird nicht von links nach rechts abgearbeitet, da die Klammerung eine andere Reihenfolge vorgibt. Die erste Berechnung `(4.0 * d)` führt wieder zu einer Bereichsüberschreitung.

Beim 3. Ausdruck `((0.5 * 4.0) * d)` wird zunächst der geklammerte Teilausdruck `(0.5 * 4.0)` ausgewertet (Ergebnis 2.0) und anschließend das Ergebnis mit `b` multipliziert. Die Berechnung `(2 * 8e+307)` ergibt `1.6E308`, welches eine mit `double` darstellbare Zahl ist.

Wir sehen anhand dieses Beispiels, dass das Assioziativgesetz für die Multiplikation

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

hier nicht gilt!

c) Der Typ `int` hat eine Länge von 32 Bits und kann damit Zahlen im Wertebereich von -2^{31} bis $2^{31}-1$ darstellen (das erste Bit wird für das Vorzeichen benötigt). Die interne Darstellung der Zahlen erfolgt im Zweierkomplement. Wir wollen nachweisen, dass für a,b,c vom Typ `int` gilt:

$$(a+b)+c = a+(b+c)$$

Seien x,y,z die Zweier-Komplementdarstellungen von a,b,c, also

$$\begin{array}{rcl} (x)_{2Kpl} = a & \Leftrightarrow & (x)_2 = \text{MOD}(a, 2^{32}) \\ (y)_{2Kpl} = b & \Leftrightarrow & (y)_2 = \text{MOD}(b, 2^{32}) \\ (z)_{2Kpl} = c & \Leftrightarrow & (z)_2 = \text{MOD}(c, 2^{32}) \end{array}$$

Wir wissen, dass gilt

$$\begin{array}{rcl} \text{MOD}(g+h,m) & = & \text{MOD}(g+\text{MOD}(h,m),m) \\ & = & \text{MOD}(\text{MOD}(g,m)+h,m) \\ & = & \text{MOD}(\text{MOD}(g,m)+\text{MOD}(h,m),m) \\ & = & g \oplus_m h \end{array}$$

Dies wenden wir nun auf die Addition von drei Summanden an. Dabei gilt natürlich das Assoziativgesetz für die „normale“ Addition ohne MOD.

$$
\begin{aligned}
(g \oplus_m h) \oplus_m k &= \mathrm{MOD}(\mathrm{MOD}(g+h,m)+k,m) \\
&= \mathrm{MOD}((g+h)+k,m) \\
&= \mathrm{MOD}(g+(h+k),m) \\
&= \mathrm{MOD}(g+\mathrm{MOD}(h+k,m),m) \\
&= g \oplus_m (h \oplus_m k)
\end{aligned}
$$

Angewandt auf die `int`-Addition in Java mit obigen Bezeichnern bedeutet das

$$
\begin{aligned}
(a+b)+c &= ((x)_2 \oplus_{2^{32}} (y)_2) \oplus_{2^{32}} (z)_2 \\
&= (x)_2 \oplus_{2^{32}} ((y)_2 \oplus_{2^{32}} (z)_2) \\
&= a+(b+c)
\end{aligned}
$$

4. Aufgabe:

Das Programm

```
public class Zahlenfolge {
  public static void main(String[] args) {
    float pf, rf;
    double pd, rd;
    int n;
    pf = 0.01f; pd = 0.01;
    rf = 3f; rd = 3;

    System.out.println("p0 (float)= " + pf
                       + ", p0 (double)= " + pd);
    for (n=1;n<=20;n++) {
      pf = pf + rf*pf*(1-pf);
      pd = pd + rd*pd*(1-pd);
      System.out.println("p" + n + " (float)= " + pf
                         + ", p" + n + " (double)= " + pd);
    }
  }
}
```

liefert folgende Ausgabe:

```
p0 (float) = 0.01,              p0 (double) = 0.01
p1 (float) = 0.0397,            p1 (double) = 0.0397
p2 (float) = 0.15407173,        p2 (double) = 0.15407173000000002
p3 (float) = 0.5450726,         p3 (double) = 0.5450726260444213
p4 (float) = 1.2889781,         p4 (double) = 1.2889780011888006
p5 (float) = 0.1715188,         p5 (double) = 0.17151914210917552
p6 (float) = 0.5978191,         p6 (double) = 0.5978201201070994
p7 (float) = 1.3191134,         p7 (double) = 1.3191137924137974
p8 (float) = 0.056273222,       p8 (double) = 0.056271577646256565
p9 (float) = 0.21559286,        p9 (double) = 0.21558683923263022
p10 (float) = 0.7229306,        p10 (double) = 0.722914301179573
p11 (float) = 1.3238364,        p11 (double) = 1.3238419441684408
p12 (float) = 0.037716985,      p12 (double) = 0.03769529725473175
p13 (float) = 0.14660022,       p13 (double) = 0.14651838271355924
p14 (float) = 0.521926,         p14 (double) = 0.521670621435246
p15 (float) = 1.2704837,        p15 (double) = 1.2702617739350768
p16 (float) = 0.2395482,        p16 (double) = 0.24035217277824272
p17 (float) = 0.7860428,        p17 (double) = 0.7881011902353041
p18 (float) = 1.2905813,        p18 (double) = 1.2890943027903075
p19 (float) = 0.16552472,       p19 (double) = 0.17108484670194324
p20 (float) = 0.5799036,        p20 (double) = 0.5965293124946907
```

Man sieht, dass `double` wesentlich genauer ist als `float`. Ab p_{16} unterscheiden sich die Werte bereits in der zweiten Stelle nach dem Komma!

5. Aufgabe:

```
import Prog1Tools.IOTools;
public class Zinseszins {

  public static void main(String [] args) {

    double anlage;
    double zins;
    double endbetrag;
    int laufzeit;

    anlage=IOTools.readDouble("Anzulegender␣Geldbetrag␣in␣Euro:␣");
    zins=IOTools.readDouble("Jahreszins␣(z.␣B.␣0.1␣fuer␣10␣Prozent):␣");
    laufzeit=IOTools.readInteger("Laufzeit␣(in␣Jahren):␣");

    endbetrag = anlage;

    int i=1;
    while (i<=laufzeit){
      endbetrag=endbetrag*(1+zins);
      System.out.println("Wert␣nach␣" + i + "␣Jahren:␣" + endbetrag);
      i++;
    }
```

```
    /* oder mit Zaehlschleife:
      for (int i=1;i<=laufzeit;i++){
        endbetrag=endbetrag*(1+zins);
        System.out.println("Wert nach " + i + " Jahren: " + endbetrag);
      }*/
  }
}
```

Interessant an der Zinseszinsberechnung ist der Zusammenhang mit der e-Funktion:

Das Kapital k_n nach einer Laufzeit von n Jahren mit einem Startkapital k_0 und einem jährlichen Zinssatz z ($0 \leq z \leq 1$) berechnet sich durch

$$k_n = k_0 \cdot (1+z)^n$$

Wenn das Kapital halbjährlich verzinst würde (natürlich mit den halben Zinsen) und die Zinsen gleich wieder angelegt werden würden, würde die Formel folgendermaßen lauten:

$$k_n = k_0 \cdot \left(1+\frac{z}{2}\right)^{2 \cdot n}$$

Analog bei täglicher Verzinsung:

$$k_n = k_0 \cdot \left(1+\frac{z}{365}\right)^{365 \cdot n}$$

Da

$$\lim_{m \to \infty} \left(1+\frac{x}{m}\right)^m = \mathrm{e}^x$$

gilt mit m als „Verzinsungshäufigkeit“

$$\begin{aligned} k_n &= k_0 \cdot \lim_{m \to \infty} \left(1+\tfrac{z}{m}\right)^{m \cdot n} \\ &= k_0 \cdot \mathrm{e}^{z \cdot n} \end{aligned}$$