Martin Götze, Bernhard Streit

5. Übung zu Informatik B

Tutorium: Marco Block, Fr 12–14 Uhr

1. **Vergleich & Wahrheit in Java**

   Betrachten Sie das folgende Codefragment

```
1  int [] x = {4,6,2,8,3};
2  int [] y = x;
3  x[3] = 7;
4  int [] z = (int []) y.clone();
5  z[0] = 3*2;
6  z[4] = 4;
7  boolean a1 = (y==z);
8  boolean a2 = (y==x);
9  boolean b1 = (y[4]==z[4]);
10 boolean b2 = (x[0]==z[0]);
11 boolean b3 = (x[3]==z[3]);
```

   Welche Werte haben danach die einzelnen Boolschen Variablen? Geben Sie kurze Begründungen.

   a1 hat den Wert **„false“**, da in Zeile 4 das Feld z als Kopie von y erzeugt wird, also sind y und z Referenzen auf verschiedene Felder, da in den auf Zeile 4 folgenden Zeilen an den Referenzen y und z nichts mehr verändert wird, bleibt das auch so (es werden ja nur noch die Feldinhalte geändert).

   a2 hat den Wert **„true“**, da in Zeile 2 y = x gesetzt wird, d.h. y und x verweisen auf dasselbe Feld, in den aus Zeile 2 folgenden Zeilen werden die Referenzen x und y nicht mehr verändert, da nur noch Feldinhalte verändert werden, also ist auch in Zeile 8 noch x == y.

   b1 hat den Wert **„false“**, da wegen Zeile 6 sicher z[4]=4 ist, y[4] aber ist gleich x[4] (siehe a2) und damit gleich 3, wegen $3 \neq 4$ ist somit b1 falsch.

   b2 hat den Wert **„false“**, da wegen Zeile 7 sicher z[0]=6 ist, es ist aber x[0] gleich 4, da x[0] seit der Initialisierung in Zeile 1 nicht mehr geändert weurde, also ist b2 falsch.

   b3 hat den Wert **„true“**, es ist nämlich ab Zeile 3 sicher x[3]=y[3]=7 (vgl. auch a2), da z in Zeile 4 als Kopie von y erzeugt wird ist dann auch z[3]=7, danach wird weder x[3] noch z[3] geändert, also ist b3 wahr.

2. **mystische Funktion**

   Betrachten Sie die folgende Definition einer Methode:

```
1  long mystic(int n) {
2      long N=1;
3      for( int i=2; i<= n; i++) {
4          int k=i;
5          int j=0;
6          while (k>1) {
7              k = k/2;
8              j++;
9          }
10         N=N*j;
11     }
12     return N;
13 }
```

   (a) Was berechnet diese Methode? Beschreiben sie den Rückgabewert mit einer kurzen Formel!

   (b) Schätzen Sie die Laufzeit (in Abhängigkeit vom Eingabeparameter n) ab, möglichst in der Form $t(n) = \Theta(\ldots)$.

   (c) Schreiben Sie eine Methode, die das gleiche leistet, aber nur **eine** while-Schleife benutzt.

   (d) Welcher Wert wird für $n = 1000$ berechnet werden, wenn man vorher die Anweisung `N = N * j;` durch `if (j>N) N=j;` ersetzt.

   (a) Um zu bestimmen, was die Methode mystic berechnet, betrachtet man zunächst die while-Schleife in den Zeilen 6–9 für einen festen Wert des for-Schleifen Zählers $2 \leq i \leq n$ aus Zeile 3:

   Sei also ein festes, aber beliebiges $2 \leq i \leq n$ vorgegeben. Man zeigt nun, dass j nach dem Durchlaufen der While Schleife gerade den Wert $\lfloor \log_2 i \rfloor$ hat. Wähle dazu $m \in \mathbb{N}_0$ maximal mit $2^m \leq i$, dann ist $m = \lfloor \log_2 i \rfloor$, man zeigt durch vollständige Induktion nach m, dass m gerade die Zahl ist, die angibt, wie oft die while-Schleife durchlaufen wird:

   - **Induktionsanfang** Sei $m = 1$ (das ist mindestens der Fall, denn $i \geq 2$), dann ist $i \in \{2, 3\}$, und damit vor dem Durchlaufen der while-Schleife wegen Zeile 4 auch $k \in \{2, 3\}$, nach dem ersten Durchlaufen von Zeile 7 ist dann also $k = 1$, d.h. die while-Schleife wird wegen der Bedingung in Zeile 6 kein weiteres Mal durchlaufen.
   - **Induktionsvoraussetzung** Für ein festes, aber beliebiges $m = \lfloor \log_2 i \rfloor$ gelte, dass die while-Schleife in den Zeilen 6–9 genau m mal durchlaufen wird.
   - **Insuktionsschluß** Sei also $m+1 = \lfloor \log_2 i \rfloor$, es sei etwa $i = 2^{m+1} + \nu$ mit einem $0 \leq \nu < 2^{m+1}$, vor dem ersten Durchlauf der while-Schleife ist dann wegen Zeile 4 sicherlich auch $k = 2^{m+1} + \nu$, nach dem ersten Durchlauf ist also wegen Zeile 7 sicher $k = 2^m + \mu$, dabei ist $\mu = \lfloor \frac{\nu}{2} \rfloor$ also $0 \leq \mu < 2^m$, also $\lfloor \log_2 k \rfloor = m$, d.h. nach Induktionsvoraussetzung wird die Schleife noch m weitere Male durchlaufen, insgesamt also $m + 1$ Mal, q.e.d.

   In der Schleife wird wegen Zeile 8 in der Variablen j gerade die Anzahl der Durchläufe gezählt, in Zeile 10 hat also j den Wert der Anzahl der

Schleifendurchläufe (also $m = \lfloor \log_2 i \rfloor$ nach obigem), da j in Zeile 5 mit 0 initialisiert wird.

Nun kann man leicht angeben, was mystic(n) ist: In Zeile 2 wird N mit 1 initialisiert, danach wird in der for-Schleife für jedes $2 \leq i \leq n$, N mit $\lfloor \log_2 i \rfloor$ multipliziert (falls $n \geq 2$, sonst wird die for-Schleife nicht durchlaufen). Hält man sich an die Konventition, das ein leeres Produkt den Wert 1 hat, so ist also für jeden int-Wert n immer

$$\text{mystic}(n) = \prod_{i=2}^{n} \lfloor \log_2 i \rfloor.$$

Eine solche Formel war aber anzugeben.

(b) Nun also zur Laufzeitabschätzung: Wir zählen alle Operationen mit 1, dann hat man, wenn man erneut die Methode betrachtet:

- In Zeile 1 wird die Methode nur definiert, keine Operation
- In Zeile 2: 1 Operation
- In Zeile 3: 1 Operation (Vergleich $2 \leq n$). Die Schleife wird $(n-1)$ mal durchlaufen, jeweils werden hier 2 Operationen aufgeführt (Erhöhen und vergleichen)
- In Zeile 4 & 5: Jeweils eine Operation
- In Zeile 6: Die Schleife wird $\lfloor \log_2 i \rfloor$ mal durchlaufen, jeweils eine 1 Operation (Vergleich)
- In Zeile 7 & 8: Je eine Operation
- In Zeile 10: 1 Operation
- In Zeile 12: 1 Operation

Insgesamt hat man also für einen Durchlauf der while-Schleife (bei festem i) gerade 3 Operationen, da sie $\lfloor \log_2 i \rfloor$ mal durchlaufen wird hat man also insgesamt $3 \cdot \lfloor \log_2 i \rfloor + 1$ Operationen (1 weiterer Vergleich beim letzten mal ohne Durchlauf), in der for-Schleife finden im inneren jeweils 5 Operationen statt, sie wird für $2 \leq i \leq n$ Durchlaufen, also hat man (die Zusätzlichen Operationen sind die Initialisierung und der Vergleich vor dem 1. Durchlauf) gerade

$$\sum_{i=2}^{n} \Big(5 + \big(3 \cdot \lfloor \log_2 i \rfloor + 1\big)\Big) + 2$$

Operationen, zusammen mit den beiden weiteren also gerade

$$\sum_{i=2}^{n} \Big(5 + \big(3 \cdot \lfloor \log_2 i \rfloor + 1\big)\Big) + 4$$

Operationen, man erhält:

$$\begin{aligned} \sum_{i=2}^{n} \Big(5 + \big(3 \cdot \lfloor \log_2 i \rfloor + 1\big)\Big) + 4 &= 6(n-1) + 4 + 3 \sum_{i=2}^{n} \lfloor \log_2 i \rfloor \\ &= \Theta(n) + 3 \sum_{i=2}^{n} \lfloor \log_2 i \rfloor \end{aligned}$$

Nun gilt aber für jedes $n \geq 2$:

$$\begin{aligned}
\sum_{i=2}^{n} \lfloor \log_2 i \rfloor &\leq \sum_{i=2}^{n} \log_2 i \\
&= \log_2 \prod_{i=2}^{n} i \\
&= \log_2 n! \\
&\leq \log_2 n^n \\
&= n \log_2 n
\end{aligned}$$

und damit ist

$$\sum_{i=2}^{n} \lfloor \log_2 i \rfloor = O(n \log_2 n)$$

Andererseits ist für $n \geq 2$ und jedes $1 \leq i \leq n$ wegen $0 \leq i-1$ und $0 \leq n-i$:

$$0 \leq (i-1)(n-i) = ni + i - n + i^2 = i(n+1-i) - n \iff n \leq i(n+1-i)$$

also gilt zunächst:

$$\begin{aligned}
n^n &= \prod_{i=1}^{n} n \\
&\leq \prod_{i=1}^{n} i(n+1-i) \\
&= \prod_{i=1}^{n} i \cdot \prod_{i=1}^{n} (n+1-i) \\
&= n! \cdot n!
\end{aligned}$$

und damit wegen der Monotonie von $\log_2$:

$$\begin{aligned}
n \log_2 n &= \log_2 n^n \\
&\leq \log_2 (n!)^2 \\
&= 2 \log_2 n! \\
&= 2 \log_2 \prod_{i=2}^{n} i \\
&= 2 \sum_{i=2}^{n} \log_2 i \\
&\leq 2 \sum_{i=2}^{n} \big(\lfloor \log_2 i \rfloor + 1 \big) \\
&\leq 2 \sum_{i=2}^{n} \big(\lfloor \log_2 i \rfloor + \lfloor \log_2 i \rfloor \big) \\
&= 4 \sum_{i=2}^{n} \lfloor \log_2 i \rfloor
\end{aligned}$$

Also zusammen

$$\sum_{i=2}^{n} \lfloor \log_2 i \rfloor = \Theta(n \log_2 n)$$

Also kann man die Anzahl der Operationen weiter abschätzen, man hat:

$$\begin{aligned}
\#\text{Operationen} &= \Theta(n) + 3\sum_{i=2}^{n} \lfloor \log_2 i \rfloor \\
&= \Theta(n) + 3\Theta(n \log_2 n) \\
&= \Theta(n \log_2 n)
\end{aligned}$$

da $n = o(n \log_2 n)$. Nimmt man nun an, dass jede Operation in etwa 1 Zeiteinheit braucht, so hat man für die Laufzeit $t(n) = \Theta(n \log_2 n)$, eine solche Abschätzung war aber gefragt.

(c) Die Idee dabei ist, anstatt die beiden Zähler j und i in verschiedenen Schleifen zu erhöhen, sie in einer einzigen zu erhöhen, dabei ist zu beachten, dass die alte while-Schleife nur durchlaufen wird, wenn $k > 1$ ist und $N = N * j$ erst nach Beendigung der while-Schleife ausgeführt wird, was man mit einer if-Abfrage erreichen kann. Man erhält:

```java
/** Bernhard Streit, Martin Götze
 *       5. Übung, 2. Aufgabe
 * eine Funktion die das Produkt aller 2er-Logarithmen (abgerundet)
 * von 2 bis zu n berechnet soll (anders als auf dem Übungszettel)
 * mit nur einer Schleife implementiert werden. */
class Mystic {

    /** führt die Mystic-Funktion für jedes
     * Argument aus und gibt die Ergebnisse
     * aus sowie die Überprüfung, die mittels
     * der Funktion pruefe ermittelt wird */
    public static void main(String[] argumente) {
        for (int i = 0; i < argumente.length; i++) {
            int zahl = Integer.parseInt(argumente[i]);
            System.out.println("mystic(" + zahl + ") = " + mystic(
                zahl));
            System.out.println("  richtig wäre: " + pruefe(zahl));
        }
    }

    /** Unsere Version der mystic-Funktion (mit nur einer
     * Schleife) */
    static long mystic(int n) {
        long N = 1;
        /* Vorher stand hier for (int i = 2; i <= n; i++) {,
            jetzt mit einer while-Schleife und 2 (!) Zählern: */
        int i = 2;
        int k = i;        // die beiden Lauf-variablen
        int j = 0;
        while (i <= n) {    //für die alte for-Schleife
            if (k>1) { // anstelle der alten while Schleife:
                k = k/2;
                j++;
            }
            else { // Jetzt wäre vorher die while-Schleife beendet
                N = N * j;
                k = ++i;
```

```
37                  j = 0;
38              }
39          }
40          return N;
41      }
42
43      /** Diese Funktion berechnet das Produkt aller 2er−Logatihmen
              von 2
44        * bis n, zur Überprüfung der Funktion von mystic */
45      static long pruefe(int n) {
46          long N = 1;
47          for (int i = 2; i <= n; i++) N = N * log2(i);
48          return N;
49      }
50
51      /** Berechnet der 2er−Logarihmus einer ganzen Zahl (normal
              gerundet) */
52      static int log2(int n) {
53          return (int) (Math.log((double) n) / Math.log(2.0));
54      }
55  }
```

Ein Testlauf des Programms lieferte die Ausgabe:

```
> java Mystic 1 2 3 4 5
mystic(1) = 1
   richtig  wäre: 1
mystic(2) = 1
   richtig  wäre: 1
mystic(3) = 1
   richtig  wäre: 1
mystic(4) = 2
   richtig  wäre: 2
mystic(5) = 4
   richtig  wäre: 4
> java Mystic 6 7 8 9 10 20
mystic(6) = 8
   richtig  wäre: 8
mystic(7) = 16
   richtig  wäre: 16
mystic(8) = 48
   richtig  wäre: 48
mystic(9) = 144
   richtig  wäre: 144
mystic(10) = 432
   richtig  wäre: 432
mystic(20) = 107495424
   richtig  wäre: 107495424
```

(d) Nach Teil (a) hat j in der zuändernden Zeile 10 gerade den Wert $\lfloor \log_2 i \rfloor$, N wird also mit der neuen Zeile nur dann geändert, wenn $\lfloor \log_2 i \rfloor$ größer ist als $\lfloor \log_2 l \rfloor$ für alle $l < i$. Das ist offenbar genau dann der Fall, wenn i eine Zweierpotenz ist, denn genau dann überschreitet $\log_2 i$ die nächste ganze Zahl, d.h. N hat immer den Wert $\log_2 l_i$, wo l_i die größte Zweierpotenz $\leq i$ ist, also gerade $\lfloor \log_2 i \rfloor$, am Ende von Mystik also $\lfloor \log_2 n \rfloor$, also liefert mystic jetzt:

$$\mathrm{mystic}_{\mathrm{neu}}(n) = \lfloor \log_2 n \rfloor$$

und damit

$$\mathrm{mystic}_{\mathrm{neu}}(1000) = \lfloor \log_2 1000 \rfloor = \log_2 512 = 9.$$

Das war aber zu bestimmen.

3. **Minimum — Maximum**

Schreiben Sie Pseudocode und ein Java-Programm zur Bestimmung des größten und des kleinsten Eintrags in einem Array von ganzen Zahlen. Analysieren Sie die Anzahl der durchgeführten Elementaroperationen.

Hinweis: Orientieren Sie sich am Pseudocode/Programm ArrayMax aus dem Goodrich/Tamassia–Buch auf Seite 101 (siehe http://datastructures.net).

Die Idee ist folgende: Hat man ersteinmal ein Maxima/Minima verschiedener Teile der Liste ermittelt, so muss man diese nur noch untereinander vergleichen und überprüfen, welches von ihnen das kleinste/größte ist. Man geht nun so vor: Zunächst für eine einelementige Liste, da ist es leicht, der einzige Eintrag ist Minimum und Maximum zu gleich, hat die Liste mehr als ein Element, so wird sie zunächst in Vorder– und Hinterteil aufgeteilt, dort werden Seperat Maxima und Minima bestimmt und diese werden dann verglichen, als Pseudocode:

```
Funktion minMax(Feld)
// Diese Funktion bekommt ein Feld ganzer Zahlen übergeben und
// bestimmt Maximum und Minimum, dazu bedient sie sich der
// Funktion MinMaxTeil, s.u.
// Die Rückgabe ist ein Feld der Form {Min, Max}

Gebe minMaxTeil(Feld, 0, Länge von Feld −1) zurück

Funktion minMaxTeil(Feld, VonIndex, BisIndex)
// Diese Funktion bestimmt Minimum und Maximum derjenigen Teilliste
// von Feld, die von VonIndex bis BisIndex (beides eingeschlossen)
// geht und zwar rekursiv nach dem Divide & Impera Prinzip,
// zurückgegeben wird ein zweielementiges Feld ganzer Zahlen,
// das Minimum und Maximum enthält.

Wenn (BisIndex = VonIndex) // nur ein Element, das ist leicht
    Min ← Feld[BisIndex]
    Max ← Min
Sonst Wenn (BisIndex = VonIndex + 1) // 2 Elemente
    Wenn (Feld[VonIndex] < Feld[BisIndex]
        Min ← Feld[VonIndex];
        Max ← Feld[BisIndex];
    Sonst
        Min ← Feld[VonIndex];
        Max ← Feld[BisIndex];
Sonst // mindestens 2 Elemente:
    Mitte ← (VonIndex + BisIndex) / 2
    vornMinMax ←minMaxTeil(Feld, VonIndex, Mitte)
    hintenMinMax ←minMaxTeil(Feld, Mitte+1, BisIndex)
        // Beachte, dass sicher Mitte < BisIndex
        // da VornIndex < BisIndex
      // stelle sicher, das der vordere Teil gerade
      // Länge hat:
    Wenn (Mitte − VonIndex !$\equiv 0 \pmod 2$!)
        Mitte ← Mitte + 1
        // tritt dies auf, so ist sicher
        // Mitte < BisIndex − 1, da es mindestens
        // drei Elemente gibt.
```

```
    Wenn (vornMinMax[0] < hintenMinMax[0])
    // Vergleich der Minima
        Min ← vornMinMax[0]
    Sonst
        Min ← hintenMinMax[0]
    // dito, für die Maxima:
    Wenn (vornMinMax[1] < hintenMinMax[1])
        Min ← vornMinMax[1]
    Sonst
        Min ← hintenMinMax[1]
Gebe das Array {Min, Max} zurück.
```

Man setzt die Vergleiche als wesentliche Elementaroperationen und erhält, wie im folgenden durch vollständige Induktion nach der Länge n des Feldes gezeigt wird, dass #Vergleiche $= \lceil \frac{3n}{2} \rceil - 2$:

- **Induktionsanfang** Betrachte also zunächst ein beliebiges Feld der Länge 1, beim Aufruf der Funktion minMaxTeil wird außer dem Feld noch 0 als erster und 0 als letzter Index übergeben, d.h. es ist VonIndex = BisIndex und damit wird kein Vergleich durchgeführt um Maximum und Minimum zu bestimmen, also gilt die Behauptung wegen
$$\left\lceil \frac{3 \cdot 1}{2} \right\rceil - 2 = 2 - 2 = 0.$$
Betrachte nun ein beliebiges Feld der Länge 2, beim Aufruf wird 0 als erster und 1 als letzter Index übergeben, es ist also die Differenz genau 1 und es wird genau ein Vergleich vorgenommen, die Behauptung gilt wegen:
$$\left\lceil \frac{3 \cdot 2}{2} \right\rceil - 2 = 3 - 2 = 1.$$
- **Induktionsvoraussetzung** Sei $n \geq 2$ beliebig aber fest und die Behauptung gelte für alle Liste einer Länge $\leq n$.
- **Induktionsschluß** Sei also eine Liste der Länge $n + 1$ gegeben, beim Aufruf von minMaxTeil ist VonIndex < BisIndex -1 (denn $n \geq 2$). Nun wird die Funktion minMaxTeil für Teile der Liste, die die Längen n_1 resp. n_2 haben sollen, aufgerufen, nach Induktionsvoraussetzung werden für diese beiden Aufrufe zusammen gerade
$$\left\lceil \frac{3 \cdot n_1}{2} \right\rceil - 2 + \left\lceil \frac{3 \cdot n_2}{2} \right\rceil - 2$$
Vergleiche gemacht, da n_1 gerade ist (dafür wird ja gerade gesorgt) ist auch $3n_1$ gerade und insgesamt werden mit den 2 zusätzlichen Verlgeichen gerade
$$\begin{aligned} & \left\lceil \frac{3 \cdot n_1}{2} \right\rceil - 2 + \left\lceil \frac{3 \cdot n_2}{2} \right\rceil - 2 + 2 \\ \overset{n_1 \text{ gerade}}{=} & \frac{3n_1}{2} + \left\lceil \frac{3 \cdot n_2}{2} \right\rceil - 2 \\ \overset{n_1 \text{ gerade}}{=} & \left\lceil \frac{3 \cdot (n_1 + n_2)}{2} \right\rceil - 2 \\ = & \left\lceil \frac{3 \cdot n}{2} \right\rceil - 2 \end{aligned}$$
Vergleiche benötigt, das war aber die Behauptung.

Nun aber noch der Algorithmus, der nach Hinweis an den Goodrich angelehnt ist, man geht einfach die Liste elementweise durch und vergleicht mit den festgehaltenen Maxima und Minima, wobei im schlechtesten Fall offenbar jedes der letzten $n-1$ Elementen mit den beiden derzeitigen festgehaltenen Werten verglichen wird, also $2(n-1) = 2n-2$ Vergleiche benötigt werden:

```
1  Funktion MinMaxGood(Feld)
2  // Diese Funktion bestimmt Minimum und Maximum einer Liste ganzer Zahlen und gibt diese
3  // als Feld {Min, Max} zurück.
4
5  Min ← Feld[0]
6  Max ← Feld[0]
7  Für i ← 1 Bis (Länge von Feld) − 1
8      Wenn Feld[i] < Min     // neues Minimum
9          Min ← Feld[i]
10     Sonst
11         Wenn Feld[i] > Max // neues Maximum
12             Max ← Feld[i]
13 Gib {Min,Max} zurück.
```

Es folgt nun (wie im Goodrich) eine Anaylse dieses Algorithmus, die *alle* Operationen (im schlechtesten/besten Fall) beachtet:

- Zeilen 5, 6: Jeweils eine Zuweisung, d.h. zusammen 2 Operationen
- Zeile 7: eine Initialisierung (Variable i), n Vergleiche (jeweils i mit $n-1$), $n-1$ mal Erhöhung von i, wo n die Länge des Feldes bezeichnet.
- Zeile 8: 1 Vergleich ($n-1$-Mal, da in Schleife), bestenfalls ist der Vergleich immer wahr, denn dann folgt nur noch eine weitere Anweisung in der Schleife, schlechtestenfalls immer falsch.
- Zeile 9: eine Zuweisung ($n-1$ Mal, nur im besten Fall ausgeführt)
- Zeile 11: 1 Vergleich ($n-1$ Mal, nur im schlechtesten Fall ausgeführt), dann immer wahr ($\rightarrow$ mehr Anweisungen)
- Zeile 12: eine Zuweisung ($n-1$ Mal, nur im schlechtesten Fall ausgeführt)
- Zeile 13: Rückgabe, also 1 Operation

Zusammen erhält man also mindestens (d.h. im besten Fall, die Liste ist absteigend sortiert)

$$1 + 1 + 2n + 2(n-1) + 1 = 4n + 1$$

Operationen, schlechtestenfalls hat man (d.h. die Liste ist echt aufsteigend sortiert)

$$1 + 1 + 2n + 3(n-1) + 1 = 5n$$

Operationen.

Hier nun beide Programme in Java umgesetzt:

```
/** Martin Götze, Bernhardt Streit
  * 5. Übung
  *
  * Programm zur Bestimmung des Maximums und des Minimus einer
  * Liste von genzen Zahlen auf zwei Arten, einmal rekursiv mit
  * möglichst wenigen Vergleichen, einmal iterativ nach dem Vor–
  * bild aus dem Goodrich. */
```

```java
class MinMax {

    /** Der Test für die Funktionen, die Eingabe Parameter
      * werden als Liste ganzer Zahlen an beide Algorithmen
      * übergeben */
    public static void main(String[] argumente) {

        // Eingabe entgegennehmen
        int [] zahlen;
        if (argumente.length > 0) {
        zahlen = new int[argumente.length];
        for (int i = 0; i < argumente.length; i++)
            zahlen[i] = Integer.parseInt(argumente[i]);
        }
        else { // keine Zahlen übergeben
            zahlen = new int []{2, 5, 6, 12, 24};
        }

        // Minimum und Maximum bestimmen
        int [] minmax1 = minmax(zahlen);
        int [] minmax2 = minmaxGood(zahlen);

        // Ausgabe
        System.out.println("Im Feld");
        for (int i = 0; i < zahlen.length; i++)
            System.out.print(zahlen[i] + ",");
        System.out.println();
        System.out.println("ganzer Zahlen wurden ermittelt als Min/Max: ");
        System.out.println("mit dem 3n/2-Algorithmus: " + minmax1[0] + "/" + minmax1[1]);
        System.out.println("nach dem Vorbild aus dem Goodrich: " +
            minmax2[0] + "/" + minmax2[1]);
    }

    /** Unser Algorithmus, Minimum und Maximum werden durch divide &
      * impera rekursiv ermittelt, dabei werden nur |3n/2| - 2 Vergleiche
      * benötigt.
      * Das Feld wird in zwei Felder aufgeteilt, wobei das vordere gerade
      * Länge oder die Länge 1 hat (das nur, wenn das hintere auch 1 lang)
      * ist, vgl. den Pseudocode,
      * zurückgegeben wird {Min, Max} */
    static int [] minmax(int[] zahlen) {
        int [] minimumMaximum;
        if (zahlen.length == 0) {
            System.out.println("Fehler: keine Zahlen da" +
                "die sortiert werden können");
            minimumMaximum = new int[]{0,0};
        }
        else {  // sortieren mit Hilfsmethode
            minimumMaximum = minmaxTeil(zahlen, 0, zahlen.length - 1);
        }
```

```
        return minimumMaximum;
    }

    /** Die Hilfsmethode für unseren Algotithmus, sie bestimmt Minimum und
      * Maximum einer Teilliste, die von vonIndex bis bisIndex geht */
    private static int[] minmaxTeil(int[] feld, int vonIndex, int bisIndex) {
        int min=0, max=0;
        if (vonIndex == bisIndex) { // alles klar
            min = max = feld[vonIndex];
        }
        else if (vonIndex == bisIndex - 1) {
            // Zwei Elemente, also:
            if (feld[vonIndex] < feld[bisIndex]) {
                min = feld[vonIndex];
                max = feld[bisIndex];
            }
            else { // feld[vonIndex] ist größer
                min = feld[bisIndex];
                max = feld[vonIndex];
            }
        }
        else { // mindestens drei Elememte, divide & impera
            int mitte = (vonIndex + bisIndex) / 2;
            // Vorne soll gerade Länge haben
            if ((mitte - vonIndex) % 2 == 0) mitte++;
            /* Beachte, dass die Länge des vorderen Teils gerade
               mitte - vonIndex + 1 ist */
            int[] minmaxVorn = minmaxTeil(feld, vonIndex, mitte);
            int[] minmaxHinten = minmaxTeil(feld, mitte + 1, bisIndex);
            // Minima vergleichen:
            if (minmaxVorn[0] < minmaxHinten[0])
                min = minmaxVorn[0];
            else
                min = minmaxHinten[0];
            // Maxima vergleichen:
            if (minmaxVorn[1] < minmaxHinten[1])
                max = minmaxHinten[1];
            else
                max = minmaxVorn[1];
            // Das war's.
        }
        return new int[]{min, max};
    }

    /** Der MinMax-Algorithmus nach dem Vorbild des Goodirch, die Elemente
      * werden nacheinander durchlaufen und verglichen, es werden im schlech-
      * testen Fall 2n-2 Vergleiche benötigt. */
    static int[] minmaxGood(int[] zahlen) {
        int min = 0, max = 0;
        if (zahlen.length == 0) {
```

```
            System.out.println("Es wurden keine Zahlen übergeben,"+
                "die man vergleichen könnte");
        }
        else { // Los gehts
            min = max = zahlen[0];
            for (int i = 1; i < zahlen.length; i++) {
                if (zahlen[i] < min) min = zahlen[i];
                else if (zahlen[i] > max) max = zahlen[i];
            }
        }
        return new int[]{min, max};
    }
}
```

Und die Ausgabe unseres Testlaufs:

```
> java MinMax
Im Feld
 2,5,6,12,24,
ganzer Zahlen wurden ermittelt als Min/Max:
mit dem 3n/2-Algorithmus: 2/24
nach dem Vorbild aus dem Goodrich: 2/24
> java MinMax 2 1 4
Im Feld
 2,1,4,
ganzer Zahlen wurden ermittelt als Min/Max:
mit dem 3n/2-Algorithmus: 1/4
nach dem Vorbild aus dem Goodrich: 1/4
> java MinMax 2 4 10 -4 1
Im Feld
2,4,10,-4,1,
ganzer Zahlen wurden ermittelt als Min/Max:
mit dem 3n/2-Algorithmus: -4/10
nach dem Vorbild aus dem Goodrich: -4/10
> java MinMax -2 4 -10 -4 1
Im Feld
-2,4,-10,-4,1,
ganzer Zahlen wurden ermittelt als Min/Max:
mit dem 3n/2-Algorithmus: -10/4
nach dem Vorbild aus dem Goodrich: -10/4
> java MinMax -1
Im Feld
-1,
ganzer Zahlen wurden ermittelt als Min/Max:
mit dem 3n/2-Algorithmus: -1/-1
nach dem Vorbild aus dem Goodrich: -1/-1
> java MinMax 0 0
Im Feld
 0,0,
ganzer Zahlen wurden ermittelt als Min/Max:
mit dem 3n/2-Algorithmus: 0/0
nach dem Vorbild aus dem Goodrich: 0/0
```

Scheint zu klappen.

4. **Pascal**

   Schreiben Sie ein Java-Programm, das Ihnen die ersten 10 Zeilen des Pascalschen Dreiecks auf dem Bildschirm ausgibt. Achten Sie auf Effizienz. Die einzelnen Zeilen sollen linksbündig sein.

   Zunächst das Programm:

```java
/** Bernhard Streit, Martin Götze
  *    5. Übung, 4. Aufgabe
  *
  * Programm zur Berechnung der ersten Zeilen des Pascalschen
  * Dreiecks, standardmäßig werden 10 berechnet und ausgeben,
  * es kann jedoch eine andere Anzahl durch Übergabe eines
  * Parameters gesichert werden. */
class Pascal {

    /** Die main-Methode: Es wird der (falls vorhanden)
      * Parameter eingelesen und für den Aufruf der Pascal-
      * Methode zur Durchführung der eigentlichen Aufgabe
      * gesorgt. */
    public static void main(String[] argumente) {

        // Eingabe verarbeiten:
        int n=0;
        if (argumente.length > 0)
            n = Integer.parseInt(argumente[0]);
        // Defaultwert setzen:
        if (n<=0) n = 10;

        // Und das Pascalsche Dreieck:
        System.out.println("Die ersten " + n + " Zeilen des" +
            " Pascalschen Dreiecks: ");
        pascal(n);
    }

    /** Die pascal-Methode: Sie gibt die ersten n Zeilen des
      * Pascalschen Dreiecks aus, dabei wird jeweils eine Zeile
      * berechnet und dann ausgegeben */
    private static void pascal(int n) {
        int[] zeile, alteZeile;
        // Die alteZeile wird zur Berechnung der nächsten
        // Zeile benötigt.
        zeile = new int[n]; // nte Zeile <-> nEinträge
        alteZeile = new int[n];
        for (int i = 1; i <= n; i++) {
            // i-te Zeile berechnen:
            zeile[0] = 1; // das erste Element ist immer 1
            for (int j = 1; j < i-1; j++) {
                zeile[j] = alteZeile[j-1] + alteZeile[j];
            }
            zeile[i-1] = 1; // das letzte auch

            //i-te Zeile ausgeben und zwischenspeichern:
            for (int j = 0; j < i; j++) {
```

```
                System.out.print(zeile[j] + " ");
                alteZeile[j] = zeile[j];
            }
            System.out.println(); // und die nächste ...
        }
        // Das wars.
    }
}
```

und die Testläufe:

```
> java Pascal
Die ersten 10 Zeilen des Pascalschen Dreiecks:
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
> java Pascal 1
Die ersten 1 Zeilen des Pascalschen Dreiecks:
1
> java Pascal 12
Die ersten 12 Zeilen des Pascalschen Dreiecks:
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
1 11 55 165 330 462 462 330 165 55 11 1
```

Scheint zu klappen.