Universität Mannheim
Lehrstuhl für Praktische Informatik IV
Prof. Dr. W. Effelsberg
Christoph Kuhmünch, Gerald Kühne

Praktische Informatik I
WS 1999/2000

Übungsblatt 8 Ausgabe: Mi, 15.12.99 Abgabe: Di, 21.12.99, 18 Uhr

Aufgabe 1: Rekursive Algorithmen in Java [14 Punkte]

Entwickeln Sie einen Algorithmus, der einen Weg aus einem Labyrinth sucht, und implementieren Sie den Algorithmus in einem Java-Programm (unter Umständen gibt es mehrere Lösungen, es reicht aber, eine Lösung anzugeben).

```
 1 import java.io.*;
 2 import java.util.StringTokenizer;
 3
 4 /*
 5   Name:      Labyrinth
 6   Author:    Christoph Kuhmuench, Gerald Kuehne
 7
 8   Mit Hilfe dieser Klasse laesst sich ein Labyrinth aus einer Datei
 9   einlesen.  Das eingelese Labyrinth kann anschliessend daraufhin
10   untersucht werden, ob von einem vorgegebenen Startpunkt aus der
11   Ausgang gefunden werden kann.
12  */
13 public class Labyrinth
14 {
15   // nachfolgende Konstanten kapseln die Zeichen, mit deren Hilfe
16   // das Labyrinth definiert wird.
17   private final static char weg     = ' ';
18   private final static char wand    = '*';
19   private final static char start   = '+';
20   private final static char markiert= '0';
21
22   // in diesem array wird das Labyrinth abgespeichert
23   private char[][] laby;
24   // Hoehe und Breite des Labyrinthes
25   private int hoehe;
26   private int breite;
27   // Startpunkt im Labyrinth, von dem aus der Ausgang gesucht wird
28   private int startZeile;
29   private int startSpalte;
30   private int ausgangZeile;
31   private int ausgangSpalte;
32
33   /*
34     Name: Konstruktor von Labyrinth
35
36     Beschreibung: Initialisiert die Attribute des Objektes
37    */
38   public Labyrinth() {
39     laby         = null;
40     hoehe        = -1;
41     breite       = -1;
42     startSpalte  = -1;
43     startZeile   = -1;
44     ausgangZeile = -1;
45     ausgangSpalte= -1;
```

```
46   }
47
48
49   /*
50      Die nachfolgenden beiden Methoden geben die Werte der
51      Attribute ausgangZeile bzw. ausgangSpalte zurueck.
52   */
53
54   public int leseAusgangZeile() {
55     return ausgangZeile;
56   }
57
58   public int leseAusgangSpalte() {
59     return ausgangSpalte;
60   }
61
62   /*
63     Name:      einlesen
64     Parameter:
65       - String dateiname // Name der einzulesenden Datei
66
67     Rueckgabewert:
68       - boolean // true, wenn einlesen erfolgreich war
69
70     Beschreibung:
71       Die Methode liest ein Labyrinth aus einer Textdatei ein. Es
72       wird ein zweidimensionales Feld erzeugt (laby), in dem das
73       eingelesene Labyrinth abgelegt wird.  Weiterhin werden die
74       Attribute hoehe, breite, startZeile und startSpalte
75       entsprechend gesetzt.
76    */
77   public boolean einlesen(String dateiname) {
78     char           c= ' ';
79
80     FileReader     fr   = null;
81     BufferedReader in   = null;
82
83     try {
84       fr= new FileReader(dateiname);
85       in= new BufferedReader(fr);
86     } catch(FileNotFoundException fnfe) {
87       System.out.println("File not found");
88     }
89
90     try {
91       String          s = in.readLine();
92       StringTokenizer st= new StringTokenizer( s );
93       breite= Integer.parseInt( st.nextToken() );
94       s= in.readLine();
95       st= new StringTokenizer( s );
96       hoehe= Integer.parseInt( st.nextToken() );
97
98       laby= new char[hoehe][breite];
99
100      for (int zeile=0; zeile < hoehe; zeile++) {
101        s= in.readLine();
102        for (int spalte=0; spalte < breite; spalte++) {
103          c= s.charAt(spalte);
104          switch(c) {
105          case start:
106            startSpalte         = spalte;
107            startZeile          = zeile;
```

```
108             laby[zeile][spalte] = weg;
109             break;
110           case wand:
111           case weg:
112             laby[zeile][spalte]= c;
113             break;
114           default:
115             System.out.println("Falsche Eingabe!");
116             return false;
117           }
118         } // end for spalte
119       } // end for zeile
120     } catch(java.io.IOException ioe) {
121       System.out.println("IO Exception");
122     }
123     return true;
124   }
125
126   /*
127     Name: toString()
128
129     Beschreibung:
130       Die Methode gibt den Inhalt des zweidimensionalen Feldes
131       laby als String zurueck.
132    */
133   public String toString() {
134     String result="";
135
136     for (int zeile= 0; zeile < hoehe; zeile++) {
137       for (int spalte= 0; spalte < breite; spalte++) {
138         result+= laby[zeile][spalte];
139       }
140       result+= "\n";
141     }
142     return result;
143   }
144
145
146   /*
147     Name: starteSuche()
148
149     Beschreibung:
150       Die Methode startet die Suche nach dem Ausgang aus dem
151       Labyrinth. Dazu wird die private rekursive Methode
152       berechne Weg benutzt.
153    */
154   public boolean starteSuche() {
155     return berechneWeg(startZeile, startSpalte);
156   }
157
158
159   /*
160     Name: berechneWeg
161     Parameter:
162       // Koordinaten des Startpunktes der suche
163       - int zeile
164       - int spalte
165
166     Rueckgabewert:
167       - boolean // true, falls ein Weg gefunden wurde.  Der
168                 // gefundene Weg ist dann markiert.
169
```

```
170     Beschreibung:
171       Sucht rekursiv in laby nach dem Ausgang: In jedem Schritt
172       wird ueberprueft, ob die aktuelle Position ein noch nicht
173       besuchter Weg ist.  Falls ja wird ueberprueft, ob vielleicht
174       schon der Ausgang erreicht worden ist (in diesem Fall wird
175       true zurueckgegeben).  Falls der Ausgang noch nicht erreicht
176       worden ist und die aktuelle Position ein unbesuchter Weg ist,
177       wird der Reihe nach ueberprueft, ob der Ausgang erreicht
178       wird, indem man nach oben, unten, links oder rechts geht.
179       Dies geschieht, indem sich die Methode selbst erneut mit
180       verschobener Position aufruft.
181   */
182   private boolean berechneWeg(int zeile, int spalte) {
183
184     //    System.out.println(this);
185
186     // Abbruchbedingung
187     if( laby[zeile][spalte] != weg ) {
188       return false;
189     };
190
191
192     laby[zeile][spalte] = markiert;
193
194     if(spalte == 0 || spalte == breite-1 ||
195        zeile == 0 || zeile == hoehe-1) {
196       ausgangZeile = zeile;
197       ausgangSpalte= spalte;
198       return true;      /* gefunden */
199     };
200
201     /* nach oben ? */
202     if(berechneWeg(zeile-1, spalte)) {
203       return true;
204     };
205     /* nach unten ? */
206     if(berechneWeg(zeile+1, spalte)) {
207       return true;
208     };
209     /* nach links ? */
210     if(berechneWeg(zeile, spalte-1)) {
211       return true;
212     };
213     /* nach rechts ? */
214     if(berechneWeg(zeile, spalte+1)) {
215       return true;
216     };
217
218     /* nix gefunden, Markierung weg */
219     laby[zeile][spalte] = weg;
220
221     return false;
222   }
223
224   /*
225     Name: main
226
227     Paramter:
228       - argv[0] Name der Datei, die ein Labyrinth enthaelt.
229
230     Beschreibung:
231       Hauptfunktion, die ein Labyrinth-Objekt erzeugt und
```

```
232       "benutzt".
233    */
234   public static void main(String[] argv) {
235     Labyrinth labyrinth= new Labyrinth();
236
237     if ( !labyrinth.einlesen(argv[0]) )
238       System.exit(1);
239
240     if ( !labyrinth.starteSuche() ) {
241       System.out.println("Es gibt kein Weg zum Ausgang");
242     } else {
243       System.out.println("Ausgang gefunden!");
244       System.out.println(labyrinth.leseAusgangZeile()  + " Zeile");
245       System.out.println(labyrinth.leseAusgangSpalte() + " Spalte");
246       System.out.println(labyrinth);
247     }
248   }
249 }
```

Aufgabe 2: Fibonacci-Zahlen [6 Punkte]

In Übungsblatt 7 wurde die Fibonacci-Funktion wie folgt definiert:

$$fib(n) = \begin{cases} 1 & , \quad \textit{falls } n = 1 \\ 1 & , \quad \textit{falls } n = 2 \\ fib(n-1) + fib(n-2) & , \quad \textit{sonst} \end{cases} \tag{1}$$

Der nachfolgende Algorithmus ist aus der Musterlösung von Blatt 7 entnommen (Fehler-Abfrage wurde absichtlich ausgelassen):

```
long fibo (long n) {
  if (n <= 2) {
    return 1;
  } else {
    return fibo (n-1) + fibo (n-2);
  }
}
```

(a) [3 Punkte] Zeigen Sie, daß der oben abgebildete Algorithmus für alle Integer-Werte $n \mid n \geq 2$ terminiert!

Bei jedem Aufruf werden höchsten zwei neue rekursive Aufrufe erzeugt. Weiterhin wird bei jedem rekursiven Aufruf der Funktionsparameter n um 1 bzw. 2 dekrementiert. Die Rekursion endet, wenn $n \leq 2$. Da n rekursiv dekrementiert wird, wird diese Abbruchbedingung für alle gültigen n erreicht.

(b) [3 Punkte] Beweisen Sie, daß der oben abgebildete Algorithmus korrekt ist!

Beweis durch vollständige Induktion:

Induktionsbasis:	$n = 1, n = 2$ *klar*
Induktionsannahme:	*Der Algorithmus liefert das gewünschte Ergebnis für* $n-1, n-2 \mid n \geq 2$.
Induktionsschluß:	*zz. Der Algorithmus funktioniert auch für n.*
	Der Aufruf von $fibo(n-1)$ *und* $fibo(n-2)$ *liefert laut Induktionsnahme das korrekte Ergebnis für* $n \geq 2$. *Wird nun der Algorithmus mit* $fibo(n)$ *aufgerufen, so liefert dieser als Ergebnis die Summe aus* $fibo(n-1)$ *und* $fibo(n-2)$. *Dies ist aber laut oben angegebener Definition (1) korrekt.*