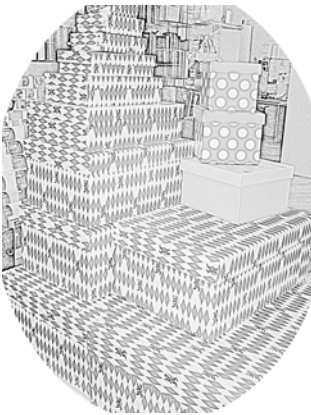


Verschachtelte Schachteln

In diesem Kapitel stellen wir ein Problem der Dynamischen Programmierung vor. Nach der vollständigen Beschreibung des Problems folgen die Problemanalyse und der Entwurf der Lösung, der in einem kurzen Pseudocode mündet. Daraus entwickeln wir ein Java-Programm, das anschließend bezüglich Java/OOP-Techniken analysiert wird.

Problembeschreibung



Wir betrachten eine n -dimensionale Schachtel. Wenn wir zwei Dimensionen annehmen, kann das Paar $(2, 3)$ eine Schachtel mit der Länge 3 und der Breite 2 repräsentieren. In der dreidimensionalen Welt kann das Tripel $(4, 8, 9)$ eine Schachtel der Länge 4, der Breite 8 und der Höhe 9 repräsentieren. Es mag schwierig sein, sich eine Schachtel mit mehr als drei Dimensionen vorzustellen, aber wir können damit operieren. Wir sagen, dass eine Schachtel $A = (a_1, a_2, \dots, a_n)$ in die Schachtel $B = (b_1, b_2, \dots, b_n)$ **passt**, wenn es eine Permutation π von $\{1, 2, \dots, n\}$ gibt, so dass $a_{\pi(i)} < b_i$ für alle $i \in \{1, 2, \dots, n\}$. Das heißt, dass man die

Reihenfolge der Werte einer Schachtel beliebig ändern

darf. Wir wollen die längste Folge von Schachteln finden, die ineinander passen. Die Schachteln C_1, C_2, \dots, C_k stellen eine solche Folge dar, wenn die Schachtel C_i in die Schachtel C_{i+1} ($1 \leq i < k$) passt.

Zum Beispiel passt die Schachtel $A = (2, 6)$ in die Schachtel $B = (7, 3)$, weil die Abmessungen von A permutiert werden können zu $A = (6, 2)$ und jede Abmessung ist kleiner als die entsprechende Abmessung von B . Die Schachtel $A = (9, 5, 7, 3)$ passt nicht in die Schachtel $B = (2, 10, 6, 8)$, weil keine Umstellung der Werte von A diese Bedingung erfüllt. Aber die Schachtel $C = (9, 5, 7, 1)$ passt in die Schachtel B , weil ihre Abmessungen zu $(1, 9, 5, 7)$ permutiert werden können und jede ist kleiner als die entsprechende Abmessung in B .

Eingabe: In der Datei *schachteln.in* gibt es eine Folge von Schachteln. Jede Folge beginnt mit einer Zeile, die die Anzahl der Schachteln k und deren Abmessungen n beschreibt. Jede der folgenden k Zeilen beinhaltet die n Abmessungen der jeweiligen Schachtel.

Ausgabe: Wie gesagt, wir müssen eine maximale Folge von Schachteln finden, die ineinander passen. Für den Fall, dass es mehrere solcher maximalen Folgen gibt, wird

nur eine von ihnen ausgegeben, wie in *schachteln.out* zu sehen ist. Die maximale Dimension einer Schachtel ist 250, die minimale ist 1. Die maximale Anzahl von Schachteln in einer Sequenz ist 300. Wir nehmen an, dass die *Eingabedaten* korrekt sind!

Beispiel:

schachteln.in	schachteln.out
5 2	Laenge: 4
3 7	-----
8 10	3 1 4 5
5 2	*****
12 7	
21 18	Laenge: 4
8 6	-----
5 2 20 1 30 10	7 2 5 8
23 15 7 9 11 3	*****
40 50 34 24 14 4	
9 10 11 12 13 14	Laenge: 5
31 4 18 8 27 17	-----
44 32 13 19 41 19	5 4 2 7 9
1 2 3 4 5 6	*****
80 37 47 18 21 9	
9 5	
7 14 2 1 3	
49 80 15 50 10	
90 53 17 60 11	
4 3 2 15 10	
1 2 3 4 5	
6 7 8 9 10	
89 53 17 60 11	
3 2 1 14 9	
92 54 65 19 15	

(ACM Internet Programming Contest 1990, Problem D. Stacking Boxes)

Problemanalyse und Entwurf der Lösung

Satz 1. Gegeben seien die *n*-dimensionalen Schachteln $A = (a_1, a_2, \dots, a_{n-1}, a_n)$ und $B = (b_1, b_2, \dots, b_{n-1}, b_n)$ mit der Eigenschaft $a_i \leq a_{i+1}, b_i \leq b_{i+1}$ für alle *i* von 1 bis *n*-1 (die Dimensionen sind aufsteigend sortiert). Die Schachtel *A* passt dann und nur dann in die Schachtel *B*, wenn $a_i < b_i$ für alle $i \in \{1, 2, \dots, n\}$.

Beweis. Wir benutzen den Beweis durch Widerspruch. Wir stellen uns vor, dass *A* in *B* passt und es ein $k \in \{1, 2, \dots, n\}$ gibt, so dass $a_k \geq b_k$. Wir betrachten das kleinste *k* mit diesen Bedingungen: $a_i < b_i$ für alle $i \in \{1, 2, \dots, k-1\}$ und $a_k \geq b_k$. Weil *a_i* eine aufsteigende Folge ist, folgt, dass auch $a_i \geq b_k$ für alle $i \in \{k+1, \dots, n\}$. Die einzige Möglichkeit, dass an Position *k* die Ungleichung $a_k < b_k$ erfüllt wird, ist der Tausch von *a_k* mit einem der Werte $\{a_1, a_2, \dots, a_{k-1}\}$. Den betreffenden Wert bezeichnen wir mit *j*. In diesem Fall gilt

an der Stelle j die Ungleichung $a_j \geq b_j$, also passt A nicht in B . Widerspruch! In der anderen Richtung ist die Implikation per Definition wahr. \square

Ein erster Schritt zum Entwurf eines Algorithmus ist dann das aufsteigende Sortieren der Dimensionen für jede Schachtel. Der zweite Schritt ist das lexikographische Sortieren aller Schachteln, mit der Speicherung der ursprünglichen Stelle. Damit Schachtel A in Schachtel B passt, ist es notwendig (aber nicht ausreichend!), dass A sich in dieser Folge vor B befindet (eine Schachtel an einer kleineren Stelle *kann* in eine Schachtel an einer größeren Stelle passen, aber umgekehrt ist das unmöglich!). Nach diesen Vorarbeiten reduzieren wir das Problem auf die Bestimmung der längsten aufsteigenden Teilfolge. Die Vergleichsbedingung „ \leq “ wird jetzt zu „*passt*“. Für die erste Sequenz aus der Eingabedatei werden die folgenden Schritte ausgeführt:

1. Aufsteigendes Sortieren der Dimensionen für jede Schachtel	2. Lexikographisches Sortieren der Schachteln mit Speicherung der ursprünglichen Stellen
<div style="display: flex; align-items: center; justify-content: center;"> <div style="border: 1px solid black; padding: 5px; margin-right: 10px;"> 3 7 8 10 5 2 12 7 21 18 </div> <div style="font-size: 2em; margin-right: 10px;">→</div> <div style="border: 1px solid black; padding: 5px;"> 3 7 8 10 2 5 7 12 18 21 </div> </div>	<div style="display: flex; align-items: center; justify-content: center;"> <div style="border: 1px solid black; padding: 5px; margin-right: 10px;"> 3 7 8 10 2 5 7 12 18 21 </div> <div style="font-size: 2em; margin-right: 10px;">→</div> <div style="border: 1px solid black; padding: 5px;"> 2 5 (3) 3 7 (1) 7 12 (4) 8 10 (2) 18 21 (5) </div> </div>

Die Bestimmung der längsten aufsteigenden Teilfolge ist ein klassisches Problem der Dynamischen Programmierung, und wird auch in Kapitel 16 behandelt. Die maximal aufsteigende Teilfolge mit der Beziehung „*passt*“ ist $(2, 5) \rightarrow (3, 7) \rightarrow (7, 12) \rightarrow (18, 21)$, und sie ist die einzige mit der Länge 4. Wir schreiben die ursprünglichen Positionen der Schachteln (3 1 4 5) in die Ausgabedatei.

Der Algorithmus

Wir betrachten n Objekte C_1, C_2, \dots, C_n vom Typ *Schachtel*, die ihre Dimensionen und ursprünglichen Stellen kennen. Für jedes i betrachten wir eine Folge von Schachteln mit der letzten Schachtel C_i . Dafür bilden wir zwei Vektoren $v[]$ und $vPred[]$. Die Länge der Folge ist $v[i]$ und der Index der vorletzten Schachtel ist $vPred[i]$ ($vPred[i]=1$, wenn $v[i]=1$). Also:

- $v[i]$ ist die maximale Länge einer korrekten, ineinander passenden Schachtelfolge, deren letzte Schachtel C_i ist. Formal schreiben wir:
 $v[1] \leftarrow 1$ (die erste Schachtel kann keine andere beinhalten, die Teilfolge beinhaltet also nur dieses Element),
 $v[i] \leftarrow 1 + \max\{v[j] \mid j < i \text{ und } C_j \text{ passt in } C_i\}$ (C_j ist die vorletzte Schachtel der Schachtelfolge).

- $vPred[i]$ beinhaltet den Index j mit der Bedingung, dass die Schachtel C_j die Vorgängerschachtel in der maximalen Teilfolge ist, die mit der Schachtel C_i endet (wenn eine Schachtel keinen Vorgänger hat, dann ist dieser Wert -1):
 $vPred[1] \leftarrow -1$ (die erste Schachtel hat keinen Vorgänger),
 $vPred[i] \leftarrow j, v[j]$ maximal mit ($j < i$ und C_j passt in C_i).

Wenn es mehrere Folgen mit maximaler Länge gibt, dann nehmen wir die erste.

Die Vektoren $v[]$ und $vPred[]$ werden sequenziell befüllt. Wir nutzen die globale Variable $imax$ zur Speicherung des aktuellen optimalen Index. Wenn die Länge $v[i]$ für die aktuelle Stelle größer als der Wert $v[imax]$ ist, dann wird $imax$ mit i aktualisiert. Nun können wir den Pseudocode des Algorithmus formulieren.

ALGORITHM_VERSCHACHTELTE_SCHACHTELN

```

1. Read Boxes  $C_1, C_2, \dots, C_n$ 
2. For ( $i \leftarrow 1, n$ ; step 1) Execute
    Sort_Dimensions( $C_i$ )
    End_For
3. Sort_Lexikographical( $C_1, C_2, \dots, C_n$ )
4.  $v[1] \leftarrow 1, vPred[1] \leftarrow -1, imax \leftarrow 1$ 
5. For ( $i \leftarrow 2, n$ ; step 1) Execute
    5.1.  $v[i] \leftarrow 1, vPred[i] \leftarrow -1$ 
    5.2. For ( $j \leftarrow 1, i-1$ ; step 1) Execute
        If ( $C_j$  passt in  $C_i$  AND  $v[j]+1 > v[i]$ ) Then
             $v[i] \leftarrow v[j]+1$ 
             $vPred[i] \leftarrow j$ 
        End_If
    End_For
    5.3. If ( $v[i] > v[imax]$ ) Then
         $imax \leftarrow i$ 
    End_If
Ende_For
6. recoverBoxesSubstring ( $C[1..n], vPred[], imax$ )
END_ALGORITHM_VERSCHACHTELTE_SCHACHTELN

```

Die Komplexität des Algorithmus ist $O(n^2m + nm \log m)$, wobei n die Zahl der Schachteln und m die Dimension einer Schachtel ist (aus Schritt 2 folgt $O(nm \log m)$, weil n Schachteln sortiert sind; aus Schritt 5 folgt $O(n^2)$, weil es zwei verschachtelte

for-Schleifen mit der Länge n gibt; in der zweiten *for*-Schleife gilt für die *passt*-Bedingung $O(m)$.

Der Algorithmus *rekonstruiereTeilfolge* () ermittelt die optimale Schachtelteilfolge auf der Basis des Vektors *vPred*[] und des optimalen Index *imax* rekursiv:

```

ALGORITHM_rekonstruiereTeilfolge (C[1..n], vPred[], imax)
  String strReturn;
  currIndex  $\leftarrow$  imax
  While ( currIndex  $\geq$  0 ) Do
    strReturn.insertAtBeginning (Original_Position(C[currIndex]));
    currIndex  $\leftarrow$  vPred[currIndex]
  End_While
  return strReturn
END_ ALGORITHM_ rekonstruiereTeilfolge (C[1..n], vPred[], i)

```

Die Komplexität dieses Algorithmus ist linear $O(n)$.

Das Programm

Um das beschriebene mathematische Modell zu implementieren, schreiben wir die Klasse *CBox*, die Attribute wie *index* (die ursprüngliche Position in der Eingabedatei) und das Array *dimensions* beinhaltet. Die Methoden *getIndex()*, *compareTo()* und *fitIn()* bearbeiten die Elemente des abstrakten Typs *CBox*.

```

import java.io.*;
import java.util.*;

public class P01_BoxInBox {

    private static class Box implements Comparable<Box> {
        private int dimensions[];
        private int index;

        public Box(int boxIndex, int boxDimensions[]) {
            this.index = boxIndex;
            this.dimensions = boxDimensions;
        }

        public int compareTo(Box otherBox) {
            assert
                this.dimensions.length == otherBox.dimensions.length :
                    "Schachteldimensionen sind nicht einheitlich!";
            int rt = 0;
            for (int i = 0;

```

```

        i < this.dimensions.length && rt == 0; i++) {
            rt = this.dimensions[i] - otherBox.dimensions[i];
        }
        return rt;
    }

    public int getIndex() {
        return index;
    }

    public boolean fitIn(Box otherBox) {
        assert
            this.dimensions.length == otherBox.dimensions.length :
                "Schachteldimensionen sind nicht einheitlich!";
        boolean fit = true;
        for (int i = 0; fit && i < this.dimensions.length; i++) {
            fit = this.dimensions[i] < otherBox.dimensions[i];
        }
        return fit;
    }
}

public static void main(String args[]) {

    Scanner scanner = null;
    PrintWriter out = null;
    try {
        scanner = new Scanner(new File("schachteln.in"));
        out = new PrintWriter(new File("schachteln.out"));
        while (scanner.hasNextInt()) {
            Box boxes[] = new Box[scanner.nextInt()];
            int numDimensions = scanner.nextInt();
            for (int i = 0; i < boxes.length; i++) {
                int boxDimensions[] = new int[numDimensions];
                for (int j = 0; j < numDimensions; j++) {
                    boxDimensions[j] = scanner.nextInt();
                }
                Arrays.sort(boxDimensions);
                boxes[i] = new Box(i, boxDimensions);
            }
            Arrays.sort(boxes);

            int v[] = new int[boxes.length];
            Arrays.fill(v, 1);
            int vPred[] = new int[boxes.length];
            Arrays.fill(vPred, -1);
            int indexMax = 0;

            for (int i = 1; i < boxes.length; i++) {
                Box currBox = boxes[i];

```

```

        for (int j = 0; j < i; j++) {
            if (boxes[j].fitIn(currBox) && v[j] + 1 > v[i]) {
                v[i] = v[j] + 1;
                vPred[i] = j;
            }
        }
        if (v[i] > v[indexMax]) {
            indexMax = i;
        }
    }

    out.print("Laenge: ");
    out.println(v[indexMax]);
    StringBuilder bf = new StringBuilder();
    for (int currIdx = indexMax; currIdx >= 0;
         currIdx = vPred[currIdx]) {
        bf.insert(0, ' ').insert(0,
                                (boxes[currIdx].getIndex() + 1));
    }
    out.println(bf);
    out.println("*****");

    }
} catch (Throwable th) {
    th.printStackTrace();
} finally {
    if (scanner != null) {
        scanner.close();
    }
    if (out != null) {
        out.close();
    }
}
}
}

```

Die Programmanalyse

Datenabstraktion. Der abstrakte Datentyp *Box* ist die Java-Darstellung des Konzeptes einer *Schachtel* aus der Problembeschreibung. Auf diese Weise lässt sich ein wichtiges Prinzip der OOP realisieren, das man Kapselung (Geheimnisprinzip) nennt: Der Zugriff auf die Daten zum Lesen oder Ändern kann nur mit Hilfe von Methoden erfolgen. Ein unbeabsichtigtes Ändern des Indexes oder der Dimensionen ist nicht möglich.

Konstruktoren. In Java ist ein Konstruktor eine Methode ohne Rückgabewert und trägt denselben Namen wie die Klasse, die ihn beherbergt. Konstruktoren werden automatisch ausgeführt, wenn neue Objekte der Klasse angelegt werden, dürfen eine

beliebige Anzahl an Parametern haben und können überladen werden. Die Anweisung *new* veranlasst den Compiler dazu, anhand der Parameterliste den richtigen Konstruktor mit den Laufzeitwerten aufzurufen. Wir verwenden in unserem Programm einen Konstruktor, der beide Instanzvariablen initialisiert:

```
public Box(int boxIndex, int boxDimensions[]) {  
    this.index = boxIndex;  
    this.dimensions = boxDimensions;  
}
```

So erzeugen wir in der *main()*-Methode ein Objekt vom Typ *Box*:

```
boxes[i] = new Box(i, boxDimensions);
```

Wenn für eine Klasse keine Konstruktoren existieren, erzeugt der Compiler automatisch einen *default*-Konstruktor ohne Parameter, der wiederum lediglich den parameterlosen Konstruktor der Superklasse aufruft. Gibt es wenigstens einen Konstruktor in der Klasse, erzeugt der Compiler keinen *default*-Konstruktor.

Polymorphismus. Der Term *polymorph* stammt aus dem Griechischen und seine Bedeutung ist: „mehrere Formen haben“. Polymorphismus in Java bedeutet, dass mehrere Methoden denselben Namen haben dürfen. Manchmal haben diese Methoden verschiedene Formalparameter-Listen (überladene Methoden, engl. *overloading*) und in anderen Fällen haben sie identische Formalparameter-Listen und identische Rückgabetypen (überlagerte Methoden, engl. *overriding*). In unserem Programm verwenden wir die überladene Methode *java.util.Arrays.sort()*, einmal mit einem Parameter vom Typ *int[]* und einmal mit einem Parameter vom Typ *Box[]*.

Die statische lokale Klasse *Box*. Die Klasse *Box* ist eine statische lokale Klasse (engl. *static inner class*) von *P01_BoxInBox*. Sie ist mit dem Attribut *static* versehen, weil sie keinen Verweis auf die instanzierende Klasse und keinen Zugriff auf deren Membervariablen braucht. Die Klasse *Box* ist *private*, weil es nicht erlaubt ist, sie von außen zu instanziiieren.

Das Interface *java.lang.Comparable*. Klassen, die nur aus abstrakten, öffentlichen Methoden und Konstanten bestehen und keine Konstruktoren besitzen, bezeichnet man als Interfaces. Diese werden mit dem Schlüsselwort *interface* deklariert und sie bilden Eigenschaften ab, die sich auf Klassen beziehen können, die in verschiedenen Hierarchien beheimatet sind. Mit Hilfe der Interfaces lässt sich das Prinzip der Mehrfachvererbung in Java realisieren. Einige Beispiele für Interfaces in Java: *java.lang.Cloneable*, *java.lang.Runnable*, *java.util.EventListener*, *java.io.Serializable*, *java.util.Collection*, *java.lang.Iterable*, *java.util.Queue*, *java.util.Formattable*, *java.util.Map*. Eine Möglichkeit, um

Objekte nach individuellen Kriterien zu sortieren, stellt die Methode `sort(Object[] a)` aus der Klasse `Arrays` dar, die Teil des Pakets `java.util` ist:

```
Arrays.sort (boxes) ;
```

Dafür müssen die Elemente das Interface `Comparable` aus dem Paket `java.lang` implementieren:

```
public int compareTo(Object o)
```

Dieses Interface beinhaltet nur die Methode `compareTo()`, die auf den Instanzen dieser Klasse eine „natürliche Ordnung“ definiert, auf Grund derer sie z. B. mit `sort`-Methoden sortiert werden können. `compareTo()` liefert

< 0, wenn das aktuelle Element **vor** dem zu vergleichenden liegt,
0, wenn das aktuelle Element und das zu vergleichende **gleich** sind und
> 0, wenn das aktuelle Element **hinter** dem zu vergleichenden liegt,
zurück.

Wir sehen auch, wie die Methode `Arrays.sort(int[] a)` das gegebene Array aufsteigend sortiert.

Die Klasse `java.util.Arrays` beinhaltet Methoden, die Arrays von primitiven Datentypen und Objekten sortieren, durchsuchen, vergleichen, `hashen`, kopieren, füllen und deren Größe anpassen. `java.util.Arrays` beinhaltet auch die Methode `asList()`, die ein Array als Liste repräsentiert. Sie wirft die Ausnahme `NullPointerException`, wenn die spezifizierte Referenz Null ist. Wir verwenden in unserem Programm die Methoden `Arrays.sort()` und `Arrays.fill()`.

Ausnahmen behandelt man in Java mit der *try-catch*-Anweisung. Allgemein sieht sie so aus:

```
try {  
    Anweisung;  
    ...  
}  
catch (Ausnahmetyp_1 x1) {  
    Anweisung;  
    ...  
}  
catch (Ausnahmetyp_2 x2) {  
    Anweisung;  
    ...  
}  
...
```

```

catch (Ausnahmetyp_n xn) {
    Anweisung;
    ...
}
[
finally {
    ...
}
]

```

Die Anweisungen, die sich im **try**-Block befinden, können bei der Ausführung zu einer Ausnahme des Typs *Ausnahmetyp_i* führen. Wenn das passiert, wird der normale Programmablauf gestoppt und mit dem Code aus dem **catch**-Block weitergeführt, der die Ausnahme mit dem korrespondierenden *Ausnahmetyp* abfängt. Die Fehlerobjekte sind Instanzen der Klasse *Throwable* oder einer ihrer Unterklassen. Unter anderem stellt *Throwable* die Methode *printStackTrace()* bereit, die einen Auszug des Laufzeitstacks zurückgibt. Die **finally**-Klausel in der **try-catch**-Anweisung ist optional. Sie wird immer ausgeführt (falls vorhanden), wenn man in einen **try**-Block eintritt, ob nun vorher ein **catch**-Block gegriffen hat oder nicht, daher ist sie prädestiniert dafür, Aufräumarbeiten vorzunehmen (z. B. Ressourcen freizugeben). Wir verwenden sie in allen Programmen des Buches dazu, um die Input- und Outputstreams zu schließen.

Die Klasse *StringBuilder*. Die Lösung geben wir mittels der Klasse *StringBuilder* aus dem Paket *java.lang* aus, die in Java 5 eingeführt wurde. Diese Klasse ist API-kompatibel mit *StringBuffer*, aber ihre Methoden sind nicht synchronisiert und daher schneller. Deswegen nutzt man sie gern dann, wenn man eine dynamische Zeichenkette innerhalb eines einzigen Threads bearbeitet. Die wichtigsten Methoden sind *append()* und *insert()*, die jeden Datentyp akzeptieren: *append()* fügt den Parameter am Ende der Zeichenkette hinzu, *insert()* fügt die Zeichen an einer gegebenen Stelle ein.

Die Klasse *Scanner* aus dem Paket *java.util* liest in allen unseren Programmen die Eingabedaten ein. Auch sie wurde mit Java 5 vorgestellt und sie ist einfacher zu bedienen als z. B. die ältere Klasse *java.io.FileReader*. Eine Instanz von *Scanner* fungiert als einfacher Textscanner, der primitive Datentypen und Strings parsen kann. Die übliche Verwendung:

- Das Erzeugen eines Scanner-Objekts mit dem *File*-Objekt-Konstruktor als Argument, wie in unserem Programm:

```

Scanner scanner = new Scanner(new File("schachteln.in"));

```

(eine Ausnahmebehandlung ist zwingend, also befindet sich diese Instanziierung innerhalb eines **try-catch**-Blocks)

Grundlegende Algorithmen mit Java
Lern- und Arbeitsbuch für Informatiker und
Mathematiker

Logofătu, D.

2014, XVI, 324 S. 115 Abb., Softcover

ISBN: 978-3-8348-1972-7