34. Bundeswettbewerb Informatik
3. Aufgabe
Dokumentation

TEAM „Der LAMABÄR“

30. November 2015

Inhaltsverzeichnis

1 Lösungsidee

Im Wesentlichen zielt diese Aufgabe darauf ab einen Algorithmus zu finden, der für eine beliebige natürliche Zahl alle möglichen Darstellungen der Selben als Summe finden soll. Es gibt dabei nur zwei Einschränkungen: Die Anzahl der Summanden ist in dem Fall die Anzahl der Behälter und die maximale Größe eines jeden Summanden entspricht dem maximalen Fassungsvermögen des dazugehörigen Behälters.
Die Aufgabe alle möglichen Kombinationen für sieben Flaschen und zwei Behältern mit einem Fassungsvermögen von drei und fünf kann man also auch folgendermaßen darstellen:

- 7=3+4
- 7=2+5

In der Informatik wird dieses Problem als Paritionierungsproblem bezeichnet.

Die Idee meines Algorithmus besteht im Wesentlichen darin, dass ein baumähnlicher, gerichteter Graph erzeugt wird und das „Backtracking“-Verfahren daran angewandt wird, also jeder mögliche Pfad bzw. in dem Fall Kombination rekursiv durchgegangen wird und der Wert eines jeden einzelnen Knoten einer gerade durchlaufenden Kombination dabei summiert wird. Dabei gibt es in jeder „Stufe“ des Baumes, welche einem Behälter entspricht, so viele Knoten, wie Flaschen in den jeweiligen Behälter passen. Die Knoten auf jeder Stufe sind von 0 bis maximales Fassungsvolumen des Behälters durchnummeriert; Diese Zahl entspricht dem Wert des Knotens. Jeder Knoten zeigt, sofern es nicht der letzte Behälter ist, auf alle Knoten des nächsten Behälter, sodass man durch die Backtracking Methode alle möglichen Kombinationen „errechnen“ kann. Die Anzahl der Flaschen ist für die Erzeugung des Graphen irrelevant, da es nur bei der Rekursion benötigt wird, um zu überprüfen, ob der Wert einer Kombination der Anzahl an Flaschen entspricht oder nicht.
Dazu ein Beispiel. Es gibt drei Behälter mit einem Fassungsvolumen von je eins, drei und vier Flaschen. In der Abbildung 1 ist der daraus entstehende Graph zu diesem Beispiel skizziert.

Der logische Ablauf des rekursiven Algorithmus, der jede mögliche Kombination durchläuft, ist in der Abbildung 2 dargestellt. Wie der gerichtete Graph erstellt wurde, wird im 2. Kapitel näher erläutert, da es kein Bestandteil der Idee ist.

Abbildung 1: beispielhafter Graph

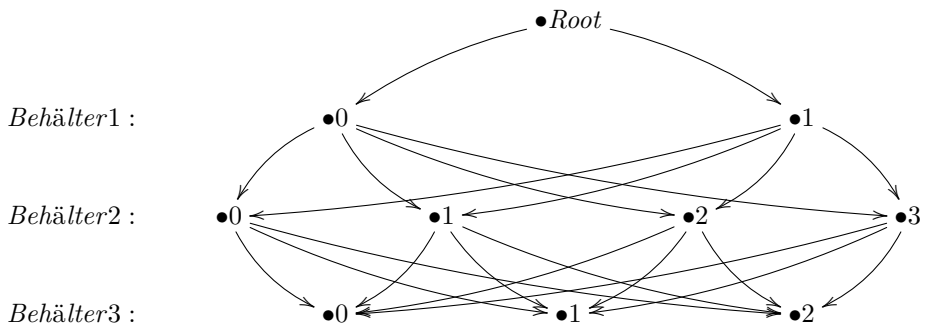


Abbildung 2: Nassi-Shneiderman-Diagramm zum Paritionierungsproblem

procedure Paritionierung(rootKnoten,wert,stufe: int)

wichtige Variablen:

FBehälterarray

{Ein dynamisches Array , in der für jeden Behälter das Fassungsvermögen angegeben ist}

FFlaschenanzahl

{}

FBehälteranzahl

{}

FmaxFlaschenInputproBehälter

{ Array : Gibt an, wie viele Flaschen noch in die Behälter, die unter dem i-ten behälter sind, hineinpassen}

wert = FFlaschenanzahl

WAHR	FALSCH
Zähler ← *Zähler* + 1 // *Kombination gefunden*	∅
Braucht nicht mehr weiterzusuchen.	

wert > FFlaschenanzahl

WAHR	FALSCH
Exit // *keine Kombination in diesem Zweig mehr möglich*	∅

FmaxFlaschenInputproBehälter[stufe]+wert < FFlaschenanzahl

WAHR	FALSCH
Exit // *keine Kombination in diesem Zweig mehr möglich*	∅

starte für jeden Zeiger, ausgehend vom aktuellen rootKnoten, folgenden Code (rekursiv):

Paritionierung(aktuell ausgewählter Zeiger vom rootKnoten, Wert des Knotens auf den der Zeiger zeigt + aktuellen Wert)

1.1 Laufzeitkomplexität

Ich habe mich hier für den Algorithmus des „Backtrackings“ und nicht für die „Tiefensuche“ entschieden, da ersteres effizienter ist, da erkannt wird, wenn eine weitere Berechnung an einem bestimmten Knoten nicht mehr sinnvoll ist, da keine Lösung mehr möglich ist, wenn z.B. der errechnete Wert größer als die Anzahl der Flaschen ist. Mein Algorithmus wurde außerdem insoweit optimiert, als dass er merkt, dass das Weiterrechnen von einem bestimmten Knoten aus nicht mehr sinnvoll ist, da der Wert zu niedrig ist, als dass es noch zu einer Lösung kommen kann. Dazu gibt es ein Array, in dem für jede Stufe, also für jeden Behälter, genannt wird, wie viele Flaschen maximal in den unter mir liegenden Behältern noch hineinpassen würden. Diese beiden Optimierungen verändern allerdings leider nichts an der Tatsache, dass die Ordnung dadurch nicht verändert wird. Zur Ermittlung der Ordnung habe ich einmal den BestCase in der Gleichung 1 und den WorstCase in der Gleichung 2 und Gleichung 3 ermittelt.

$$Pfadanzahl = 1 \tag{1}$$

Zur Erläuterung: Im besten Fall haben wir genau null Flaschen oder genau so viele Flaschen, wie die Summe der Fassungsvermögen aller Behälter ist, die maximale Anzahl an Flaschen, da hier einer der beiden Optimierungen greifen, und haben dementsprechend eine Ordnung 1, wenn wir uns auf die Anzahl der Pfade beziehen, da alle anderen Pfade durch die „Backtracking“-Methode nicht betreten werden. Also gilt $\mathcal{O}(1)$.
Der WorstCase wäre der Fall, wenn die Flaschenanzahl genau die Hälfte der Summe der Fassungsvermögen aller Behälter ist. Dann würden beide Optimierungen nur teilweise greifen und am geringsten Bewirken, da sie nur bei geringen oder hohen Flaschenanzahlen sehr wirksam sind.
i=Fassungsvermögen eines speziellen Behälters; V_i=Volumen des i-ten Behälters; n=Anzahl der Behälter:

$$Pfadanzahl = \frac{\prod_{i=1}^{n}(a_i + 1)}{2} \tag{2}$$

Der Term mit der Produktformel setzt sich mit der Annahme auseinander, dass die letzte Optimierung nicht greift und der WorstCase der Fall ist, und die maximal mögliche Flaschenanzahl der schlimmste Fall wäre. Dann wäre der Average-Fall die obengenannte Formel, welche mit beiden Optimierungen nun mal der schlimmste Fall ist.
Der exponentielle Zusammenhang wird durch einen Spezialfall deutlich: Alle Behälter n haben das gleiche Fassungsvermögen V:

$$Pfadanzahl = \frac{n^{V+1}}{2} \tag{3}$$

Im WorstCase gilt also $\mathcal{O}(n^x)$. Genau bei dieser Laufzeitanalyse sieht man den Vorteil des „Backtrackings“ gegenüber der „Tiefensuche“, da selbst im WorstCase durch die Optimierungen erstere effizienter ist als letztere und - abgesehen von den beiden einfachen ressourcenunbelastenden If-Abfrage, ob die Flaschenanzahl größer als der aktuell errechnete Wert ist bzw. eine Lösung aufgrund einer zu geringen Flaschenanzahl an der aktuellen Position mehr möglich ist, die schließlich bei jeder Rekursionstiefe durchgeführt wird - in jeden anderen Fall ist die Rechenleistung deutlich humaner und man könnte z.B. auch fünf Behälter mit je 300 Fassungsvermögen bei nur zwei Flaschen bzw. 288 Flaschen in einer akzeptablen Zeit lösen.

Nichtsdestotrotz besteht hier ein exponentieller Zusammenhang (s. Beispiele), was mit der Komplexitätsklasse des Paritionierungsproblems (NP-vollständig) übereinstimmt, was impliziert, dass es keinen effizienteren Algorithmus mit z.B. $\mathcal{O}(n)$ oder gar $\mathcal{O}(\log n)$ gibt. Jegliche Verbesserung des Codes würde die Steigung der exponentielle Funktion nur um ein Faktor verringern, sie kann jedoch nicht das exponentielle Wachstum verhindern.

Meine Laufzeitanalyse geht in dem Fall nur auf die Anzahl der durchlaufenden Pfade ein, jedoch nicht auf die Anzahl der durchlaufenden Knoten, was durchaus aussagekräftiger wäre, da bei einem unausbalancierten Graphenbaum es mehr Pfade gibt als Knoten durchlaufen werden; bei einem ausbalancierten Graphenbaum werden jedoch weitaus mehr Knoten bei der gleichen Pfadanzahl durchlaufen. In dem Fall würde der zusätzliche Rechenaufwand die Anzahl der Knoten zu berechnen, was die Komplexität der Formel deutlich erhöhen würde, nicht im Einklang mit dem Nutzen stehen. Dieser ist nämlich nicht ein exaktes Ergebnis zu bekommen, sondern eine sehr gute Näherung, was die Pfadanzahl durchaus angibt.
Als einen weiteren Optimierungsfaktor, der allerdings erst bei der Betrachtungsweise mit den durchlaufenden Knoten deutlich wird und daher nur exemplarisch und nicht durch eine Formel gezeigt werden kann, ist die Sortierung nach Fassungsvermögen der Behälter vor der eigentlichen Rekursion. Hierbei stellt sich die Frage,

was effizienter ist: Aufsteigend oder absteigend oder ob für die Beantwortung dieser Frage die Flaschenanzahl in Relation zu dem maximalen Fassungsvermögen aller Behälter eine Rolle spielt oder nicht.
Dazu einmal ein exemplarischer Graph mit drei Behältern mit einem Fassunsgvermögen von je 2, 3 und 4:

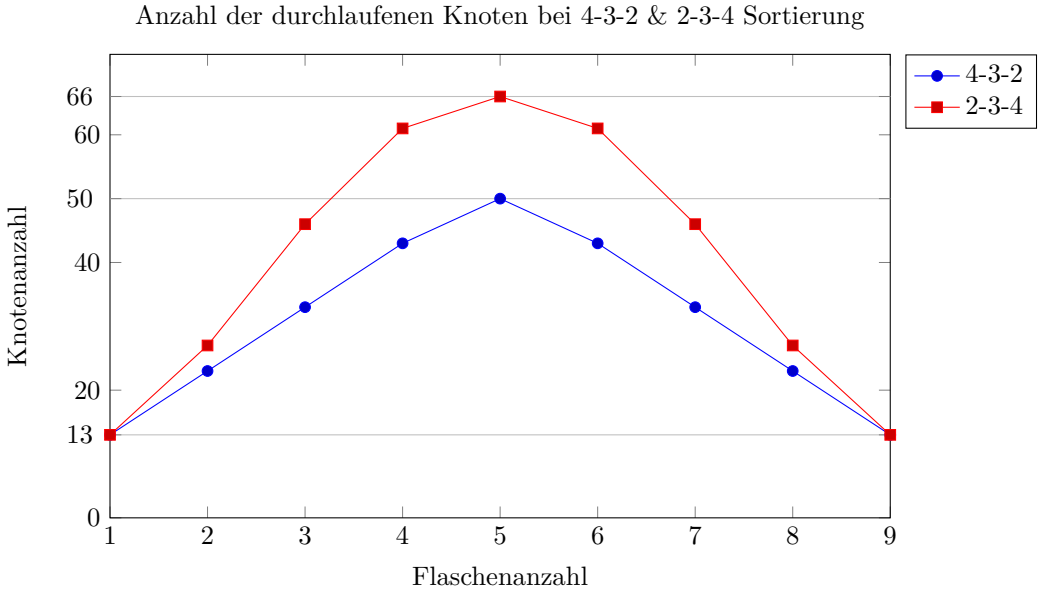


Abbildung 3: Abhängigkeit der Sortierung bei der Laufzeit

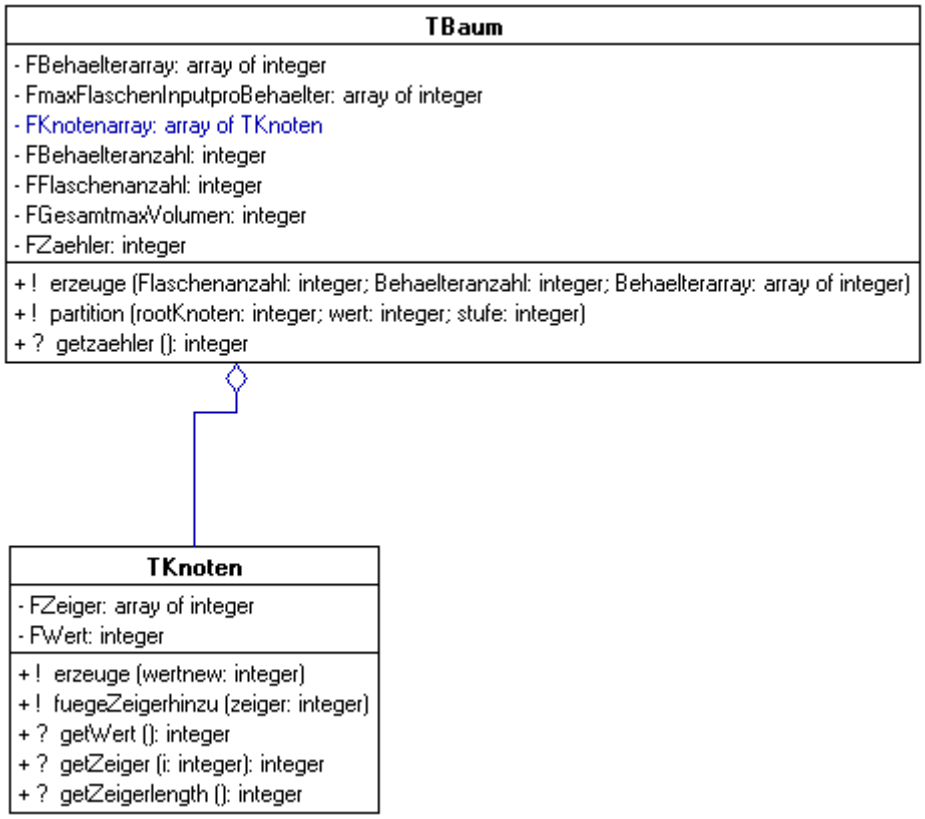
Was man aus dem Graphen exemplarisch erkennen kann, ist zum einen, dass sich meine Annahme zum Worst-Case sich bestätigt hat (dieser ist bei der Hälfte von der Summe aller Fassungsvermögen). Zum Zweiten, dass die Angabe der Pfadanzahl nur ungenau ist, da es auf die Reihenfolge der Behälter ankommt und sich so eine Amplitude von bis zu 18 Knoten in diesem Fall bildet. Zum Letzteren, dass die Sortierung 4-3-2 effizienter ist, was allerdings mit meinen Optimierungen zusammenhängt, da ohne die letzte Optimierung (Erkennung eines zu niedrigen Wertes, als dass man fortfahren könnte und noch ein Ergebnis finden könnte) beide Methoden sich nichts tun. Offensichtlich ist die 4-3-2 Sortierung in den Bereichen effizient, wo die anderen beiden Optimierungen, die in der Rekursion mittels If-Clauses verwendet werden, ineffizient sind. In den Bereichen, wo die 4-3-2 Sortierung ineffizient ist, macht das nichts, da da die beiden Optimierungen greifen.

2 Umsetzung

Für die Umsetzung habe ich das Modell-View-Controller(MVC)-Prinzip verwendet, wobei ich View und Controller der Einfachheit halber zu einer Instanz vereint habe. Hinzu kommt eine objektorientierte Programmierung. Von dem View-Controller werden die Flaschenanzahl, Behälteranzahl und das Volumen eines jeden Behälters über die Konsole abgefragt und mit den Werten ein Objekt der Klasse TBaum erzeugt. Diese erstellt auch sogleich den gerichteten Graphen. Danach führt der Benutzer die Procedure „parition“ aus, welche dann wiederum die Rekursion ausführt. Nach dem Durchlauf der Rekursion wird durch den aufgerufenen Getter „getzaehler“ die Anzahl aller möglichen Kombinationen in der Konsole ausgegeben. Das Objekt TBaum greift dafür auf die Klasse TKnoten zurück, um alle Knoten des Baumes zu erzeugen. Für jeden Knoten wird ein Objekt der Klasse TKnoten erstellt, welche dann in einem dynamischen Array des Typs TKnoten gespeichert werden. Jedes Objekt der Klasse TKnoten hat einen Wert (z.B. 4 Flaschen) und ein oder mehrere Zeiger auf seine „Kinder“, also auf die Knoten, die dem nächsten Behälter zugehörig sind. Gibt es keinen nächsten Behälter, so bleibt diese Variable leer. Zur Veranschaulichung kann das UML-Diagramm in Abbildung 4 verwendet werden.

Im 1. Kapitel habe ich bereits den eigentlichen Algorithmus, also die Rekursion, beschrieben, es fehlte aber noch der Algorithmus, der den Baum erstellt. Dazu habe ich noch ein Nassi-Shneidermann-Diagramm in Abbildung 5 erstellt. Der Algorithmus besteht im Wesentlichen aus zwei Elementen. Nachdem der Root-Knoten erzeugt wurde, wird für jeden Behälter erst einmal die entsprechenden Knoten von dem derzeitigen Behälter nach dem Prinzip aus Abbildung 1 erstellt. Dann werden im zweiten Schritt für alle Knoten des vorherigen Behälters Zeiger auf die Knoten im aktuellen Behälter erstellt. Dies wird dann für jeden Behälter wiederholt. Mit diesem Prinzip macht man also aus einem Graphen ein Array mit Knoten, die man nur durch das Inspizieren aller Knoten wieder zurück in den Graphen einordnen kann, da jeder Knoten nur die Information enthält, welche Knoten unter dem jeweiligen Knoten sind. Mehr Informationen werden aber auch nicht für das „Backtracking“ benötigt, sodass Speicher durch das Wegfallen unnötiger Informationen gespart werden kann.

Abbildung 4: UML-Diagramm der Klasse TBaum und TKnoten



3 Beispiele

Als Programmeingabe gibt es nur, wie es auch in der Aufgabenstellung und in den Beispielen vorgegeben wurde, die Flaschenanzahl, die Anzahl der Behälter und daraus resultierend die Anzahl von jedem Behälter. Danach wird das Programm gestartet. Da es hier außer den bereits beschriebenen Phänomen bedingt durch meine Laufzeitoptimierungen (s. Abbildung 3) keine besonderen „Randbeispiele“, „Sonderfälle“ oder ähnliches gibt und auch selbst erdachte Beispiele genauso viel Sinn ergeben wie die Verwendung der vorgegebenen Beispiele, werde ich im Folgenden nur solche Beispiele auflisten.
Ab dem dritten Beispiel dauerte die Berechnung zu lange, als dass ich die Anzahl der Kombinationen ausrechnen könnte. Bei Beispiel 4 und 5 würde es aufgrund der noch größeren Anzahl der durchlaufenen Knoten, noch länger dauern, weshalb ich das Ergebnis hier nicht aufgeführt habe.

3.1 Beispieltabelle

Tabelle 1: Beispiele

	Flaschen-anzahl	Behälter-anzahl	Fassungsvermögen der i-ten Behälter	Kombinations-anzahl	Anzahl der durchlaufenen Knoten
Beispiel 0	7	2	{3, 5}	2	15
Beispiel 1	5	3	{2, 4, 4}	13	58
Beispiel 2	10	3	{5, 8, 10}	48	336
Beispiel 3	30	20	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}	Ø	22412157560166

Abbildung 5: Nassi-Shneiderman-Diagramm zur Erstellung des gerichten Graphens

constructor erzeuge(Flaschenanzahl, Behälteranzahl: int ; Behälterarray: array of int)

wichtige Variablen:

Behälterarray
{s. Abbildung 2}

Flaschenanzahl
{}

Behälteranzahl
{}

Knotenarray
{Ein dynamisches Array vom Typ TKnoten , welcher alle Knoten beinhaltet. Der Index ist dabei die eindeutige Zuordnung zu einem bestimmten Knoten.}

Knotenarray[0]:=TKnoten.erzeuge(0) // *Erzeuge Wurzel*

für I:=1 bis Behälteranzahl tue:

für J:=0 bis Behälterarray[i] tue:

setlength(Knotenarray,length(Knotenarray)+1) // *erweitere das dynamische Array um einen Wert*

Knotenarray[length(Knotenarray)-1]:=TKnoten.erzeuge(j) // *und erzeuge einen neuen Knoten*

für J:=0 bis Behälterarray[i-1] tue:

für K:=0 bis Behälterarray[i] tue:

Knotenarray[length(Knotenarray)-j-Behaelterarray[i]-2].addZeiger(length(Knotenarray)-k-1) // *Füge jedem „Vater“ des aktuellen Knotens einen Verweis auf dem derzeitigen Knoten zu*

4 Quelltext

Im Folgenden werden die beiden zentralen Quelltexte, die Rekursion und die Erzeugung des gerichteten Graphens, wie sie auch schon in den beiden Nassi-Sneiderman-Diagrammen dargestellt wurden (Abbildung 2 und Abbildung 5), dargelegt. Da der restliche Code entweder für die Kommunikation zwischen den Klassen (Getter, Setter und Constructor) oder für die grafische Aufbereitung inklusive das vorherige Erkennen von falschen Eingaben von dem Benutzer zuständig ist, wäre es nicht sinnvoll, diesen hier zu integrieren, da diese Komponente die eigentliche Lösungsfindung nicht beeinflusst.

4.1 Erzeugung des gerichteten Graphen

```
1  constructor TBaumi.erzeuge(Flaschenanzahl: Integer; Behaelteranzahl: Integer; Behaelterarray: array
       of Integer);
2  var i,j,k:integer;
3  begin
4    FFlaschenanzahl:=Flaschenanzahl;
5    FBehaelteranzahl:=Behaelteranzahl;
6    setlength(FBehaelterarray,length(Behaelterarray));
7    for I := 0 to length(Behaelterarray) do begin
8      FBehaelterarray[i]:=Behaelterarray[i];
9    end;
10   QuickSortRekursiv(0,length(FBehaelterarray)); //Sortieren aus Optimierungsgruenden
11   FBehaelterarray[0]:=0;
12
13   setlength(FKnotenarray,1);
14   FKnotenarray[0]:=TKnoteni.erzeuge(0); //Wurzel des Baums; jeder Knoten bekommt einen Wert zugewiesen
15   for i:=1 to FBehaelteranzahl do begin //fuer jede Stufe tue
16     for j:=0 to FBehaelterarray[i] do begin //fuer jede moegliche Flaschenanzahl in dem i-ten Behaelter tue
17       setlength(FKnotenarray,length(FKnotenarray)+1);
18       FKnotenarray[length(FKnotenarray)-1]:=TKnoteni.erzeuge(j); //erzeuge einen neuen Knoten mit dem Wert
19     end;
20     for j:=0 to FBehaelterarray[i-1] do begin //und erstelle einen Zeiger von den Elternknoten zu dem neuen Knoten
21       for k:=0 to FBehaelterarray[i] do begin
22         FKnotenarray[length(FKnotenarray)-j-FBehaelterarray[i]-2].fuegeZeigerhinzu(length(
               FKnotenarray)-k-1); //auf alle Knoten einer Ebene
               darunter
23       end;
24     end;
25   end;
26   FGesamtmaxVolumen:= SumInt(FBehaelterarray);
27   setlength(FmaxFlaschenInputproBehaelter,length(FBehaelterarray));
28   for I:=0 to length(FBehaelterarray)-1 do begin //Erstelle das Array, wo man fuer jede Stufe die maximal
         moeglichen Flaschen, die unter dieser Stufe hineinpassen, herausfinden kann
29     for J:=I+1 to length(FBehaelterarray)-1 do begin
30       FmaxFlaschenInputproBehaelter[i]:=FmaxFlaschenInputproBehaelter[i]+FBehaelterarray[j];
31     end;
32
33   end;
34
35 end;
```

Listing 1: Erzeugung eines gerichteten Graphen in Turbo Pascal

4.2 Rekursion

```
1  procedure TBaumi.partition(rootKnoten:integer;wert:integer;stufe:integer);
2  var i:integer;
3  begin
4      FAnzahldurchlaufendenKnoten:=FAnzahldurchlaufendenKnoten+1;
5      if wert=FFlaschenanzahl then begin Fzaehler:=Fzaehler+1; exit end; //Eine Kombination gefunden; braucht
           nicht mehr weiterzuschauen
6      if wert>FFlaschenanzahl then exit; //zu gross -> braucht nicht mehr weiter suchen
7      if FmaxFlaschenInputproBehaelter[stufe]+wert<FFlaschenanzahl then exit; //zu klein -> keine Kombination
           mehr moeglich
8
9      for i:=1 to FKnotenarray[rootKnoten].getZeigerlength do begin
10         partition(FKnotenarray[rootKnoten].getZeiger(i),FKnotenarray[FKnotenarray[rootKnoten].
               getZeiger(i)].getWert+wert,stufe+1)
11     end;
12 end;
```

Listing 2: Rekursion in Turbo Pascal

Abbildungsverzeichnis