Hausaufgabe „Grundlagen der Algorithmik“
TU Berlin, 30.05.2018

(Niedermeier/Bentert, Sommersemester 2018)
Abgabe bis zum 26.06.2018
Viel Erfolg!

Hinweise:

- Die Hausaufgabe kann im Zeitraum **vom 20.06.18 bis 26.06.18** in den Tutorien abgegeben werden. Eine Abgabe in der Vorlesung ist am **20.06.18** möglich. Letzte Abgabemöglichkeit ist am **26.06.18** in den Tutorien.
- Die Hausaufgabe kann alleine oder in Gruppen der Maximalgröße drei bearbeitet werden.
- Plagiate werden nicht toleriert. Sollte eine Lösung in zwei Abgaben mit den gleichen Formulierungen/Programmen vorkommen, so werden **beide** Abgaben mit **0 Punkten** bewertet!
- Es gibt einen Programmier- und einen Theorieteil. Es können insgesamt bis zu **25 Portfoliopunkte** erreicht werden.
- Es gibt insgesamt 4 Aufgaben, wobei die ersten beiden mit Divide & Conquer und die letzten beiden mit einem dynamischen Program gelöst werden müssen. Andere Techniken dürfen **zusätzlich** verwendet werden.
- Theorieteil:
  - Schreiben Sie Name und Matrikelnummer aller Gruppenmitglieder auf die erste Seite Ihrer Abgabe. Schreiben Sie zusätzlich auf jedes abgegebene Blatt die Matrikelnummer aller Gruppenmitglieder. Gruppen aus unterschiedlichen Tutorien sind zugelassen, bitte geben Sie nur ein Mal ab.
  - Alle in der Vorlesung gezeigten Sätze und Lemmas (insbesondere „Master Theorem“) dürfen benutzt werden.
  - Begründen Sie alle Antworten. Antworten ohne Begründung erhalten **0 Punkte**.
  - Beschreiben Sie knapp (Fließtext oder Pseudocode, **nicht mehr als eine DinA4-Seite**) Ihren Algorithmus für das jeweilige Problem.
  - Beweisen Sie die Korrektheit einer der beiden „Divide & Conquer“-Algorithmen (4 Punkte) und einer der beiden dynamischen Programme. (3 Punkte)
  - Analysieren Sie die Laufzeit der beiden im vorderen Punkt **nicht** gwählten Algorithmen. (je 3 Punkte)
- Programmierteil:
  - Tragen Sie sich als Team (identisch mit Team beim Theorieteil!) im ISIS in eine Gruppe ein und erstellen Sie auf unserem „Domjudge-System“ (**domjudge.akt.tu-berlin.de**) einen Account mit dem Namen der Gruppe in ISIS (Beispiel: Gruppe 42).
  - Schreiben Sie ein Programm in **C, C++, Java oder Python** für jede der Aufgaben und laden Sie dieses auf unserem Domjudge-System im „Grundlagen der Algorithmik“-Wettbewerb hoch. Sie erhalten dann automatisch eine Antwort, ob das Programm alle Testfälle innerhalb des Zeitlimits gelöst hat oder nicht. Sie erhalten nicht viel mehr Informationen, es kann also sinnvoll sein, eigene Testfälle zu schreiben. Ein- und Ausgabe erfolgen immer über **Standard Input** und **Standard Output**!
  - Für jede Aufgabe im Programmierteil gibt es 3 Punkte. Für fehlerhafte bzw. zu langsame Abgaben gibt es anteilig Punkte. Da Sie nicht sehen können, welche Abgabe wie viele Testfälle löst, werden wir die **beste der drei letzten** Abgaben bewerten.
  - Es dürfen nur „Standardbibliotheken“ verwendendet werden. Nur Bibliotheken die im Domjudgesystem installiert sind, sind zulässig.

Aufgabe 1: **„Leistungspunkte“** (Divide & Conquer, Punkte)

Geben Sie einen Algorithmus an, der das folgende Problem in $O(n \log(n))$ Schritten löst.

Sie befinden sich in einer Prüfungssituation und haben die Auswahl zwischen verschiedenen Aufgaben, die Sie als nächstes lösen könnten. Um sich die Entscheidung ein wenig einfacher zu machen, entschließen Sie sich, zunächst alle Aufgaben zu lesen und abzuschätzen, wie lange Sie dafür jeweils bräuchten und die Aufgaben nach diesem Kriterium zu sortieren (wir nehmen an, dass die Aufgaben, die mehr Zeit benötigen vorne sind). Es gibt für die Aufgaben unterschiedlich viele Punkte und in diesem ersten Schritt geht es nicht darum, eine finale Entscheidung zu treffen, sondern „unsinnige“ Alternativen auszuschließen. Eine Aufgabe ist unsinnig, wenn es eine andere Aufgabe gibt, die weniger Zeit benötigt (also in ihrer Sortierung später kommt) und mehr Punkte bringt.

Spezifikation für den Programmierteil:

Eingabe:

Die erste Zeile der Eingabe enthält die Anzahl n der Aufgaben. Die folgenden n Zeilen enthalten jeweils zwei durch ein Leerzeichen getrennte natürliche Zahlen, die Position in der Sortierung der Aufgaben nach Zeit und der Wertigkeit der Aufgabe. (Erinnerung: Aufgaben vorne in der Sortierung benötigen viel Zeit.) Sie dürfen davon ausgehen, dass keine zwei Aufgaben genau gleichviele Punkte bringen und dass keine zwei Aufgaben genau gleich lange dauern. Sie dürfen **nicht** davon ausgehen, dass alle vorkommenden Werte zwischen 1 und n liegen (weder in der Sortierung, noch in der Wertigkeit).

Ausgabe:

Geben Sie die Anzahl der „sinnvollen“ Aufgaben über die Standardausgabe (Standard Output) aus.

Beispielinstanzen

1. Instanz

Eingabe

4
1 1
3 5
6 2
8 4

Ausgabe

2

2. Instanz

Eingabe

8
2 7
3 13
6 11
7 12
10 5
11 8
14 10
1000 6

Ausgabe

4

Zeitlimit: 10 Sekunden

——————Lösungsskizze——————

Sortiere die Punkte zunächst absteigend nach Zeitaufwand.

Jeder rekursive Aufruf auf einer sortierten Aufgabenliste macht folgendes: Teile die Liste in der Mitte. Bestimme rekursiv die Aufgaben in den beiden Hälften, die innerhalb der Hälften sinnvoll sind. Lösche aus dem Ergebnis der linken Hälfte (Augaben die länger brauchen) alle Aufgaben, die weniger Punkte bringen, als die größte Punktzahl einer Aufgabe in der rechten Hälfte Gib das Ergebnis der rechten Hälfte sowie den Rest des Ergebnisses der linken Hälfte aus.

Oder als Pseudocode:

```
1 function sinnvoll((x_1, y_1), ..., (x_n, y_n)) { // Aufgaben bereits nach Zeitaufwand sortiert
2       gibt es nur eine Aufgabe, so gib diese aus und beende den Aufruf;
3       A := Aufgaben der linken Hälfte;
4       B := Aufgaben der rechten Hälfte;
5       A' := sinnvoll(A);
6       B' := sinnvoll(B);
7       y_max := größte Punktzahl in B;
8       lösche alle Aufgaben (x_i, y_i) aus A' mit y_i < y_max;
9       return(A' ∪ B'); }
```

Wichtig ist, dass nicht in jedem rekursiven Aufruf erneut sortiert wird und dass beim Kombinieren der beiden Teillösungen nicht jede Aufgabe der linken Teillösung mit jeder Aufgabe der rechten Teillösung verglichen wird, denn das dauert zu lange.

Laufzeit: $O(n\log(n))$ für das Sortieren am Anfang plus $T(n) = 2\cdot T(n/2)+O(n) = O(n\log(n))$.

In jedem Aufruf werden bis zu zwei weitere Aufrufe gestartet ($a = 2$ im Mastertheorem), deren Größe jeweils halbiert wird ($b = 2$ im Mastertheorem). Außerdem wird in jedem Aufruf ein Maximum berechnet, was in $O(n^{(1)})$ Zeit möglich ist ($d = 1$ im Mastertheorem). Damit ergibt sich $1 = \log_2(2) \implies T(n) \in O(n\log n)$.

Aufgabe 2: **Turmbau** (Divide & Conquer, Turm)

Sie haben einen unbegrenzten Vorrat an ununterscheidbaren quaderförmigen Bauklötzen der Maße $2 \times 1 \times 1$. Wie viele verschiedene (massive) quaderförmige Bauklotztürme mit einer Grundfläche von 2x2 und einer Höhe gegebenen Höhe lassen sich damit errichten? (Spiegelungen und Drehungen werden mitgezählt, statische Überlegungen bleiben außen vor).

Spezifikation für den Programmierteil:

Eingabe:

Die Eingabe besteht aus einer einzelnen natürlichen Zahl n. Die betrachtete Höhe ist dann 2^n!

Ausgabe:

Ihr Programm soll die Anzahl der möglichen Türme der Höhe 2^n berechnen. Da die Werte sehr groß werden können, geben Sie Ihr Ergebnis modulo 10^5 aus.

Beispielinstanzen

1. Instanz

Eingabe

0

Ausgabe

2

2. Instanz

Eingabe

1

Ausgabe

9

3. Instanz

Eingabe

7

Ausgabe

5409

Zeitlimit: 20 Sekunden

——————Lösungsskizze——————

Die Idee besteht darin, einen Turm der Höhe $2n$ durch einen horizontalen Schnitt in zwei Türme der Höhe n zu zerteilen. Es kann jedoch passieren, dass dabei einige vertikal stehende Bausteine durchtrennent werden. Dadurch enthalten die Turmhälten in der obersten bzw. untersten Ebene möglicherweise halbe Blöcke (also Würfel der Größe 1x1x1). Wenn wir die 4 möglichen Positionen, an denen sich so ein halber Block befinden kann (von oben betrachtet im Uhrzeigersinn) mit 1-4 nummerieren, können wir die "Schnittstelle" durch eine 4-Bit-Zahl codieren, deren i-tes Bit angibt, ob an Position i ein vertikaler Block durchtrennt wurde.

Für $n \geq 0$ und t, b zwei 4-Bit-Zahlen, definieren wir $T(n, t, b)$ als die Anzahl verschiedener Türme der Höhe n, die in der obersten und untersten Ebene genau an den durch t bzw. b angegebenen Stellen halbe Blöcke enthalten. Im Fall $n = 1$ gibt es dabei eine Besonderheit: Da die oberste und unterste Ebene identisch sind, müssen wir darauf achten, dass die durch t vorgegebenen halben Blöcke ihre „Schnittfläche" oben haben, während die durch b vorgegebenen halben Blöcke ihre Schnittfläche unten haben müssen (insbesondere kann kein Block gleichzeitig in t und b vorkommen). Man sieht dann leicht, dass

$$T(1,t,b) = \begin{cases} 2 & \text{wenn } (t \operatorname{OR} b) = 0000_b \\ 1 & \text{wenn } (t \operatorname{OR} b) \in \{0011_b, 0110_b, 1100_b, 1001_b, 1111_b\} \text{ und } (t \operatorname{AND} b) = 0000_b \\ 0 & \text{sonst} \end{cases}$$

gilt, wobei AND und OR die entsprechenden bitweisen Operatoren sind.

Um nun $T(2n, t, b)$ zu berechnen, schneiden wir den Turm gedanklich in der Mitte durch und summieren die Anzahl Möglichkeiten für alle[1] potentiellen Schnittstellen c auf:

$$T(2n,t,b) = \sum_c T(n,t,c) \cdot T(n,c,b)$$

Die Lösung des Ursprungsproblems ergibt sich dann als $T(2^n, 0000_b, 0000_b)$, da hier keine 1x1x1-Blöcke erlaubt sind. Zur Berechnung müssen wir lediglich für alle $k < n$ und alle[2] möglichen Paare t, b den Wert von $T(2^k, t, b)$ bestimmen, brauchen also $O(n)$ Zeit.

Alternative Lösung (zunächst ohne Divide and Conquer):

Sei $h_i(n)$ die Anzahl Türme mit Höhe n, deren oberste Schicht genau i liegende Blocke enthält ("Typ i"). Offensichtlich ist dann $t(n) := h_0(n) + h_1(n) + h_2(n)$ die Anzahl aller Türme.

Es gilt

$$\begin{aligned} h_2(n) &= 2 \cdot t(n-1) \\ h_0(n) &= t(n-2) \\ h_1(n) &= h_1(n-1) + 2 \cdot h_2(n-1) \end{aligned}$$

Die ersten beiden Gleichungen sind unmittelbar ersichtlich. Die dritte folgt aus der Überlegung, dass sich jeder Typ-1-Turm mit Höhe n aus einem Turm mit Höhe $n-1$ konstruieren lässt, indem man einen liegenden Block aus der obersten Ebene entfernt, durch zwei stehende Blöcke ersetzt und die verbleibende Lücke mit einem liegenden Block füllt. Dazu muss der kleinere Turm natürlich vom Typ 1 oder 2 gewesen sein, in letzterem Fall gibt es zwei Möglichkeiten, dies zu tun.

Es folgt:

$$\begin{aligned} t(n) &= h_2(n) + h_0(n) + h_1(n) \\ &= 2 \cdot t(n-1) + t(n-2) + 2 \cdot h_2(n-1) + h_1(n-1) \\ &= 2 \cdot t(n-1) + t(n-2) + 4 \cdot t(n-2) + h_1(n-1) \\ &= 2 \cdot t(n-1) + t(n-2) + 4 \cdot t(n-2) + t(n-1) - h_0(n-1) - h_2(n-1) \\ &= 3 \cdot t(n-1) + 3 \cdot t(n-2) - t(n-3) \end{aligned}$$

Dieser Ansatz hat eine Laufzeit in $O(2^n)$, da 2^n Einträge berechnet werden müssen und jeder Eintrag in konstanter Zeit berechnet werden kann. Dieser Ansatz kann noch durch Divide and Conquer (Matrixpotenzierung) verbessert werden, sodass der resultierende Algorithmus eine Laufzeit $O(n)$ hat. Weitere Details können hier gefunden werden:
https://stackoverflow.com/questions/50896879/possibilities-to-construct-2n-height-tower-with-2x2-base-of-2x1x1-blocks

[1] Von den theoretisch denkbaren 2^4 können nur 6 tatsächlich vorkommen.
[2] $6 \cdot 6 \in O(1)$ (oder $2^4 \cdot 2^4 \in O(1)$)

Aufgabe 3: **Ähnlichkeit von Folgen** (Dynamisches Programm, Folge)

Ihre Aufgabe ist es, endliche Folgen von rationalen Zahlen zu vergleichen. Insbesondere geht es darum festzustellen, ob eine Folge eine „gestreckte“ oder „gestauchte“ Version einer anderen Folge ist. Implementieren Sie hierzu ein Programm, was (unter anderem) folgendes Ähnlichkeitsmaß berechnen kann: Gegeben zwei endliche Folgen F_1 und F_2 rationaler Zahlen. Sei n die Länge von F_1 und m die Länge von F_2. Es sei $F_1[i]$ das i-te Element der Folge F_1 und analog $F_2[i]$ das i-te Element der Folge F_2. Wir definieren eine gültige Folge $F_{(1,2)}$ von Zahlenpaaren zur Angleichung der Zahlenfolgen folgendermaßen:

- Das erste Paar lautet $(F_1[0], F_2[0])$.
- Das letzte Paar lautet $(F_1[n-1], F_2[m-1])$.
- Ist $(F_1[i], F_2[j])$ ein Element von $F_{(1,2)}$, dann ist das nächste Elemente für $F_{(1,2)}$ eines der drei folgenden Elemente:
  (a) $(F_1[i+1], F_2[j])$,
  (b) $(F_1[i], F_2[j+1])$,
  (c) $(F_1[i+1], F_2[j+1])$.

Es gelte ferner, dass $F_{(1,2)}[a]$ das a-te Zahlenpaar in $F_{(1,2)}$ ist und $F_{(1,2)}[a][1]$ die erste Zahl im Zahlenpaar $F_{(1,2)}[a]$ und $F_{(1,2)}[a][2]$ die zweite Zahl im Zahlenpaar $F_{(1,2)}[a]$ ist. Nun messen wir die Ähnlichkeit zweier endlicher Folgen F_1 und F_2 bezüglich einer gültigen Folge $F_{(1,2)}$ als:

$$A(F_{(1,2)}) = \sqrt{\sum_{i=0}^{|F_{(1,2)}|-1} (F_{(1,2)}[i][1] - F_{(1,2)}[i][2])^2}$$

und suchen die gültige Folge, die diesen Wert minimiert.

Spezifikation für den Programmierteil:

Eingabe:

Jede Zeile enthält jeweils eine Folge, die aus nichtnegativen rationalen (oder natürlichen) Zahlen besteht, die durch Leerzeichen getrennt sind. Die erste Folge ist die „Referenzfolge“.

Ausgabe:

Ihr Programm soll nun die Zeilennummer der Folge ausgeben, die der ersten Folge am ähnlichsten ist (und nicht die erste Folge ist). Die Referenzzeile ist die Nummer eins und danach wird ab zwei weitergezählt.

Beispielinstanz

Eingabe

1.0 2.0 3.0 4.0

2.0 2.0 2.0 2.0 2.0

1.0 1.0 2.0 3.0 4.0 4.0

4.0 3.0 2.0 1.0

10 1

1 1.5 1 1.5 1 1.5 1

Ausgabe 3

3

Zeitlimit: 20 Sekunden

——————Lösungsskizze——————

Wir geben zunächst ein dynamisches Programm zur Bestimmung der Ähnlichkeit zweier Folgen F_1 und F_2 an. Wir berechnen das Quadrat der geforderten Ähnlichkeit. Die Wurzel kann am Ende gezogen werden, da nur die ähnlichste Folge gesucht wird und die Wurzelfunktion streng monoton wachsend ist, muss dies aber nicht getan werden. Es seien ℓ, k die Längen von F_1 bzw. F_2. Es sei $T\colon \mathbb{N} \times \mathbb{N} \mapsto \mathbb{R}$ eine zweidimensionale Tabelle. Wir nennen die Differenz zwischen zwei Zahlen in der Folge $C[i][j] = (F_1[i] - F_2[j])^2$ für $0 \leq i \leq \ell$ und $0 \leq j \leq k$. Wir initialisieren $T[0,0] = C[0,0]$. Als nächstes füllen wir die „Ränder“ der Tabelle $T[0,i] = C[0,i] + T[0,i-1]$ und $T[i,0] = C[i,0] + T[i-1,0]$ für steigende $i > 0$. Nun kann für jeden Eintrag $T[i][j]$, für den $T[i-1,j], T[i,j-1]$ und $T[i-1,j-1]$ vorher berechnet wurden, der Wert $T[i,j] = C[i,j] + \min\{T[i-1,j], T[i,j-1], T[i-1,j-1]\}$ berechnet werden. Eine entsprechende Reihenfolge ist zum Beispiel durch „Für wachsende i **do** Für wachsende j **do** ...“ gegeben. Der gesuchte Ähnlichkeitswert ist dann am Ende in $T[\ell, k]$ gespeichert.

Mit diesem dynamischen Programm kann nun die Referenzfolge mit jeder anderen verglichen werden und der Index der ähnlichsten Folge ausgegeben werden.

Die Laufzeit ist durch $O(n^3)$ beschränkt (wobei n die Eingabegröße ist), da maximal n Folgen mit der Referenzfolge verglichen werden müssen, jede Tabelle maximal n^2 Einträge hat (n beschränkt die Länge der beiden Folgen) und jeder Eintrag in konstanter Zeit berechnet werden kann. Alle anderen Operationen benötigen nur konstante Zeit.

Es verbleibt die Korrektheit des Algorithmus. Für jede Kombination aus i und j beschreibt $T[i,j]$ die Ähnlichkeit der beiden Folgen, wenn die erste Folge nach i Zahlen und die zweite Folge nach j Zahlen abgeschnitten wird. Wir nutzen Induktion über i und j für den Korrektheitsbeweis. Der Eintrag $T[0,0]$ beschreibt korrekt die Ähnlichkeit der beiden Folgen, wenn beide aus nur einer Zahl bestehen. Für den Induktionsschritt seien $T[i',j']$ mit $i' \leq i$ und $j' \leq j$ (und mindestens eines der beiden echt kleiner) schon korrekt berechnet. Es gibt eine optimale Zuordnung zwischen den Einträgen der beiden Folgen. Wir nennen diese Zuordnung wieder $F_{(1,2)}$. Nach Aufgabenstellung ist

$$A(F_{(1,2)})^2 = \left(\sum_{h=0}^{|F_{(1,2)}|-2} (F_{(1,2)}[h][1] - F_{(1,2)}[h][2])^2 \right) + (F_{(1,2)}[|F_{(1,2)}|-1][1] - F_{(1,2)}[|F_{(1,2)}|-1][2])^2.$$

Da nach Voraussetzung $\sum_{h=0}^{|F_{(1,2)}|-2} (F_{(1,2)}[h][1] - F_{(1,2)}[h][2])^2$ schon korrekt berechnet ist, dies nach Aufgabenstellung in $T[i-1,j], T[i,j-1]$ oder $T[i-1,j-1]$ gefunden werden kann und $(F_{(1,2)}[|F_{(1,2)}|-1][1] - F_{(1,2)}[|F_{(1,2)}|-1][2])^2 = C[i,j]$, ist auch $T[i,j]$ korrekt.

Aufgabe 4: **Neidfreie Verteilung** (Dynamisches Programm, Neid)

Sie haben eine Menge von Süßigkeiten $S = \{s_1, s_2, \ldots, s_n\}$, die Sie auf drei Kinder aufteilen sollen. Alle Kinder bewerten die Süßigkeiten genau gleich (gemessen als ganze Zahl zwischen 0 und 10). Es gibt also eine Zufriedensheitfunktion $f\colon S \to \{0, 1, \ldots, 10\}$.

Ihre Aufgabe ist es nun, die Süßigkeiten so aufzuteilen, dass alle Kinder gleich glücklich sind und keines Neid verspürt. Die Zufriedenheit eines Kindes ist die Summe der Zufriedenheiten von den einzelnen Süßigkeiten die das Kind bekommt.

Spezifikation für den Programmierteil:

Eingabe:

Die erste Zeile der Eingabe enthält die Anzahl n der Süßigkeiten. Die folgenden n Zeilen enthalten jeweils eine natürliche Zahl, die den „Wert“ der neuen Süßigkeit angibt.

Ausgabe:

Ihr Programm soll „YUPIII!“ ausgeben, wenn eine neidfreie Aufteilung existiert und andernfalls „ARGH!“.

Beispielinstanzen

1. Instanz

Eingabe

3
1
1
1

Ausgabe

YUPIII!

2. Instanz

Eingabe

4
2
2
1
1

Ausgabe

YUPIII!

3. Instanz

Eingabe

5
3
2
1
1
3

Ausgabe

ARGH!

Zeitlimit: 40 Sekunden

——————Lösungsskizze——————

Betrachte eine vierdimensionale Tabelle $T\colon \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \mapsto \{\text{true}, \text{false}\}$.

Wir füllen die Tabelle für alle Einträge $T[a, b, c, d]$ mit $a, b, c \leq \frac{10}{3}n$ und $d \leq n$, wobei n die Anzahl der Süßigkeiten ist und ein Tabelleneintrag „true“ enthalten soll, wenn es mit den ersten d Süßigkeiten möglich ist, dem ersten Kind a, dem zweiten Kind b und dem dritten Kind c „Zufriedenheit“ zu geben. Formal füllen wir für wachsende d die Tabelle $T[a, b, c, d]$. Es sei hierbei w_d die Zufriedenheit, die die d-te Süßigkeit gibt. Initial ist nur $T[0, 0, 0, 0]$ mit einem positiven Wahrheitswert gefüllt.

$$T[a, b, c, d] = T[a - w_d, b, c, d - 1] \vee T[a, b - w_d, c, d - 1] \vee T[a, b, c - w_d, d - 1]$$

Da jeder Eintrag nur auf Tabelleneinträgen mit kleinerem d zugreift, ist die Reihenfolge zwischen Einträgen mit gleichem d irrelevant.

Es sei s die Summe der Zufriedenheiten aus allen Süßigkeiten. Wenn s nicht durch drei teilbar ist, dann gibt es keine Lösung. Das Ergebnis ist andernfalls in $T[s/3, s/3, s/3, n]$. Da $s \leq 10n$, füllen wir höchstens $(\frac{10}{3}n)^3 \cdot n \in O(n^4)$ Tabelleneinträge. Da jeder Eintrag in konstanter Zeit berechnet werden kann, ist die Laufzeit $O(n^4)$.

Die Rekurenzgleichung ist korrekt, da die neue Süßigkeit entweder an das erste Kind, das zweite Kind oder das dritte Kind gegeben werden muss und dann mit den vorherigen $d-1$ Süßigkeiten die anderen Werte aufgefüllt werden müssen. Wenn $T[s/3, s/3, s/3, n]$ wahr ist, dann ist es möglich, die n Süßigkeiten auf die drei Kindern gerecht aufzuteilen.

Möchte man zusätzlich die Verteilung wissen, so kann man jedes Mal, wenn man einen Tabelleneintrag auf wahr setzt, auch einen Zeiger auf den Vorgängereintrag setzen und am Ende endlang dieser Kette die Verteilung zurückverfolgen.

———————————————