Datenstrukturen und Algorithmen

Vorlesung 18: Dynamische Programmierung (K15)

Prof. Dr. Erika Ábrahám

Theorie Hybrider Systeme
Informatik 2

http://ths.rwth-aachen.de/teaching/ss-14/
datenstrukturen-und-algorithmen/

Diese Präsentation verwendet in Teilen Folien von Joost-Pieter Katoen.

10. Juli 2014

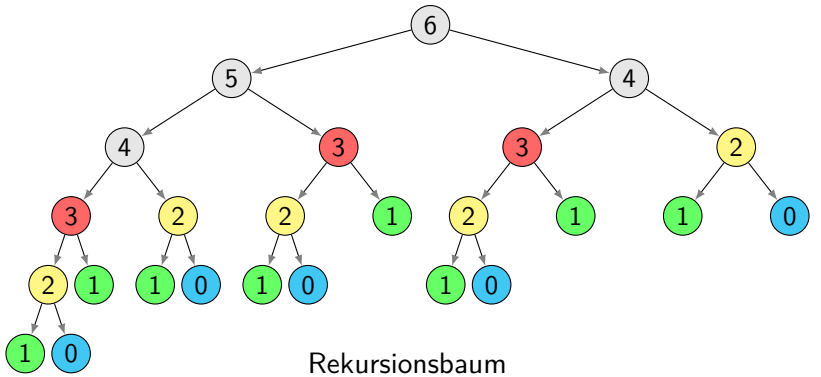
RWTH AACHEN UNIVERSITY

Übersicht

Übersicht

Erinnerung: Fibonacci

$$Fib(0) = 0, \qquad Fib(1) = 1, \qquad Fib(n) = Fib(n-1) + Fib(n-2)$$



Rekursionsbaum

- Wir wollen z. B. $Fib(3)$ nicht ständig neu berechnen.
- *Idee: Speichere einmal berechnete Werte.*

Memoization

Memoization

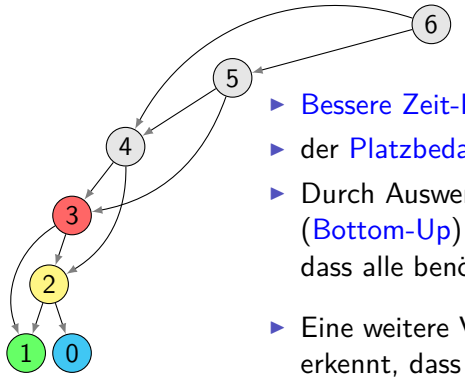
- Bei jedem Funktionsaufruf überprüfe, ob das Ergebnis bereits berechnet wurde (im Cache ist).
- Ist das nicht der Fall, berechne den Wert und speichere zusätzlich das Ergebnis.

Beispiel

```
1 int fibDP(int n) {
2   if (n < 2) return n;
3   int f1 = getCache(n-1), f2 = getCache(n-2);
4   if (f1 == -1) f1 = fibDP(n-1); // nicht gefunden
5   if (f2 == -1) f2 = fibDP(n-2);
6   int fib = f1 + f2;
7   setCache(n, fib);
8   return fib;
9 }
```


Memoization: Dynamische Programmierung

- Memoization hilft, wenn die Teilprobleme überlappen.



Abhängigkeitsgraph

- Bessere Zeit-Komplexität: $\Theta(n)$ statt $\Theta(2^n)$, aber
- der Platzbedarf wächst dabei auf $\Theta(n)$.
- Durch Auswertung von unten-nach-oben (Bottom-Up) kann sogar sichergestellt werden, dass alle benötigten Werte bereits berechnet sind.
- Eine weitere Verbesserung ergibt sich, indem man erkennt, dass hier jeweils nur die zwei letzten Werte benötigt werden (in-place).

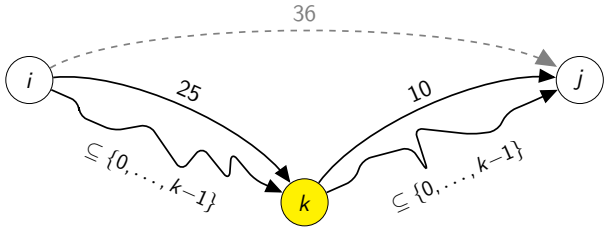
$\Rightarrow$ Auf diesen Grundideen basiert die Dynamische Programmierung.

Example

https://www.cs.usfca.edu/~galles/visualization/DPFib.html

Übersicht

Rekursionsgleichungen: Floyd-Warshall



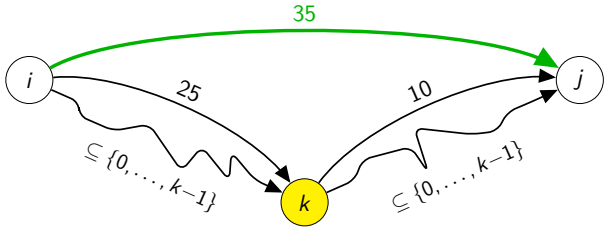
Sei $d_{ij}^{(k)}$ die Länge eines kürzesten Pfades von i nach j über Knoten in der Menge $\{0, 1, \ldots, k\}$.
Rekursionsgleichung die dem Algorithmus zur Grunde liegt:

$$d_{ij}^{(k)} = \begin{cases} W(i,j) & \text{für } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{für } k > 0 \end{cases}$$

- Top-down Abhängigkeit: $d_{ij}^{(k)}$ hängt von $d_{ij}^{(k-1)}$ ab.
- Bottom-up Berechnung: $d_{ij}^{(0)}, d_{ij}^{(1)}, d_{ij}^{(2)}, \ldots$ usw.

Dynamische Programmierung Dynamische Programmierung

Rekursionsgleichungen: Floyd-Warshall



Sei $d_{ij}^{(k)}$ die Länge eines kürzesten Pfades von i nach j über Knoten in der Menge $\{0, 1, \ldots, k\}$.
Rekursionsgleichung die dem Algorithmus zur Grunde liegt:

$$d_{ij}^{(k)} = \begin{cases} W(i,j) & \text{für } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{für } k > 0 \end{cases}$$

- Top-down Abhängigkeit: $d_{ij}^{(k)}$ hängt von $d_{ij}^{(k-1)}$ ab.
- Bottom-up Berechnung: $d_{ij}^{(0)}, d_{ij}^{(1)}, d_{ij}^{(2)}, \ldots$ usw.

Dynamische Programmierung Dynamische Programmierung

Dynamische Programmierung

- Die meisten DP-Probleme sind Optimierungsprobleme, d. h. es gibt verschiedene Lösungen, die mit einer Kostenfunktion bewertet werden. Gesucht ist jeweils eine Lösung mit minimalen Kosten (bzw. maximalem Wert).

Dynamischer Programmierung kann man i. d. R. in vier Teile gliedern:

1. Charakterisiere die Struktur einer optimalen Lösung, und stelle fest ob diese DP ermöglicht.
   - Teilprobleme sind (teilweise) überlappend
   - Rekursive Abhängigkeit zwischen den Teilproblemen.
2. Stelle die Rekursionsgleichung (top-down) für den Wert der Lösung auf.
3. Löse Rekursionsgleichung bottom-up.
4. Bestimme aus dem Wert der Lösung die Argumente der Lösung. Rekonstruiere die Lösung.

Übersicht

Longest Common Subsequence: einige Begriffe

Teilsequenz

Sei $A = (a_1, \ldots, a_m)$ eine Sequenz. Die Sequenz $A_{i_k} = (a_{i_1}, \ldots, a_{i_k})$ ist eine Teilsequenz von A wobei $i_1 < i_2 < \ldots < i_k$ und $i_j \in \{1, \ldots, m\}$.

Eine Teilsequenz von A entsteht aus A indem Elemente weggelassen werden.

Beispiel

$bcdb$ und aa sind Teilsequenzen von $A = abcbdab$.

Gemeinsame Teilsequenz

Sequenz C ist eine gemeinsame Teilsequenz von A und B wenn C sowohl von A als auch von B eine Teilsequenz ist.

Beispiel

bca ist eine gemeinsame Teilsequenz von $A = abcbdab$ und $B = bdcaba$.

Longest Common Subsequence: einige Begriffe

Problem der längsten gemeinsamen Teilsequenz

Gegeben die zwei Sequenzen $A = (a_1, \ldots, a_m)$ und $B = (b_1, \ldots, b_n)$, bestimme deren längsten gemeinsamen Teilsequenz (longest common subsequence), $LCS(A, B)$.

Beispiel

bca ist eine gemeinsame Teilsequenz von $A = abcbdab$ und $B = bdcaba$, aber keine LCS.

$bcba$ ist eine LCS von $A = abcbdab$ und $B = bdcaba$.

$bdab$ ist auch eine LCS von $A = abcbdab$ und $B = bdcaba$.

Naiver Ansatz: betrachte alle Teilsequenzen von A, und überprüfe, welche auch eine Teilsequenz von B sind, und bewahre die längste gefundene Teilsequenz. Zeitkomplexität $\Theta(2^m)$ da es 2^m Teilsequenzen von A gibt.

Longest Common Subsequence: Eigenschaften

Sei $A_i = (a_1, \ldots, a_i)$ der i-te Präfix von $A = (a_1, \ldots, a_m)$ für $0 \leqslant i \leqslant m$.

Lemma (Optimale Teilstruktur)

1. *Enden zwei Sequenzen mit dem selben Zeichen $a_m = b_n$, dann ist dieses Zeichen auch Teil der LCS:*
$$LCS(A_m, B_n) = (LCS(A_{m-1}, B_{n-1}), a_m).$$

2. *Andernfalls gilt entweder*
$$LCS(A_m, B_n) = LCS(A_m, B_{n-1}) \quad \textit{oder}$$
$$LCS(A_m, B_n) = LCS(A_{m-1}, B_n).$$
Insbesondere ist
$$|LCS(A_m, B_n)| = \max(|LCS(A_m, B_{n-1})|, |LCS(A_{m-1}, B_n)|).$$

Longest Common Subsequence: Eigenschaften

Lemma (Optimale Teilstruktur)

Enden zwei Sequenzen mit dem selben Zeichen $a_m = b_n$, dann: $LCS(A_m, B_n) = (LCS(A_{m-1}, B_{n-1}), a_m)$.

Beweis.

Sei $C = LCS(A_m, B_n)$ der Länge k. Wenn a_m nicht als letztes Zeichen in C vorkommt, dann könnte man a_m an C anhängen, und würde eine Teilsequenz von A_m und B_n erhalten. Dies widerspricht der Annahme, dass C eine LCS ist.

Nun ist der Präfix C_{k-1} eine gemeinsame Teilsequenz von A_{m-1} und B_{n-1} der Länge $k-1$. Wir zeigen $C_{k-1} = LCS(A_{m-1}, B_{n-1})$.
Widerspruchsbeweis. Nehme an, es gibt eine gemeinsame Teilsequenz D von A_{m-1} und B_{n-1} mit einer Länge von mindestens k. Dann würde das Anhängen von $a_m = b_n$ an D zu einer gemeinsamen Teilsequenz von A und B führen, deren Länge größer ist als k. Widerspruch. □

Longest Common Subsequence: Rekursiongleichung

Wir können wieder die Rekursiongleichung für den *Wert*, also die Länge der LCS aufstellen: $L[i,j] = |LCS(A_i, B_j)|$

$$L[i,j] = \begin{cases} 0 & \text{für } i = 0 \text{ oder } j = 0 \\ L[i-1, j-1] + 1 & \text{falls } a_i = b_j,\ i,j > 0 \\ \max(L[i, j-1], L[i-1, j]) & \text{falls } a_i \neq b_j,\ i,j > 0 \end{cases}$$

- Das lässt sich direkt als Algorithmus umsetzen.
- Dessen Laufzeit ist $O(|A| \cdot |B|)$, ebenso seine Platzkomplexität.
- Ähnlich dem Rucksackproblem lässt sich dann die LCS rekonstruieren.

Example

`https://www.cs.usfca.edu/~galles/visualization/DPLCS.html`

Übersicht

Ketten von Matrixmultiplikationen: Motivation

Matrixmultiplikation

$C = A \cdot B$ mit $A \in \mathbb{R}^{i \times j}$, $B \in \mathbb{R}^{j \times k}$, $C \in \mathbb{R}^{i \times k}$.

- Die Anzahl der Spalten in Matrix A muss dabei gleich der Anzahl der Zeilen in B sein.
- Komplexität: $i \cdot j \cdot k$ Fließkomma-Multiplikationen.

- Betrachte nun die Multiplikation mehrerer Matrizen: $M = A_1 \cdot A_2 \cdot \ldots \cdot A_n$
- Wir gehen davon aus, dass die Matrizen jeweils miteinander kompatibel sind.
- Solche Ketten lassen sich wegen der Assoziativität der Matrixmultiplikation in beliebiger Reihenfolge berechnen / klammern.

Ketten von Matrixmultiplikationen: Motivation

Beispiel

Sei $A_1 \in \mathbb{R}^{10\times 100}$, $A_2 \in \mathbb{R}^{100\times 5}$, $A_3 \in \mathbb{R}^{5\times 50}$.

- Berechnen wir $A_1 \cdot (A_2 \cdot A_3)$, so muss man $10\cdot 100\cdot 50 + 100\cdot 5\cdot 50 = 50\,000 + 25\,000 = 75\,000$ mal multiplizieren.
- Berechnen wir aber $(A_1 \cdot A_2)\cdot A_3$, dann ergeben sich $10\cdot 100\cdot 5 + 10\cdot 5\cdot 50 = 5000 + 2500 = 7500$ Multiplikationen.

Problem

Finde für eine Kette von Matrizen $A_1, A_2, \ldots, A_n$, mit Dimensionen $d_0 \times d_1$, $d_1 \times d_2$, $\ldots$, $d_{n-1} \times d_n$, eine Klammerung, so dass die Anzahl der Fließkomma-Multiplikationen minimal ist.

Ketten von Matrixmultiplikationen

Sei $P(n)$ die Anzahl der möglichen Klammerungen für n Matrizen:

$$\underbrace{(A_1 \cdot \ldots \cdot A_k)}_{k \text{ Matrizen}} \cdot \underbrace{(A_{k+1} \cdot \ldots \cdot A_n)}_{n-k \text{ Matrizen}}$$

Damit erhält man die Rekursionsgleichung

$$P(n) = \sum_{k=1}^{n-1} P(k)\cdot P(n-k), \qquad P(1) = 1.$$

- Deren Lösung liegt in $\Omega(2^n)$.
- Einfach alle Möglichkeiten auszuprobieren ist keine Option.

Idee: Stelle nach dem selben Prinzip eine Rekursionsgleichung für die minimale Anzahl an Multiplikationen auf.

Rekursionsgleichung

$$\underbrace{(A_1 \cdot \ldots \cdot A_k)}_{k \text{ Matrizen}} \cdot \underbrace{(A_{k+1} \cdot \ldots \cdot A_n)}_{n-k \text{ Matrizen}}$$

$m[i,j]$ sei die minimale Anzahl Multiplikationen für die Teilkette $A_i \cdot \ldots \cdot A_j$.

- Offenbar ist $m[i,i] = 0$ für alle $0 < i \leqslant n$.
- Die Dimension einer Teilkette ist $d_{i-1} \times d_j$.
- Teilen bei Position k ergibt: $m[i,j] = m[i,k] + d_{i-1}\cdot d_k \cdot d_j + m[k{+}1, j]$.
- Wir suchen dabei das optimale k, also:

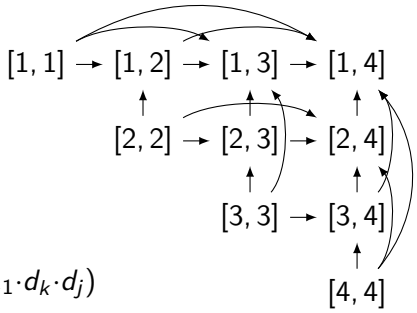
$$m[i,j] = \begin{cases} 0 & \text{für } i = j, \\ \min_{i\leqslant k<j}(m[i,k] + m[k{+}1,j] + d_{i-1}\cdot d_k \cdot d_j) & \text{für } i < j. \end{cases}$$

Ketten von Matrixmultiplikationen: Bottom-Up-Lösung

$$m[i,j] = \begin{cases} 0 & \text{für } i = j, \\ \min_{i\leqslant k<j}(m[i,k] + m[k{+}1,j] + d_{i-1}\cdot d_k \cdot d_j) & \text{für } i < j. \end{cases}$$

- Wie bei Fibonacci wird z. B. das Teilproblem $m[0,1]$ mehrfach verwendet: von $m[0,2], m[0,3], \ldots, m[0,n]$.
- Es gibt für alle $1 \leqslant i < j \leqslant n$ ein Teilproblem, also insgesamt nur $\binom{n}{2} \in \Theta(n^2)$ Teilprobleme.
- Wählen wir eine geschickte Berechnungsreihenfolge und speichern alle $m[i,j]$, dann lässt sich $m[i,j]$ in $\Theta(n)$ berechnen, da die Werte $m[i,k]$ und $m[k{+}1,j]$ bereits bekannt sind.

Ketten von Matrixmultiplikationen: Bottom-Up-Lösung



$$m[i,j] = \min_{i \leqslant k < j} (m[i,k] + m[k{+}1,j] + d_{i-1} \cdot d_k \cdot d_j)$$

- Damit ist eine Zeitkomplexität von $\Theta(n^3)$ bei einem Platzbedarf von $\Theta(n^2)$ möglich.
- Erinnerung: Die naive Variante hätte $\Omega(2^n)$ Zeit, allerdings bei nur $\Theta(1)$ Platzbedarf, benötigt (*Time-Memory-Tradeoff*).
- In der Regel ist es nun eine große Zeitersparnis, zunächst die optimale Klammerung zu finden, statt unüberlegt zu multiplizieren.

Ketten von Matrixmultiplikationen: Algorithmus

```
1  // Eingabe: Die Dimensionen der Matrizen: dim[i]=d_i für i=0...n
2  int matMultOrder(int dim[n+1], int n) {
3    int m[n,n]; // hier 0-basiert!
4    for (int i = 0; i < n; i++)
5      m[i,i] = 0; // Diagonale
6    for (int i = n-1; i >= 0; i--) // Zeilen
7      for (int j = i+1; j < n; j++) { // Spalten
8        int curMin = +inf;
9        for (int k = i; k < j-1; k++) {
10         curMin = min(curMin,
11                      m[i,k] + m[k+1,j] + dim[i]*dim[k+1]*dim[j+1]);
12       }
13       m[i,j] = curMin;
14     }
15   return m[0,n-1];
16 }
```

- Zur einfacheren Rekonstruktion der Lösung werden wir in einer zweiten Matrix jeweils den Index k mit dem Minimum speichern.

Ketten von Matrixmultiplikationen: Beispiel

Beispiel

Sei $A_0 \in \mathbb{R}^{30\times 1}$, $A_1 \in \mathbb{R}^{1\times 40}$, $A_2 \in \mathbb{R}^{40\times 10}$, $A_3 \in \mathbb{R}^{10\times 25}$.

```
m[i,j] = min(m[i,k] + m[k+1,j] + dim[i]*dim[k+1]*dim[j+1]);
dim[] = {30,   1,   40,   10,  25};
```

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}\!\!\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 1200 & 700 & 1400 \\ & 0 & 400 & 650 \\ & & 0 & 10\,000 \\ & & & 0 \end{bmatrix} \end{array} \quad \begin{bmatrix} 0 & 0 & 0 \\ & 1 & 2 \\ & & 2 \\ & & \end{bmatrix}$$

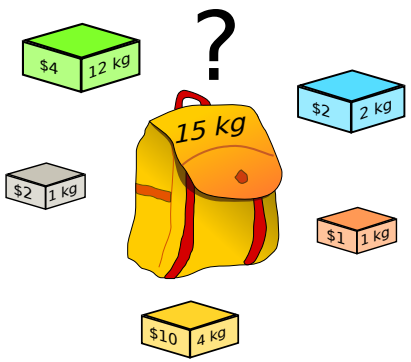
- $((A_0A_1)A_2)A_3$ benötigt 20 700 Multiplikationen.
- Rekonstruktion: $A_0 \cdot ((A_1 \cdot A_2) \cdot A_3)$ ist optimal – 1400 Multiplikationen.

Übersicht

Das Rucksackproblem

Das Rucksackproblem (0-1 Knapsack)

Gegeben sei ein Rucksack, mit maximaler Tragkraft M, sowie n Gegenstände, die sowohl ein Gewicht als auch einen Wert haben.
Nehme möglichst viel Wert mit, ohne den Rucksack zu überladen.



Das Rucksackproblem

Gegeben:

- Maximale Tragkraft M,
- n Gegenstände: $G = \{0, \ldots, n-1\}$,
- Gewichte: $w_i \in \mathbb{N}_0$ für $i \in G$,
- Wert: $c_i \in \mathbb{N}_0$ für $i \in G$ (bzw. Kosten).

Gesucht:

- Der maximale Wert $c_{\max}$.
- $S \subseteq G$ mit $c_{\max} = \sum_{i \in S} c_i$ unter der Nebenbedingung $\sum_{i \in S} w_i \leqslant M$.

Das Rucksackproblem: Rekursionsgleichung

Wir wollen das Problem mittels dynamischer Programmierung lösen.

- Wir bestimmen zunächst $c_{\max}$:
- Angenommen wir kennen den maximalen Wert $c'_{\max}$ des Rucksacks mit Tragkraft M', bei dem nur die ersten $n-1$ Gegenstände berücksichtigt werden.
- Für $c_{\max}$ stellt sich also die Frage, ob der n-te Gegenstand (Gewicht: w_{n-1}, Wert: c_{n-1}) mitgenommen wird.

Betrachten wir beide Fälle:

Ohne: $c_{\max}$ wäre dann gleich $c'_{\max}$ für $M' = M$.

Mit: $c_{\max}$ wäre dann gleich $c'_{\max} + c_{n-1}$ für $M' = M - w_{n-1}$.
Falls $M' < 0$, dann setzen wir $c'_{\max} = -\infty$ („geht nicht“).

$\Rightarrow$ Wähle den Fall mit dem größeren Wert.

Das Rucksackproblem: Rekursionsgleichung

Sei also $C[i,j]$ der maximale Wert des Rucksacks mit Tragkraft j, wenn man nur die Gegenstände $\{0, \ldots, i-1\}$ berücksichtigt.

Es ergibt sich folgende Rekursionsgleichung:

$$C[i,j] = \begin{cases} \max(C[i-1,j], c_{i-1} + C[i-1, j-w_{i-1}]) & \\ -\infty & \text{für } j < 0 \\ 0 & \text{für } i = 0,\ j \geqslant 0 \end{cases}$$

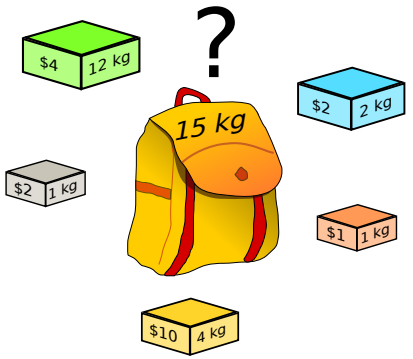
- Dann ist $c_{\max} = C[n, M]$.
- Diese Rekursionsgleichung lösen wir nun bottom-up, indem wir die Rucksäcke mit allen möglichen Gewichten $\{0, \ldots, M\}$ berechnen, wenn wir jeweils einen weiteren Gegenstand hinzunehmen.

Das Rucksackproblem: Algorithmus

```
1  // Eingabe: Gewichte w[i], Werte c[i], Tragkraft M
2  int knapDP(int w[n], int c[n], int n, int M) {
3    int C[n+1,M+1];
4    for (int j = 0; j <= M; j++)
5      C[0,j] = 0;
6    for (int i = 1; i <= n; i++)
7      for (int j = 0; j <= M; j++)
8        if (w[i-1] <= j) {
9          C[i,j] = max(C[i-1,j], c[i-1] + C[i-1,j-w[i-1]]);
10       } else {
11         C[i,j] = C[i-1,j]; // passt nicht
12       }
13   return C[n,M];
14 }
```

- Zeitkomplexität: $\Theta(n \cdot M)$, Platzkomplexität: $\Theta(n \cdot M)$.

Das Rucksackproblem: Beispiel



Das Rucksackproblem: Rekonstruktion

Offen ist noch die Frage, *welche* Gegenstände ($S \subseteq G$) nun eigentlich mitgenommen werden müssen, um $c_{\max}$ zu erreichen.

- Falls $C[i,j] = C[i-1,j]$ ist, dann wurde der Gegenstand nicht mitgenommen (auch bei $c_i = 0$).
- Ausgehend von $C[n,M]$ kann man somit (mit Hilfe der w_i) die Menge S rekonstruieren (in $\Theta(n)$).

Beispiel

`w[] = { 2, 12, 1, 1, 4 }, c[] = { 2, 4, 2, 1, 10 }, M = 15`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2
2	0	0	2	2	2	2	2	2	2	2	2	2	4	4	6	6
3	0	2	2	4	4	4	4	4	4	4	4	4	4	6	6	8
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15