Das Rucksackproblem

Problemlösung durch traditionelle Verfahren

Vollständige Suche und dynamische Programmierung

Juni 2001, J. Wachter

---

1 Einführung

Das *Rucksackproblem* (engl. knapsack problem) gehört zu den klassischen Problemen der Informatik. Es ist ein Problem der kombinatorischen Optimierung, das wegen seiner Komplexität - es ist NP-vollständig - immer wieder zu neuen algorithmischen Lösungen anregt. Beim Rucksack-Problem geht es im Wesentlichen darum, einen Rucksack mit gegebenem Fassungsvermögen so mit wertvollen Gegenständen einer gegebenen Kollektion zu packen, dass der Gewinn möglichtst groß ist, das Fassungsvermögen des Rucksacks aber nicht überschritten wird. Formal ist das Rucksackproblem folgendermaßen definiert:

Definition 1.1. Rucksack-Problem

Gegeben: *Ein Rucksack des Fassungsvermögens $G \in \mathbb{N}$ und n Gegenstände mit Gewichten $w_1, \cdots, w_n \in \mathbb{N}$ und Profiten $p_1, \cdots, p_n \in \mathbb{N}$.*

Gesucht: *Eine optimale Packung des Rucksacks, beschrieben durch eine Teilmenge $I \subseteq \{1, \cdots, n\}$ mit $\sum_{i \in I} w_i \leq G$ und $\sum_{i \in I} p_i = max\{\sum_{i \in I'} p_i : \sum_{i \in I'} w_i \leq G\}$ wobei $I' \subseteq \{1, \cdots, n\}$.*

In dieser Ausarbeitung sollen zwei traditionelle Verfahren zur Bestimmung einer optimalen Lösung für eine Instanz des Rucksackproblems vorgestellt werden. Die *vollständige Suche* ist ein für praktische Anwendungen mit einer großen Anzahl an Gegenständen ungeeigneter Algorithmus, da er exponetielle Laufzeit hat, während eine Lösung mittels *dynamischer Programmierung* praktisch besser verwertbar ist, jedoch leider nur pseudopolynomiell.

2 Vollständige Suche

Ein algorithmisches Problemlösungsverfahren, das nach dem Prinzip der *vollständigen Suche* (engl. *exhaustive search*) arbeitet, betrachtet *jede* und *alle* Instanzen des Suchraums, bis die beste Lösung gefunden wurde. Wir können dabei zwei verschiedene Vorgehensweisen unterscheiden:

- Bei einer *aufzählenden* Vorgehensweise (engl. *enumerative search*) werden sukzessieve alle möglichen Lösungen durch Aufzählung erzeugt und nacheinander bewertet.
- Bei einer *randomisierte* Vorgehensweise (engl. *randomized search*) werden Lösungen zufällig ausgewählt und anschließend bewertet. Bei dieser Vorgehensweise ist es notwendig, über bereits betrachtete Lösungen Buch zu führen, um zu gewährleisten, dass solche Lösungen nicht erneut betrachtet werden.

Da die im zweiten Fall notwendige Buchführung über bereits betrachtete Lösungen umständlich sein kann, wird bei den meisten Problemen der aufzählenden Vorgehensweise den Vorzug gegeben.
Im folgenden Kapitel werden wir die Methode der vollständige Suche am Beispiel des Rucksackproblems demonstrieren.

Beispiel 2.1. Rucksackproblem mit sechs Gegenständen
Betrachten wir dieses Problem an einem konkreten Beispiel. Es sollen sechs Gegenstände mit den in Tabelle 1 angegebenen Gewichten und Profiten gegeben sein. Der zu packende Rucksack soll dabei ein Fassungsvermögen von $G = 20$ besitzen.

Gegenstand	1	2	3	4	5	6
Gewicht	4	7	8	6	5	3
Profit	9	11	14	10	8	7

Tabelle 1: Eine Instanz des Rucksackproblems mit sechs Gegenständen

Wir könnten z.B. den Rucksack mit den Gegenständen 1, 3 und 4 packen, was zulässig ist, da so nur ein Gewicht von 18 zusammenkommt und das Fassungsvermögen des Rucksacks nicht überschritten wird. Damit hätten wir dann einen Profit von $9 + 14 + 20 = 43$ erzielt. Hätten wir allerdings die Gegenstände 1,2, 4 und 6 eingepackt, so hätten wir mit dieser zulässigen Packung (Gesamtgewicht 20) immerhin einen Profit von $9+11+20+7 = 47$ gemacht. Es stellt sich also die Frage ob dies die gewinnbringendste Packung ist, oder ob es noch bessere Lösungen gibt. Diese Frage kann vermutlich erst dann mit Gewißheit beantwortet werden, wenn man alle anderen möglichen Packungen betrachtet und mit der bisher optimalsten Lösung verglichen hat. Wie ermittelt man aber alle möglichen Kombinationen der gegebenen Gegenstände?

Um ein allgemeines Verfahren zur Ermittlung aller Kombinationen zu entwickeln, überlegen wir uns zunächst einmal, wie wir eine Packung beschreiben könnten, damit wir sie von einer anderen Packung unterscheiden können. Tabelle 1 zu unserem Beispiel können wir entnehmen, dass die n Gegenstände mit $1...n$ durchnummeriert wurden. Dies können wir uns für eine Beschreibung einer Packung zunutze machen. Ein Packung $I \subseteq \{1, ..., n\}$ kann dann mit Hilfe eines Arrays $I[1..n]$ beschrieben werden, wobei $I[k] = 1$ falls $k \in I$ und $I[k] = 0$, falls $k \notin I$. Abbildung 1 beschreibt die bereits genannte Packung mit den Elementen 1, 2, 4 und 6 unseres Beispiels. Auf diese Weise kann man also alle Packungen von n Gegenständen als

1	2	3	4	5	6
1	1	0	1	0	1

Abbildung 1: Eine Packung $I = \{1, 2, 4, 6\}$ als Array

n-elementiges Array über $\{0, 1\}$ darstellen. Mathematisch betrachtet können wir Rucksackpakungen von n Gegenständen also durch n-dimensionale Vektoren über $\{0, 1\}$ beschreiben.
Das bietet uns einen Ansatzpunkt, eine Aussage über die Anzahl der möglichen Packungen zu machen. Wir wissen nämlich, dass es 2^n Zeichenketten der Länge n über $\{0, 1\}$ gibt. Damit können wir sagen, dass wir 2^n mögliche Packungen betrachten müssen, um eine optimale Packung zu ermitteln.
Zudem wird uns sofort klar, dass der Ausdruck 2^n eine sehr große Zahl werden kann, wenn n groß wird. Natürlich brauchen wir den Profit nicht für alle diese Kombinationen berechnen, denn einige von ihnen werden ja unzulässig sein, da sie mit ihrem Gesamtgewicht das Fassungsvermögen des Rucksacks überschreiten. Dennoch müssen wir alle Kombinationen erzeugen und das möglichst systematisch. Dabei hilft uns ein Trick, der aus folgenden Beispiel klar wird.
Um alle Kombinationen von n-dimensionalen Vektoren über $\{0, 1\}$ systematisch zu

erzeugen, können wir für $n = 6$ z.B. wie folgt vorgehen.

$$000000, 000001, 000010, 000011, \cdots, 111101, 111110, 111111$$

Dieses Vorgehen entspricht dem binären Aufzählen der Zahlen 0 bis $2^n - 1$!
Mit diesem Ansatz können wir nun eine Algorithmusidee formulieren.

Algorithmusidee
Wir zählen alle Kombinationen von Packungen als Binärvektor durch binäres Zählen nacheinander auf und berechnen für jede zulässige Packung den Gewinn. Die Packung mit dem jeweils höchsten Gewinn merken wir uns.

Algorithmus
Der sich aus dieser Algorithmusidee ergebene Algorithmus 1 ist recht implementierungsnah formuliert. Er bedient sich einiger weiterer einfacher Verfahren, die anschließend kurz erläutert werden.

Algorithmus 1 : RUCKSACKPROBLEM - VOLLSTÄNDIGE SUCHE

EINGABE: int n, G; array $w[1..n]$,$p[1..n]$
AUSGABE: array $opt[1..n]$

```
begin
maximum ← 0
for d ← 0 to 2^n − 1 do
    bvec ← dec2bin(d, n)
    if (weigth(bvec) ≤ G) and (profit(bvec) ≥ maximum)
        then opt ← bvec; maximum ← profit(bvec)
return opt
end
```

Die Funktion *dec2bin(d,n)* wandelt eine Dezimalzahl d in einen n-stelligen Binärvektor mit führenden Nullen um. Der folgende Algorithmus ermöglicht dies.

Algorithmus 2 : DEC2BIN

EINGABE: int d, int n
AUSGABE: array b[1..n]

```
begin
b[1..n] ← 0
for i ← n downto 1 do
    begin
    b[i] ← d mod 2
    d ← d div 2
    end
end
```

Die Funktion *weigth(bvec)* ist die Funktion zur Ermittlung der zulässigen Lösungen. Sie berechnet das Gewicht einer Packung, die durch den Binärvektor *bvec* repräsentiert wird. Eine zulässige Lösung ist eine Packung, deren Gesamtgewicht, das Fassungsvermögen des Rucksacks nicht überschreitet. Die Funktion *profit(bvec)* ist die eigentliche Bewertungsfunktion der vollständigen Suche. Sie berechnet den Profit einer entsprechenden Packung. Die Implementation beider Funktionen ist einfach.

Der angegebene Algorithmus ermittelt für die in Beispiel 2.1 angegebenen Gegenstände und einen Rucksack des Fassungsvermögens $G = 20$ tatsächlich die Kombination der Gegenstände 1, 2, 4 und 6 mit einem Profit von 47 als optimale Lösung.

2.1 Hilfsfunktionen

Bei der oben angegebenen funktion *dec2bin(d,n)* gehen wir davon aus, dass die Anzahl der notwendigen Binärstellen bekannt ist. Dies ist beim Rucksackproblem ja auch gegeben, da die Anzahl der Gegenstände durch die Problemstellung vorgegeben ist.
Ist die Anzahl der Binärstellen jedoch nicht bekannt so muß diese noch ermittelt werden. Dies kann die folgende Funktion *binlength(n)* realisieren.

Algorithmus 3 : BINLENGTH

EINGABE: int n
AUSGABE: int x

Algorithmus binlength (int n) returns int
begin
$x \leftarrow 0$
while $2^x \leq n$ **do**
    $x \leftarrow x + 1$
return x
end

Die Umwandlung einer natürlichen Zahl n in eine Binärzahl kann dann folgendem Algorithmus, der die Funktion *binlength(n)* geschehen.

Algorithmus 4 : DEC2BIN

EINGABE: int n
AUSGABE: int array $b[1..binlength(n)]$

begin
for $i \leftarrow 1$ **to** $binlength(n)$ **do** $b[i] \leftarrow 0$
$z \leftarrow binlength(n)$
$nn \leftarrow n$
repeat
    $b[z] \leftarrow nn \, mod \, 2$
    $z \leftarrow z - 1$
    $nn \leftarrow nn \, div \, 2$
until $nn \leq 0$
end

---

3 Dynamische Programmierung

Rekursive algorithmische Verfahren gehören zu den Standardverfahren der Informatik. Sie ermöglichen häufig sehr elegante Lösungen für verschiedenste Problemklassen und gehören in vielen Fällen zu den schnellsten bekannten Algorithmen. Rekursive Lösungen drängen sich meistens bedingt durch die Problemstruktur oder die Art der verwendeten Datenstrukturen auf.
In diesem Kapitel sollen jedoch vor allem Algorithmen vorgestellt werden, die nach dem Prinzip der sogenannten **dynamischen Programmierung** arbeiten. Das Verfahren der dynamischen Programmierung macht sich die rekursive Struktur des Problems und das sogenannte *Bellmannsche Optimalitätsprinzip* zunutze, welche gewährleisten, dass man eine optimale Lösung aus optimalen Lösungen von Teilproblemen ermitteln kann. Dynamische Programmierung gehört damit zu den traditionellen Verfahen, die auf unvollständigen (Teil-)Lösungen arbeiten und erst im letz-

ten Schritt ihrer Arbeit die optimale Gesamtlösung ermitteln. Das Verfahren der dynamischen Programmierung bringt in einigen Fällen zudem den vorteilhaften Effekt mit sich, laufzeitproblematische rekursive Verfahren bezüglich ihrer Ausführungszeit zu verbessern. Dies werden wir an einem einfachen Einführungsbeispiel, der Berechnung der *Fibonacci-Zahlen*, demonstrieren. Leider bedeutet dies nicht für alle Probleme - wie wir am Beispiel des Rucksackproblems sehen werden - dass sich die Komplexität des Problems verbessert.

3.1 Die Fibonacci-Zahlen

Wir möchten den Zusammenhang zwischen rekursiven Algorithmen und dynamischer Programmierung zunächst an einem Beispiel der rekursiven Definition der **Fibonacci-Zahlen** verdeutlichen, einem Beispiel, das sehr gut bekannt ist und häufig im Informatikunterricht - z.B. im Zusammenhang mit Vermehrungsprozessen bei Mikroorganismen - behandelt wird. Wegen des hohen Bekanntheitsgrad können wir ohne Motivation direkt zur Definition der Fibonacci-Zahlen kommen:

Definition 3.1. Fibonacci-Zahlen
Die n-te Fibonaccizahl ($n \in \mathbb{N}_0$) ist rekursiv wie folgt definiert:

$$f(0) = 0$$
$$f(1) = 1$$
$$f(n) = f(n-1) + f(n-2) \text{ für } n \geq 2$$

Wir können also sehen, dass jede Fibonacci-Zahl aus der Summe der beiden vorhergehenden Fibonacci-Zahlen berechnet werden kann. Betrachten wir dies einmal am Beispiel der ersten 10 Fibonaccizahlen.

n	0	1	2	3	4	5	6	7	8	9	10	$\cdots$
$f(n)$	0	1	1	2	3	5	8	13	21	34	55	$\cdots$

Tabelle 2: Wertetabelle der ersten zehn Fibonacci-Zahlen

Entsprechend der rekursiven Definition 3.1 der Fibonacci-Zahlen lässt sich leicht ein rekursiver Algorithmus 5 angeben.

Algorithmus 5 : FIBONACCI REKURSIV

EINGABE: int n
AUSGABE: int

```
begin
if n ≤ 0
    then return 0
    else if n = 1
        then return 1
        else return FibonacciRekursiv(n − 1)+FibonacciRekursiv(n − 2)
end
```

Implementiert man diesen Algorithmus als Funktion in einer Programmiersprache, so wird man feststellen, dass die Rechenzeit für die Berechnung der n-ten Fibonacci-Zahl - je nach Rechnertyp - ab Eingabewerten zwischen 20 und 30 stark wächst. Diesen Effekt kann man sich erklären, wenn man bedenkt, wie viele Einzelberechnungen zur Berechnung der n-ten Fibonacci-Zahl auch für kleine n notwendig sind. Dabei müssen viele Zwischenergebnisse erneut berechnet werden. Die graphische Darstellung für einen Aufruf *FibonacciRekursiv(6)* in Abbildung 2 verdeutlicht dies. Man kann zeigen, dass der rekursive Algorithmus zur Berechnung

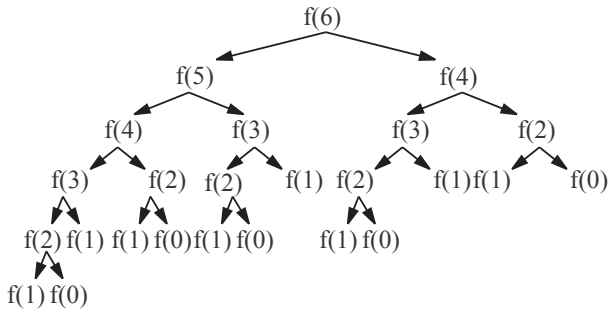

Abbildung 2: Berechnung der sechsten Fibonacci-Zahl

der n-ten Fibonacci-Zahl eine Laufzeit von $O(2^n)$ hat, was ihn wegen seiner nichtpolynomiellen Laufzeit für die Praxis ungeeignet macht.
Die erneute Berechnung von bereits ermittelten Zwischenergebnissen kann entfallen, wenn diese Zwischenergebnisse abgespeichert werden, sobald sie berechnet wurden. Wird ein solches Ergebnis erneut benötigt, dann kann man es einfach abrufen, statt neu zu berechnen. Es muss jedoch gewährleistet sein, dass ein Zwischenergebnis bereits vorliegt, wenn man es benötigt. Ansonsten würde eine Berechnung ja unmöglich.
Dies ist die Idee der **Dynamischen Programmierung**. Bei diesem Verfahren berechnet man die Ergebnisse einer Zwischenrechnung aufgrund der Ergebnisse vorher berechneter Werte. Es handelt sich bei diesem Verfahren um eine Variante des **divide and conquer**-Verfahrens, wobei bei der Dynamischen Programmierung jedoch *bottom-up*, d.h. „von unten nach oben“vorgeht. Der Name der Dynamischen Programmierung rührt daher, dass man die Zwischenergebnisse bei diesem Verfahren in einer Tabelle zwischenspeichert. Mit *Programmierung* ist also das „Berechnen in Tabellen“gemeint. Eine wichtige Voraussetzung für die Anwendbarkeit des Prinzips der Dynamischen Programmierung ist, dass das Problem so beschaffen ist, dass sich die Berechnung einer Lösung des Problems der Größe n auf Berechnungen kleinerer Teillösungen zurückführen läßt. Dieses Prinzip nennt man das *Bellmannsche Optimalitätsprinzip*. Es wird sich in der im folgenden vorgeschlagenen Methodik zur Entwicklung eines Algorithmus nach dem Prinzip der Dynamischen Programmierung widerspiegeln.

Prinzip der Dynamischen Programmierung
Zur Entwicklung eines Algorithmus nach dem Prinzip der Dynamischen Programmierung sind folgende Punkte zu beachten:

1. Analyse und Beschreibung des Lösungsraums und der Struktur der gewünschten Lösung.

2. Entwurf einer rekursiven Lösung, so dass das Ergebnis aus Zwischenergebnissen berechnet werden kann.

3. Überführung der rekursiven Lösung in einen Algorithmus, der Teillösungen berechnet, tabellarisch speichert und diese Teillösungen sukzessive zu einer Gesamtlösung zusammensetzt. Dabei ist zu beachten, dass eine Teillösung der Größe k aus Teillösungen der Größe $< k$ berechnet werden kann, d.h. dass Teillösung der Größe $< k$ auch wirklich zur Verfügung stehen.

Algorithmus für die Fibonacci-Zahlen
Um einen Algorithmus für die Berechnung der Fibonacci-Zahlen nach dem Prinzip der Dynamischen Programmierung zu entwickeln, können wir auf unsere Überlegungen zur rekursiven Lösung zurückgreifen (Punkte 1. und 2.). Wir müssen also nur noch Punkt 3. zufriedenstellend lösen. Dazu überlegen wir uns, dass wir als Tabelle

für das Speichern der Zwischenergebnisse nur ein eindimensionales Feld benötigen. Bei diesem Feld belegen wir die beiden ersten Feldelemente mit den Initalisierungswerten der rekursiven Definition der Fibonacci-Zahlen (Abbruchbedingung oder Elementarfall) und berechnen die verbleibenden Feldeinträge von links nach rechts sukzessive nach der rekursiven Definition der Fibonacci-Zahlen. Abbildung 6 verdeutlicht dieses Vorgehen. Die Rekursion des ursprünglichen Algorithmus' kann

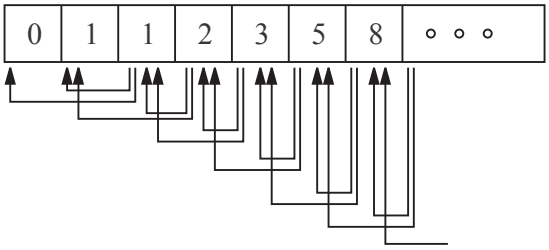

Abbildung 3: Berechnung der Fibonacci-Zahlen in einer Tabelle

sogar in eine Iteration mit einer entsprechenden Schleife überführt werden. Wir erhalten dann den folgenden Algorithmus 6:

Algorithmus 6 : FIBONACCI DYNAMISCHE PROGRAMMIERUNG

```
EINGABE:    int n
AUSGABE:    int f[n]
    array f[0..n]
    begin
    f[0] = 0; f[1] = 1
    if n ≥ 2   then
               begin
               for i ← 2 to n do
                        f[i] = f[i − 1] + f[i − 2]
               end
    return f[n]
    end
```

Der Algorithmus für die Berechnung der n-ten Fibonacci-Zahl mittels Dynamischer Programmierung hat ein für die Praxis sehr viel angenehmeres Laufzeitverhalten. Ein Blick auf die Schleifenstruktur des Algorithmus' zeigt uns, dass wir es mit einer Komplexität von $O(n)$ zu tun haben. Dies verdeutlicht den Vorteil der Dynamischen Programmierung für das Problem der Fibonacci-Zahlen.

Bemerkung 3.1. Interessant ist im Zusammenhang mit den Fibonacci-Zahlen noch zu bemerken, dass auch mit der folgenden Funktion, der Wert der n-ten Fibonacci-Zahl berechnet werden kann.

$$f(n) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

Aufgabe 3.1. Der in Abschnitt 3.1 vorgestellte Algorithmus zur Berechnung der n-ten Fibonacci-Zahl mittels Dynamischer Programmierung hat zwar ein günstiges Laufzeitverhalten von $O(n)$, benötigt aber ebenso $O(n)$ viel Platz. Entwickeln Sie einen Algorithmus, der bei einer Laufzeit von $O(n)$ mit $O(1)$ Platz auskommt.

3.2 Das Rucksackproblem

Es soll nun ein Algorithmus zur Lösung des in Definition 1.1 beschriebenen Rucksackproblems entworfen werden, der nach dem Prinzip der dynamischen Programmierung arbeitet. Zunächst betrachten wir eine besonders einfache Beispielinstanz des Rucksackproblems. Anhand dieses Beispiels wird es leichter sein, die Arbeitsweise des Algorithmus' zu erläutern. Es sollen diesmal nur drei Gegenstände mit den in Tabelle 3 angegebenen Gewichten und Profiten gegeben sein. Der zu packende Rucksack soll dabei ein Fassungsvermögen von $G = 5$ besitzen. In diesem Fall

Gegenstand	1	2	3
Gewicht	3	2	1
Profit	15	10	5

Tabelle 3: Eine Instanz des Rucksackproblems mit drei Gegenständen

möchten wir mit Abbildung 4 auch eine visuelle Veranschaulichung angeben, die wir später noch einmal wieder aufgreifen werden. Nachdem wir uns mit dem Beispiel

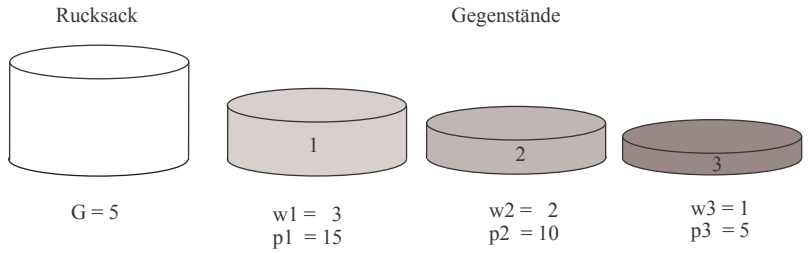

Abbildung 4: Visuelle Darstellung einer Instanz des Rucksackproblems

gut vertraut gemacht haben, werden wir nun die in Abschnitt 3.1 genannten Punkte 1 bis 3 zur Algorithmenentwicklung behandeln.

Überlegungen zur Algorithmusidee
Zu 1: Um zu einer optimalen Lösung zu gelangen, müssen Kombinationen von möglichen Packungen betrachtet werden. Dies soll systematisch erfolgen.
Es stellt sich die Frage, ob es einen rekursiven Ansatz zum systematischen Erzeugen von möglichen Packungen und für deren anschließende Bewertung gibt. Der in vielen anderen Fällen durchaus sinnvolle Ansatz, nämlich einfach einen Gegenstand außeracht zu lassen und eine Lösung für die verbleibenden $n-1$ Gegenstände zu suchen, erweist sich als Irrtum. Eine optimale Lösung für $n-1$ Gegenstände würde den Rucksack ja schon voll packen und der zurückgelassene Gegenstand hätte möglicherweise keinen Platz mehr im Rucksack. Es muß also ein anderer Ansatz für die Aufteilung in Teilprobleme gewählt werden.
Dazu müssen wir uns folgenden Zusammenhang deutlich machen: Wenn der Rucksack mit dem Fassungsvermögen G optimal mit einer Packung $I \subseteq \{1, \cdots, n\}$ der gegebenen Gegenstände gepackt wurde, dann gilt für jeden Gegenstand $i \in I$, dass der um das Gewicht $G - w_i$ verkleinerte Rucksack ebenfalls optimal mit einem Teil der Gegenstände $\{1, \cdots, n\} \setminus \{i\}$ gepackt wurde, indem genau die Packung $I \setminus \{i\}$ benutzt wird. Dies entspricht dem Bellmannschen Optimalitätsprinzip, nach dem eine optimale Lösung eines Problems aus optimalen Teillösungen kleinerer Größe zusammengesetzt ist. Wir können uns dies mit Abbildung 5 für unser einfaches Beispiel einmal vor Augen führen.

Wir können sehen, dass der Rucksack mit einem Fassungsvermögen von $G = 5$ optimal mit den Gegenständen 1 und 2 gepackt ist, wobei ein Gewinn von 25 gemacht wird. Ebenso ist ein um $w_2 = 2$ reduzierter Rucksack mit einem Fassungsvermögen von $G = 3$ optimal gepackt, wenn er nur den Gegenstand 1 beinhaltet.

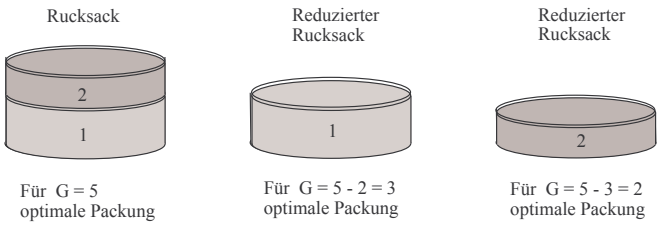

Abbildung 5: Verdeutlichung des Bellmannschen Optimalitätsprinzips

Er macht einen Gewinn von 15. Einen höheren Gewinn kann man mit keiner anderen Packung erhalten. Analog dazu ist ein um $w_1 = 3$ reduzierter Rucksack (Fassungsvermögen $G = 2$) optimal gepackt, wenn er nur Gegenstand 2 beinhaltet. Der Gewinn beträgt dann 10, was mit keiner anderen Packung zu erreichen ist.
Diese Überlegungen sind natürlich nur veranschaulichender Art. Sie müssen noch bewiesen werden. Dies kann geschehen, wenn die angedeute rekursive Beziehung vollständig aufgestellt ist. Kommen wir damit zur eigentlichen Algorithmusidee der dynamischen Programmierung.

Zu 3: Im Sinne einer *bottom up*-Vorgehensweise sollen sukzessive kleine Teillösungen ermittelt werden, die schrittweise zu einer Gesamtlösung führen sollen. Dies können wir folgendermaßen tun:
Wir nehmen sukzessive kleine Kollektionen von Gegenständen und zwar so, dass unsere kleinste Kollektion aus keinem Gegenstand besteht, die nächste Kollektion aus dem Gegenstand mit der Nummer 1, die nächste Kollektion aus die Gegenstände 1 und 2 usw., bis wir die größte Kollektion mit allen Gegenständen von 1 bis n haben. Für diese Kollektionen prüfen wir, welcher Gewinn maximal zu machen ist, wenn schrittweise kleine Rucksäcke mit einem Fassungsvermögen von anfangs 0 bis letztendlich G bepackt werden.
Nehmen wir an, wir hätten einen Rucksack des Fassungsvermögens $h \leq G$ optimal gepackt, indem wir eine Auswahl der Gegenstände $1 \cdots i-1$ mit $i \leq n$ verwendet haben. Wollen wir nun den i-ten Gegenstand hinzufügen, dann müssen wir folgende Fälle unterscheiden.

- Verfügen wir über keinen Gegenstand (als $i = 0$ definiert), so können wir ihn auch nicht hinzufügen.
- Ist das Fassungsvermögen des Rucksacks $h < w_i$, so ist kein Platz in diesem Rucksack und wir sollten einen größeren Rucksack nehmen.
- Ist jedoch genug Platz für den Gegenstand i im Rucksack (also $h \geq w_i$), dann müssen wir unterscheiden in welchem Fall der Gewinn größer ist:
  - Man nimmt den Gegenstand i hinzu, gewinnt damit zwar einen Wert von p_i hinzu, für die übrigen Gegenstände bleibt dann aber nur noch ein Rest von $h - w_i$ an Fassungsvermögen. Der Gewinn ergibt sich dann aus der Summe des maximalen Gewinns, den man mit einer Kollektion der Gegenstände $1, \cdots i-1$ bei einem Rucksack des Fassungsvermögens $h - w_i$ erreichen kann und dem Profit des i-ten Gegenstands.
  - Man nimmt den Gegenstand nicht hinzu, das Fassungsvermögen bleibt bestehen und der Gewinn entspricht dem maximalen Gewinn, den man mit einer Kollektion der Gegenstände $1, \cdots i-1$ bei einem Rucksack des Fassungsvermögens $h - w_i$ erreichen konnte.

Diese Überlegungen können formalisiert werden, indem man eine Funktion $w(i, h)$ wie folgt definiert:
$w(i, h)$ repräsentiere den maximal möglichen Gewinn beim Packen eines Rucksacks

des Fassungvermögens $h \leq G$ mit einer Auswahl der Gegenstände $1, \cdots, i$ wobei $i \leq n$.
Dann berechnet sich $w(i,h)$ nach den obigen Überlegungen rekursiv wie folgt:

$$w(i,h) = \begin{cases} 0, & i = 0 \\ w(i-1,h), & i > 0 \text{ und } h < w_i \\ max(w(i-1,h), w(i-1,h-w_i) + p_i) & \text{sonst.} \end{cases}$$

Die Korrektheit dieser Funktion ergibt sich aus den oben angegebenen Überlegungen.
Aufgrund der bisher gemachten Vorüberlegungen können wir nun einen Algorithmus zur Berechnung des maximal möglichen Gewinns angeben.

Algorithmus 7 : RUCKSACKPROBLEM DYNAMISCHE PROGRAMMIERUNG (1)

EINGABE: int n,G,array $w[1..n]$,$p[1..n]$
AUSGABE: int
begin
array $w[0..n, 0..G]$
for $i \leftarrow 0$ **to** n **do**
  for $h \leftarrow 0$ **to** n **do**
    if $i = 0$ **then** $w[i,h] \leftarrow 0$
    else if $i > 0$ and $h < w[i]$
      then $w[i,h] \leftarrow w[i-1,h]$
      else $w[i,h] \leftarrow max(w[i-1,h], w[i-1,h-w[i]] + p[i])$
return $w[n,G]$
end

Bisher können wir nur den maximal möglichen Gewinn für einen Rucksack gegebenen Fassungsvermögens und eine Kollektion an Gegenständen berechnen. Natürlich interessiert uns auch die Lösung selbst, also *die* Packung $I \subseteq \{1, \cdots, n\}$, die den maximalen Gewinn bringt. Diese Lösung können wir ebenso sukzessive ermitteln, indem wir entsprechend der beschriebenen Fallunterscheidung - je nachdem in welchem Fall der höhere Gewinn zu erreichen ist - den Gegenstand i hinzunehmen oder nicht. Es ergibt sich der entsprechende Algorithmus, der ein Array $a[0..n, 0..G]$ für die Berechnung der jeweiligen Auswahlen verwendet. Dieses Array speichert jedoch Mengen, im Gegensatz zu dem array $w[0..n, 0..G]$ des obigen Algorithmus'. Die optimale Packung liegt dann in $a[n,G]$ vor. Algorithmus 8 verdeutlicht diese Vorgehensweise.

Verdeutlichen wir uns nun den Ablauf des Algorithmus an unserem kleinen Beispiel. Die in Abbildung 6 dargestellte Tabelle stellt die Berechnung der Arrays $w[0..n, 0..G]$ und $a[0..n, 0..G]$ gleichzeitig dar. Mit $\{\}$ ist die leere Menge gemeint. Die Pfeile verdeutlichen den Berechnungsvorgang, wobei die fett gezeichneten Pfeile die „Gewinner "bei der Maximumsbildung darstellen. Grau unterlegt wurden die Einträge, die die Unmöglichkeit von Packvorgängen wegen zu wenig Fassungsvermögens repräsentieren.

Betrachten wir Abbildung 6, so können wir die Konstruktion einer optimalen Lösung und die Berechnung des zugehörigen Gewinnes genau beobachten. Wir gehen dabei schrittweise im Sinne der äußeren Schleife vor.

- Zunächst steht uns mit $i = 0$ noch kein Gegenstand zur Verfügung. Entsprechend ergibt sich für jeden denkbaren Rucksack des Fassungsvermögens $h \leq G$ eine leere Packung und ein Gewinn von 0.
- Im nächsten Schritt der äußeren Schleife des Algorithmus' steht uns der Gegenstand $i = 1$ zur Verfügung. Rucksäcke des Fassungsvermögens $h < w[1] = 3$ können damit noch nicht bepackt werden. Erst ab einem Fassungsvermögen

Algorithmus 8 : RUCKSACKPROBLEM DYNAMISCHE PROGRAMMIERUNG (2)

```
EINGABE:    int n,G,array w[1..n],p[1..n]
AUSGABE:    set
begin
array w[0..n, 0..G]
array a[0..n, 0..G] of set
for i ← 0 to n do
    for h ← 0 to n do
            a[i, h] ← ∅
for i ← 0 to n do
    for h ← 0 to n do
    if i = 0    then w[i, h] ← 0
                else if i > 0 and h < w[i]
                    then    w[i, h] ← w[i − 1, h]
                            a[i, h] ← a[i − 1, h]
                    else w[i, h] ← max(w[i − 1, h], w[i − 1, h − w[i]] + p[i])
                        if (w[i − 1, h − w[i]] + p[i]) > (w[i − 1, h])
                            then    a[i, h] ← a[i − 1, h − w[i]] ∪ {i}
                            else    a[i, h] ← a[i − 1, h]
return w[n, G]
end
```

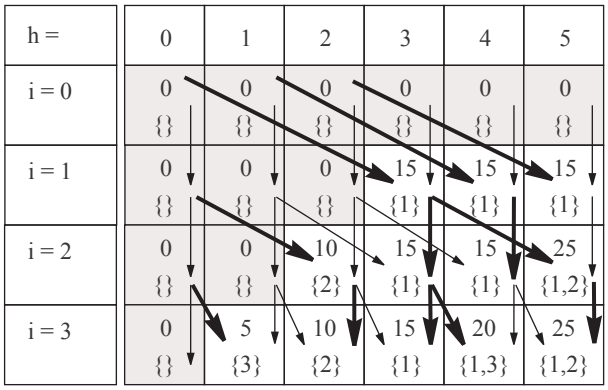

Abbildung 6: Ablauf des Algorithmus, Array w[i,h], a[i,h]

von $h \geq w[1] = 3$ kann Gegenstand 1 in den Rucksack gepackt werden. Der maximale Gewinn beträgt dabei 15.

- Jetzt steht uns mit $i = 2$ die Kollektion $\{1, 2\}$ zur Verfügung. Wieder wird systematisch geprüft, wie Gegenstände in Rucksäcke gepackt werden können. Bei einem Rucksack mit einem Fassungsvermögen von $G = 2$ ist es möglich, Gegenstand 2 einzupacken. Es wird damit ein Gewinn von 10 erwirtschaftet. Für Rucksäcke mit einem Fassungsvermögen von $G = 3$ und $G = 4$ ist es besser, Gegenstand 1 einzupacken, da so ein Gewinn von 15 gemacht wird. Erst ab einem Fassungsvermögen von $G = 5$ ist es besser, die Gegenstände 1 und 2 einzupacken.

- Zum Schluß stehen alle Gegenstände zur Verfügung. Sind die Rücksäcke von kleinerem Fassungsvermögen als das Gewicht des kleinsten Gegenstandes, so können sie nicht bepackt werden. Erst ein Rucksack mit dem Fassungsvermögen $G = 1$ kann mit Gegenstand 3 bepackt werden. Sobald das Fassungsvermögen auf $G = 2$ wächst, ist es am gewinnbringendsten, Gegenstand

2 einzupacken. Bei einem Rucksack mit $G = 3$ sollte Gegenstand 1 eingepackt werden, bei $G = 4$ ist die Packung mit den Gegenständen 1 und 3 am gewinnbringendsten, während die optimale Packung für einen Rucksack mit $G = 5$ aus den beiden Gegenständen 1 und 2 besteht. Damit hat der Algorithmus die optimale Lösung in $w[n, G]$ bzw. $a[n, G]$ berechnet.

Komplexität des Algorithmus

Die beiden Faktoren, die die Komplexität des Algorithmus bestimmen, sind die Größe des Arrays sowie der Aufwand für die Berechnung eines einzelnen Arrayeintrags. Die Größe des Arrays haben wir schon mehrmals angesprochen, sie beträgt $O(n \cdot G)$, der Aufwand für die Berechnung eines Arrayeintrags ist konstant, also O(1). Wir kämen dann auf eine Gesamtkomplexität von $O(nG)$.

Dabei sind wir aber stillschweigend von einem uniformen Komplexitätsmaß ausgegangen, bei dem wir annehmen, dass alle arithmetischen Operationen $O(1)$ Kosten erzeugen und die Eingabe aus natürlichen Zahlen besteht, die jeweils nur einen Speicherplatz benötigen.

Tatsächlich müssen wir aber von der Bit-Komplexität ausgehen, denn die Komplexitätsklassen P und NP sind über Turingmaschinen definiert, die auf binären Codierungen arbeiten. Nennen wir also $k = log(G)$ die Länge der Binärdarstellung von G, dann ist G proprotional zu 2^k. So gelangen wir zu der Feststellung, dass die Bit-Komplexität des vorgestellten Algorithmus $O(n \cdot 2^k)$ ist, womit wir es mit einem exponentiell wachsenden Algorithmus zu tun haben.