Backtracking
am Beispiel des Rucksackproblems

Volker Claus, Universität Stuttgart

Angepasstes Exemplar aus dem Simba-Projekt PAL für die Teilnehmer der Vorlesung "Formale Methoden für Wirtschaftsinformatik"
WS 02/03

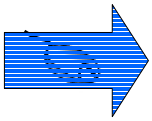
Hinweise:

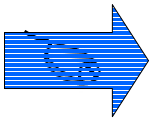
Dieser Kurs sollte mit PowerPoint 97 angesehen werden.

Das Material wird automatisch eingeblendet, ebenso erfolgt ein automatischer Übergang zur nächsten Folie.

Falls Ihnen dies zu schnell ist, so drücken Sie die rechte Maustaste oder die Rückwärts-Pfeil-Taste.

Ist es zu langsam, so klicken Sie mit der linken Maustaste, um sofort den nächsten Gegenstand angezeigt zu bekommen.

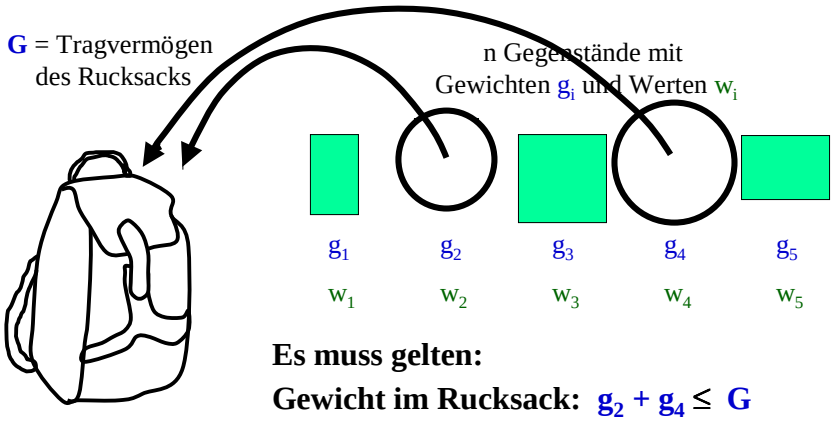
Wenn das Zeichen 

erscheint, so hält die Präsentation an, d.h. es wird nicht automatisch weitergeschaltet, sondern Sie müssen mit der Maustaste zur nächsten Folie übergehen. 

Defintion: *Rucksackproblem*

Gegeben sind n Gegenstände, deren Gewichte $g_1, g_2, ..., g_n$ seien. Der Rucksack hat eine maximale Tragfähigkeit von G Gewichtseinheiten; wird er mit einem größeren Gewicht gefüllt, reißt er und wird unbrauchbar. Jeder der Gegenstände hat zusätzlich einen Wert (auch "Wichtigkeit" oder "Nutzen" genannt), der mit $w_1, w_2, ..., w_n$ beziffert wird. Man soll den Rucksack nun mit Gegenständen so füllen, dass alle eingepackten Gegenstände *höchstens* das Gewicht G besitzen und dass deren Wichtigkeit *mindestens* einen vorgegeben Wert W erreicht.

Z.B. die Gegenstände 2 und 4 einpacken:



Es muss gelten:

Gewicht im Rucksack: $g_2 + g_4 \leq G$

Angestrebt wird beim Einpacken:

Wert (Wichtigkeit): $w_2 + w_4 \geq W$

Anwendung: Auf eine U-Boot- oder eine Weltraumfahrt darf jede(r) Mitreisende persönliche Gegenstände eigener Wahl mitnehmen, die pro Person ein Gesamtgewicht G nicht übersteigen; jede(r) wird sie so auswählen, dass der erwartete persönliche Nutzen dieser Gegenstände möglichst groß wird bzw. einen vorgegebenen "Nützlichkeitswert" W übersteigt.

Ähnliche Anwendung: Bei der Konzipierung eines Autos kann man viele verschiedene Annehmlichkeiten einbauen, z.B. Klimaanlage, kleine Elektromotoren für alle möglichen Hilfen, einen Wassertank, einen größeren Benzintank, eine kleine Bar, diverse Computer, stabilere Achsen usw. Es passt aber nicht alles raum- oder gewichtsmäßig hinein. Man wird daher (auf Grund von Kundenbefragungen) diesen Geräten eine Wichtigkeit zuordnen und muss dann ermitteln, wie man den Nutzwert möglichst groß machen kann unter der räumlichen oder gewichtsmäßigen Nebenbedingung.

Zahlen-Beispiel 1: $G = 12$, $W = 6$, $n = 4$

Folge der g_i:	4	5	6	9
Folge der zugehörigen w_i:	2	3	3	6

Finden Sie eine geeignete Teilmenge!

Sicher haben Sie rasch eine Lösung gefunden.
Vielleicht diese:

Teilfolge der Gewichte g_i:	5	6
Zugehörige Teilfolge der Werte w_i:	3	3

Kontrolle: $5 + 6 = \mathbf{11} \leq G = 12$
$3 + 3 = \mathbf{6} \geq W = 6$

Hinweis 1: Es gibt noch eine weitere Lösung: $g_4 = 9$ mit $w_4 = 6$.

Hinweis 2: Setzt man $W = 7$, so gibt es keine Lösung.

Zahlen-Beispiel 2: $G = 40$, $W = 15$, $n = 8$

Folge der g_i:	6	7	8	9	9	11	13	15
Folge der zugehörigen w_i:	2	2	3	4	5	5	6	5

Eine geeignete Teilmenge lautet:

Teilfolge der g_i:	6	7	8	9	9
Zugehörige Teilfolge der w_i:	2	2	3	4	5

Kontrolle:

$6 + 7 + 8 + 9 + 9 = \mathbf{39} \leq G = 40$

$2 + 2 + 3 + 4 + 5 = \mathbf{16} \geq W = 15$

Versuchen Sie, weitere geeignete Teilmenge zu finden!

Zahlen-Beispiel 2: $G = 40$, $W = 15$, $n = 8$

Folge der g_i:	6	7	8	9	9	11	13	15
Folge der zugehörigen w_i:	2	2	3	4	5	5	6	5

Es gibt weitere Lösungen, zum Beispiel:

Teilfolge der g_i:	11	13	15
Zugehörige Teilfolge der w_i:	5	6	5

Kontrolle:

$11 + 13 + 15 = \mathbf{39} \leq G = 40$

$5 + 6 + 5 = \mathbf{16} \geq W = 15$

Zahlen-Beispiel 2: G = 40, W = 15, n = 8

Folge der g_i:	6	7	8	9	9	11	13	15
Folge der zugehörigen w_i:	2	2	3	4	5	5	6	5

Noch eine Lösung, sogar mit größerer Wichtigkeit:

Teilfolge der g_i:	7	9	11	13
Zugehörige Teilfolge der w_i:	2	5	5	6

Kontrolle:

$7 + 9 + 11 + 13 = \mathbf{40} \leq G = 40$

$2 + 5 + 5 + 6 = \mathbf{18} \geq W = 15$

Es gibt noch mehrere weitere Lösungen, aber keine mit größerer Wichtigkeit als 18. Bitte selbst untersuchen.

Zahlen-Beispiel 3: G = 300, W = 100, n = 10

Folge der g_i:	26	30	42	46	57	60	70	83	88	94
Folge der zugehörigen w_i:	8	6	13	12	15	23	22	30	29	34

Versuchen Sie in 4 Minuten, eine Lösung zu finden!

Zahlen-Beispiel 3: G = 300, W = 100, n = 10
Folge der g_i: 26 30 42 46 57 60 70 83 88 94
Folge der zugehörigen w_i: 8 6 13 12 15 23 22 30 29 34

Eine Lösung hierzu (es gibt weitere):

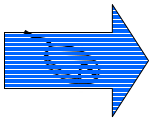
Teilfolge der g_i:	57	60	88	94
Zugehörige Teilfolge der w_i:	15	23	29	34

Kontrolle:
$57 + 60 + 88 + 94 = 299 \leq G = 300$
$15 + 23 + 29 + 34 = 101 \geq W = 100$

Gibt es weitere Lösungen für dieses Beispiel? Dies lässt sich durch ein systematisches Probierverfahren nachprüfen, das auf folgendem Prinzip beruht:

Man betrachtet einen Gegenstand, z.B. den letzten mit der Nummer n, und ermittelt, wie eine Lösung aussieht, zu der dieser Gegenstand gehört, und wie eine Lösung aussieht, zu der er nicht gehört.

Diese Überlegung führt zu dem folgenden Prinzip der Aufspaltung des Problems in (zwei) Teilprobleme.



Man zerlege das Problem

P

Zu G, W, g_1, g_2, ..., g_n und w_1, w_2, ..., w_n finde man eine Teilfolge g_{i_1}, g_{i_2}, ..., g_{i_r}, deren Summe nicht größer als G und deren zugehörige Summe $w_{i_1} + w_{i_2} + ... + w_{i_r} \geq W$ ist.

in die beiden Probleme

P1

Zu G, W, g_1, g_2, ..., g_{n-1} und w_1, w_2, ..., w_{n-1} finde man eine Teilfolge g_{i_1}, g_{i_2}, ..., g_{i_r}, deren Summe nicht größer als G und deren zugehörige Summe $w_{i_1} + w_{i_2} + ... + w_{i_r} \geq W$ ist.

und

P2

Zu $G-g_n$, $W-w_n$, g_1, g_2, ..., g_{n-1} und w_1, ..., w_{n-1} finde man eine Teilfolge g_{i_1}, g_{i_2}, ..., g_{i_r}, deren Summe nicht größer als $G-g_n$ und deren zugehörige Summe $w_{i_1} + w_{i_2} + ... + w_{i_r} \geq W-w_n$ ist.

Dieses Prinzip ist unmittelbar klar: Wenn es eine Lösung für P gibt, dann gehört der n-te Gegenstand entweder zur Lösung und dann hat P2 eine Lösung, oder er gehört nicht zur Lösung und dann hat P1 eine Lösung. Gibt es andererseits keine Lösung für P, dann können weder P1 noch P2 eine Lösung haben.

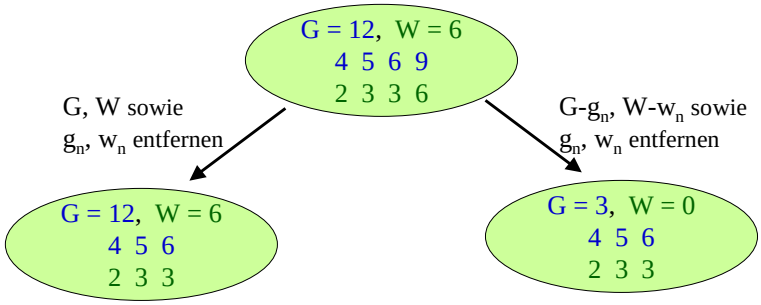
Also kann man die Frage, ob P eine Lösung hat, auf die Frage zurückführen, ob P1 oder P2 eine hat. Genau dieses besagt obiges Prinzip. Man hat damit ein "Problem der Größe n" auf zwei "Probleme der Größe n-1" zurückgeführt.

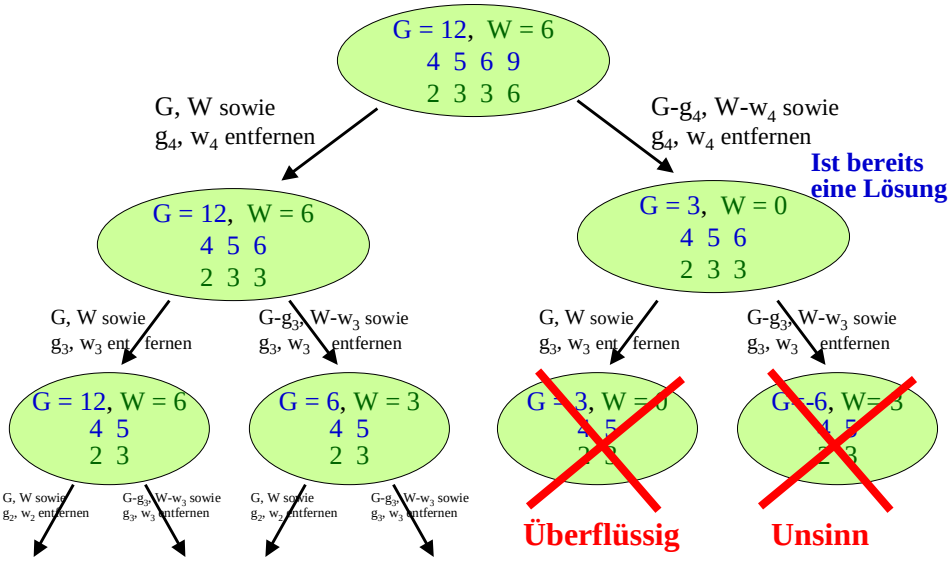
Das zugehörige Verfahren veranschaulicht man durch eine ständige Aufteilung in zwei Unterprobleme; man verfolgt beide Zweige (nacheinander) weiter, bis man unmittelbar entscheiden kann, ob eine Lösung vorliegt oder nicht; dies ist spätestens der Fall, wenn alle Elemente ausgesondert wurden (also für n=0).

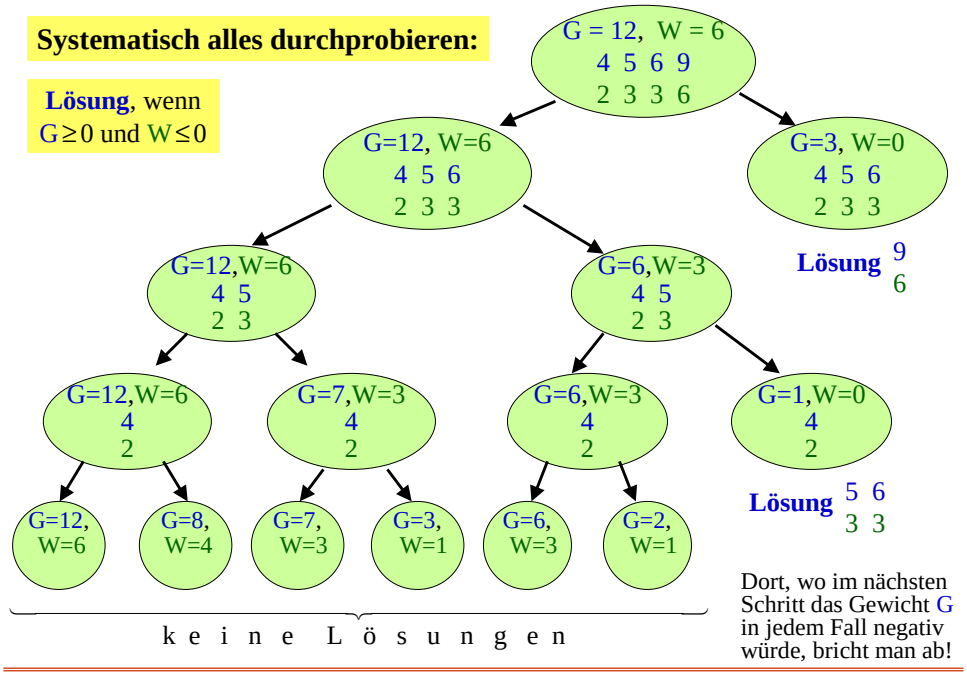
Veranschaulichung am Zahlen-Beispiel 1: G = 12, W = 6, n = 4

Folge der g_i: 4 5 6 9

Folge der zugehörigen w_i: 2 3 3 6







Ein vollständiges Probierverfahren durchläuft also einen (auf die Spitze gestellten) "Baum", bei dem sich nach unten in jedem Schritt die Zahl der durchzuprüfenden Fälle verdoppelt. Hin und wieder kann man an einer Stelle abbrechen, weil G negativ wird, aber hierdurch wird der Baum in der Regel nicht wesentlich kleiner. Dieses regelmäßige Verdoppeln führt zu einem "exponentiellen Wachstum" des Baumes und damit zu einer sehr hohen Laufzeit: Wenn man n Gegenstände vorgibt, so muss man mit 2^n Zeiteinheiten rechnen. Auch moderne Computer können dies höchstens bis n=35 in angemessener Zeit durchführen. Für größere Probleme muss man nach anderen Verfahren suchen.

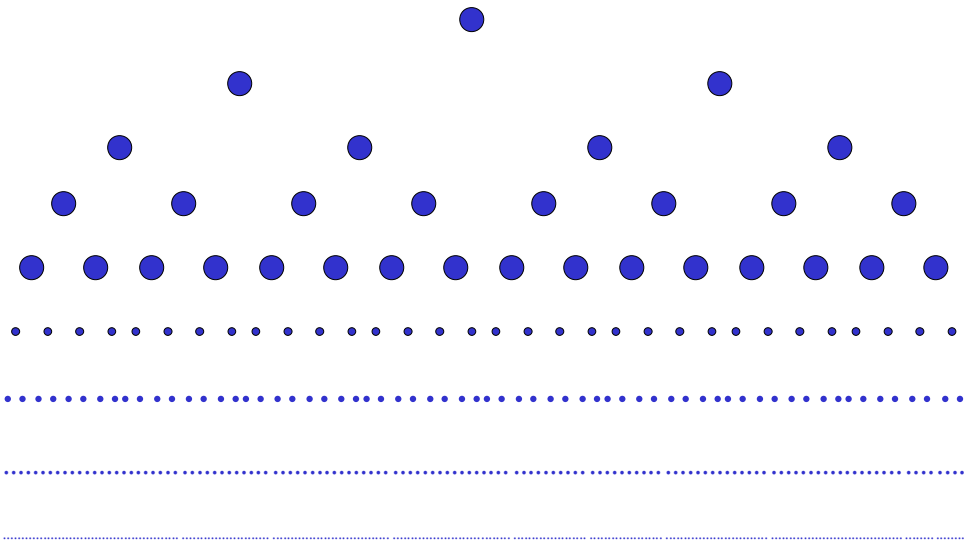
Dieses systematische Durchsuchen aller möglichen Fälle bezeichnet man als **Backtracking** (Rückverfolgen), bei dem man den Baum algorithmisch von rechts nach links durchläuft: Man wählt immer zunächst den rechten Zweig, ohne sich um alles, was links steht, zu kümmern, und sobald man auf den Fall stößt, dass G negativ wird, geht man zurück auf die darüber liegende Ebene und verfolgt dort den linken Zweig weiter. [Hier kann man links und rechts beim Durchlaufen natürlich auch generell vertauschen.]

Um eine Vorstellung von dem Wachstum des Baumes aller möglichen Fälle zu bekommen, ist auf der nächsten Folie die Verdoppelung in jedem Schritt dargestellt. Auf der obersten (der nullten) Ebene startet man mit $2^0 = 1$ Möglichkeiten, dann folgen auf der ersten Ebene $2^1 = 2$ Möglichkeiten, danach auf der zweiten Ebene $2^2 = 4$, dann $2^3 = 8$ usw. Auf der i-ten Ebene befinden sich 2^i Möglichkeiten, so dass bei n Ebenen insgesamt

$$2^0 + 2^1 + 2^2 + ... + 2^i + ... + 2^n = 2^{n+1} - 1$$

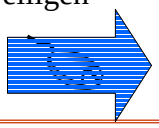
Stellen in diesem Baum existieren, die auszuwerten sind. Die Entscheidung, ob eine Lösung vorliegt, fällt meist erst auf der untersten, also der n-ten Ebene, auf der sich 2^n Elemente (sog. "Blätter") befinden.

Verdoppelung auf jeder Ebene. Hier: neun Ebenen von 2^0 bis 2^8 Elementen:



In unserem Beispiel konnte man bei $G < 0$ abbrechen, weil die nachfolgenden Fälle keine Lösungen mehr erlauben. Will oder muss man in der Praxis ein Backtracking einsetzen, so wird man versuchen, möglichst früh (also auf einer kleinen Ebene) die Fälle zu erkennen, unterhalb derer keine Lösungen mehr möglich sind. Anschaulich: Man muss möglichst frühzeitig die Zweige absägen, die nicht mehr zu einer Lösung führen können. Solche Verfahren bezeichnet man als **Branch-and-Bound**-Verfahren (Verzweige und Begrenze), da hier wie beim Backtracking immer in zwei oder mehr Unterbereiche verzweigt wird, aber bei bestimmten Situationen auch der Unterbaum abgeschnitten und somit der gesamte aufzubauende Baum deutlich begrenzt werden kann.

Branch-and-Bound-Verfahren sind meist stark vom jeweiligen Problem abhängig, so dass wir es bei dieser allgemeinen Bemerkung belassen wollen.



Definition: Allgemeines Rucksackproblem

Gegeben seien zwei natürliche Zahlen G und W und eine Menge von n Gegenständen, wobei jeder Gegenstand durch zwei natürliche Zahlen g_i und w_i, sein Gewicht und seinen Wert, charakterisiert ist.
Frage: Gibt es eine Teilmenge von Gegenständen, so dass die Summe ihrer Gewichte kleiner als G und die Summe ihrer Werte größer als W sind?

Formal:

Gibt es eine Teilmenge $\{i_1, i_2, ..., i_r\}$ der Menge $\{1, 2, ..., n\}$ mit

$$\sum_{j=1} g_{i_j} \leq G \quad \text{und} \quad \sum_{j=1} w_{i_j} \geq W \quad ?$$

Spezialfall: Man setze das Gewicht gleich dem Wert (dies ist der Fall, wenn die Gegenstände beispielsweise Goldklumpen sind). Dann erhält man:

Definition: Spezielles Rucksackproblem

Gegeben seien eine natürliche Zahl G und eine Folge von n Zahlen $g_1, g_2, ..., g_n$.
Frage: Gibt es eine Teilfolge, deren Summe gleich G ist?

Formal:

Gibt es eine Teilmenge $\{i_1, i_2, ..., i_r\}$ der Menge $\{1, 2, ..., n\}$ mit

$$\sum_{j=1}^{r} g_{i_j} = G \quad ?$$

Englische Bezeichnung: "**Subset Sum**"

Erneuter Spezialfall: Gerechte Erbschaftaufteilung

Das spezielle Rucksackproblem umfasst zugleich das Erbschaftsproblem: Jemand hinterlässt seinen zwei Erben n Gegenstände im Wert $g_1, g_2, ..., g_n$.
Gibt es eine Aufteilung, so dass beide Erben den gleichen Wert erhalten, d.h., gibt es eine Teilmenge $\{i_1, i_2, ..., i_r\}$ der Menge $\{1, 2, ..., n\}$ mit

$$\sum_{j=1}^{r} g_{i_j} = G \quad \text{wobei } G = \left(\sum_{i=1}^{n} g_i\right) / 2 \quad ?$$

(Falls die Summe der g_i nicht geradzahlig ist, so runde man nach unten oder verbiete solche Gewichte; die volle Schwierigkeit des Problems steckt bereits in den geradzahligen Summen.)

Englische Bezeichnung: "**Partition**"

In der englischsprachigen Literatur werden das spezielle Rucksackproblem als "Subset Sum" und das Erbschafts-problem als "Partition" bezeichnet.

Das Erbschaftsproblem sieht harmlos aus; es sind aber bis heute nur Algorithmen bekannt, die im allgemeinen Fall eine Lösung, sofern sie existiert, in exponentiell vielen Schritten liefern, also mit einem Aufwand proportional zu d^n für eine Konstante $d > 1$ (n = Zahl der Gegenstände).

Untersuchen Sie ein Beispiel:

Beispiel:

Für das Erbschaftsproblem betrachte folgende 20 natürlichen Zahlen (bereits geordnet aufgeschrieben, ihre Summe ist 1300 und somit G = 650):

32, 35, 40, 41, 44, 46, 51, 59, 60, 64,
72, 75, 76, 78, 80, 85, 86, 89, 92, 95

Die Frage lautet also: Man stelle fest, ob es eine Teilfolge dieser Zahlen gibt, deren Summe 650 ist.

Wer solche Zahlen einige Male untersucht hat, wird spontan behaupten, dass es zu obigem speziellen Beispiel eine Lösung geben wird. Warum?

Erläuterung dieser spontanen Behauptung:
Folgende Überlegung besagt, dass es für unser Beispiel mit recht hoher Wahrscheinlichkeit eine solche Teilfolge gibt: Insgesamt existieren 2^{20}, also ungefähr eine Million Teilfolgen, deren Summen zwischen 0 und 1300 liegen müssen. Es ist sehr unwahrscheinlich, dass die Zahl 650 nicht unter diesen eine Million Zahlen sein sollte.

Damit haben wir eine solche Folge aber noch nicht. Doch bei so hohen Wahrscheinlichkeiten kann man einige Zahlen einfach vorgeben (z.B. 40+60+80+85+95=360) und darauf hoffen, die Ergänzung (also 290) leicht zu entdecken. Durch Probieren findet man diese auch innerhalb kurzer Zeit (z.B. 32, 41, 59, 64, 92 oder 51, 72, 78, 89), so dass z.B. die Teilfolge 32, 40, 41, 59, 60, 64, 80, 85, 92, 95 das Problem löst.

Gefundene Teilfolge (rot): **32**,35,**40**,**41**,44,46,51,**59**,**60**,**64**,72,75,76,78,**80**,**85**,86,89,**92**,**95**

Nimmt man dagegen 20 Zahlen, die oberhalb von 2^{30} liegen, dann ist die Chance, dass die Erbschaft sich genau in zwei Teile teilen lässt, i.A. sehr gering; denn dann liegen die eine Million Summen der Teilfolgen ebenfalls oberhalb von 2^{30} (ungefähr eine Milliarde), und dann ist es sehr unwahrscheinlich, dass sich eine vorgegebene Zahl unter diesen Summen befindet.

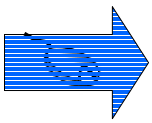
Wählt man aber die 20 Zahlen so, dass ihre Summe ungefähr 2^{20} beträgt, dann besteht immer noch eine hohe Chance, eine vorgegebene Zahl als Teilsumme darstellen zu können. Bei 2^{21} läge die Wahrscheinlichkeit in der Größenordnung von 0,5.

Fassen wir zusammen: Zu n Zahlen, deren Summe $2 \cdot G$ ist, gibt es 2^n Teilfolgen, deren Summen zwischen 0 und $2 \cdot G$ liegen müssen. Je tiefer G unterhalb von 2^n liegt, um so eher kann man damit rechnen, eine zufällig gewählte Teilfolge, deren Summe kleiner G sei, durch Ausprobieren zu einer Lösung ausbauen zu können. Je stärker G oberhalb von 2^n liegt, um so geringer wird (rein aus der Anzahl der Möglichkeiten betrachtet) die Wahrscheinlichkeit, dass sich die Folge in zwei gleiche Teilsummen aufspalten lässt. Man muss in diesem Fall meist alle Möglichkeiten durchprobieren, vor allem, wenn keine Lösung des Erbschaftsproblems existiert. Das Erbschaftsproblem ist also voraussichtlich dann nur mit dem größtmöglichen Aufwand zu lösen, wenn alle n Zahlen größer als $2^{n+1}/n$ sind, da in diesem Fall sicher $2 \cdot G > 2^{n+1}$, also $G > 2^n$ gilt.

Als Beispiel eignen sich daher besser die folgenden 20 Zahlen, deren Summe 33.480.070 ist, d.h., G = 16.740.035.

1.976.834, 1.864.558, 1.755.621, 1.575.931, 2.169.504,
1.567.429, 2.001.571, 1.682.544, 1.289.337, 1.223.752,
1.884.283, 1.671.449, 1.400.530, 1.547.733, 1.338.626,
1.438.792, 2.010.563, 1.422.589, 1.863.866, 1.794.558

Wir haben nicht nachgeprüft, ob für diese Zahlen eine Lösung des Erbschaftsproblems existiert. Untersuchen Sie diese Zahlen einige Minuten lang.



Systematisches Durchtesten (backtracking)

Das Backtracking-Verfahren haben wir am Beispiel des Rucksackproblems bereits kennen gelernt. Es führt ein Problem mit Parametern auf sich selbst, aber mit anderen Parameterwerten zurück, wobei die maximale Rekursionstiefe meist durch die Eingabewerte vorab feststeht.

Erinnerung an frühere Folien:
Dort wurde das Prinzip des Backtrackings speziell für das Rucksackproblem angegeben. Wir formulieren es nochmals in sehr allgemeiner kurzer Form (im allgemeinen Fall kann das Problem in mehr als zwei Teilprobleme zerlegt werden, siehe später "Binpacking"):

Man zerlege das Problem

P P mit Parametern a und k

in die beiden Probleme

P1 P mit Parametern a_1 und k-1

und

P2 P mit Parametern a_2 und k-1

Falls eine Terminierungsbedingung erfüllt ist (z.B. k=0, wobei k in irgendeiner Weise auch die Rekursionstiefe mitzählt), wird die Aufspaltung natürlich nicht mehr durchgeführt, sondern es wird getestet, ob eine Lösung vorliegt usw.
Vor und nach dem rekursiven Aufruf muss man das Problem in der Regel anpassen bzw. diese Anpassung rückgängig machen.

Dies lässt sich unmittelbar als Prozedurschema für das Backtracking (abgekürzt durch BT) formulieren:

```
procedure BT (a, k);
begin
    if  Terminierungsbedingung erfüllt  then  ....
    else   passe das Problem an die erste Rekursion P1 an;
           BT (a1, k-1);
           mache diese Anpassung wieder rückgängig und
           passe das Problem an die zweite Rekursion P2 an;
           BT (a2, k-1);
           mache diese Anpassung wieder rückgängig
    fi
end;
```


Anwenden auf das Rucksackproblem:

Die Variablen n, G, W: natural und g, w: array [1..n] of natural seien global. Sie enthalten die einzugebenden Werte für das Rucksackproblem.
Der Parameter a_1 bzw. a_2 aus dem Schema für das Backtracking ist beim Rucksackproblem das Paar (G, W) bzw. ($G-g_k$, $W-w_k$). Nun wird man hierbei nicht die vorgegebenen Werte für G und W verändern, vielmehr wird man sich zu jedem Zeitpunkt das Gewicht, das noch in den Rucksack maximal hinein gegeben werden kann, merken (Variable *RestG*) sowie den Wert (die Wichtigkeit), die noch bis zum Erreichen von W fehlt (Variable *RestW*). Die Parameter sind also (neben der noch verbleibenden Länge k der Folge) RestG, RestW: integer, die anfangs auf den Wert von G bzw. W zu setzen sind. ("integer" statt "natural", da diese Werte beim Subtrahieren negativ werden können.)

Nochmals: Die Variablen n, G, W: natural und g, w: array [1..n] of natural seien global. Die Parameter RestG, RestW, k: integer (anfangs auf den Wert von G bzw. W bzw. n zu setzen) enthalten die aktuellen Daten für das restliche, noch mögliche Gewicht, für den bisher im Rucksack noch fehlenden Wert und für "n minus der Tiefe der Rekursion".

```
procedure Rucksack (RestG: integer, RestW: integer, k: integer);
begin
    if k=0 then if (RestG≥0) and (RestW≤0) then "Es gibt eine Lösung" fi
    else Rucksack (RestG, RestW, k-1);
         Rucksack (RestG-g[k], RestW-w[k], k-1)
    fi
end;
```

Sobald die globalen Variablen n, G, W, g und w mit den richtigen Daten belegt sind, wird diese Prozedur mittels *Rucksack (G, W, n)* aufgerufen.

Diese Prozedur liefert nur eine Ja-Nein-Entscheidung. In der Regel möchte man auch mindestens eine Lösung, d.h., die Teilfolge ermitteln. Hierzu legen wir ein globales Boolesches Feld ja: array [1..n] of Boolean an. Am Ende der Rekursionen, d.h., bei k=0, gibt ja[i] jedes Mal an, ob das i-te Element der Folge zur aktuell betrachteten Teilfolge gehört oder nicht.

Das Feld ja wird anfangs mit false initialisiert. Immer, wenn man in die zweite Rekursion verzweigt, in der g[k] und w[k] von RestG bzw. RestW abgezogen werden, d.h., in der das Element k zur Teilfolge hinzu genommen wird, wird ja[k] auf true gesetzt. Kehrt man aus dieser Rekursion zurück, muss ja[k] wieder auf false zurückgesetzt werden.

Zur Erinnerung an frühere Folien: Eine Lösung liegt vor, wenn RestG≥0 und RestW≤0 sind; man kann zusätzlich k=0 annehmen (klar).

Globale Variable: n, G, W: natural; g, w: array [1..n] of natural;
ja: array [1..n] of Boolean;
Initialisierung: n, G, W und die Felder g und w werden eingelesen; das Feld ja ist anfangs komponentenweise auf false zu setzen.

```
procedure Rucksack (RestG: integer, RestW: integer, k: integer);
var i: natural;
begin
    if  k=0  then
        if  (RestG≥0) and (RestW≤0)  then  {Lösung gefunden}
            for i:=1 to n do  if ja[i] then drucke(i) fi  od;
            {hier: möglicher Abbruch}  fi
    else   Rucksack (RestG, RestW, k-1);
        ja[k] := true; Rucksack (RestG-g[k], RestW-w[k], k-1); ja[k] := false
    fi
end;
```

Will man nur eine Lösung haben, so bricht man die Prozedur an der Stelle *{hier: möglicher Abbruch}* ab. Anderenfalls werden alle Lösungen berechnet.

Dieses Schema des Backtrackings wenden wir nun noch auf das spezielle Rucksackproblem an. Zur Erinnerung die Definition:

Gegeben sind natürliche Zahlen n und G und eine Folge von n natürlichen Zahlen $g_1, g_2, ..., g_n$. Gibt es eine Teilmenge

$$\sum_{j=1}^{r} g_{i_j} = G \quad ?$$

$\{i_1, i_2, ..., i_r\}$ der Menge $\{1, 2, ..., n\}$ mit

Liest man G nicht ein, sondern setzt man anfangs $G := ½ \cdot \sum_{i=1}^{n} g_i$

so erhält man das Erbschaftsproblem.

Das Programm für das spezielle Rucksackproblem ergibt sich nun unmittelbar aus dem Programm des Rucksackproblems.

Globale Variable: n, G: natural; g: array [1..n] of natural;
ja: array [1..n] of Boolean;

n und das Feld g werden eingelesen, G wird ebenfalls eingelesen bzw. beim Erbschaftsproblem aus g berechnet; das Feld ja ist anfangs komponentenweise auf false zu setzen. Prozedur für das spezielle Rucksackproblem, die mit *SpezRucksack (G, n)* aufgerufen wird:

```
procedure SpezRucksack (RestG: integer, k: integer);
var i: natural;
begin
    if k=0 then
        if RestG = 0 then {Lösung gefunden}
            for i:=1 to n do if ja[i] then drucke(i) fi od;
            {hier: möglicher Abbruch} fi
    else SpezRucksack (RestG, k-1);
        ja[k] := true; SpezRucksack (RestG-g[k], k-1); ja[k] := false
    fi
end;
```


Bei dieser Formulierung des Algorithmus bricht die Rekursion stets erst auf der n-ten Stufe ab. Auch wenn RestG negativ geworden ist, arbeitet das Verfahren weiter, obwohl keine Lösung mehr möglich ist. Wir können die Rekursion sicher abbrechen, wenn RestG kleiner als das Minimum der verbleibenden Gewichte geworden ist. Wenn die Gegenstände nach der Größe ihrer Gewichte geordnet vorliegen, so kann man also die Rekursion auf jeden Fall abbrechen, wenn RestG < g[1] ≤ g[2] ≤ ... ≤ g[k] ist, da dann keine Veränderung an RestG mehr statt finden kann.

Dieses Abbruchkriterium für die Rekursion bezog sich auf die Gewichte. Für den Wert W gilt etwas Ähnliches: Man kann abbrechen, wenn RestW echt größer ist als die Summe der noch nicht betrachteten Werte, d.h., wenn RestW > w[1] + w[2] + ... + w[k] gilt. Um diese Summen nicht jedes Mal auswerten zu müssen, berechnet man sie einmal zu Beginn; man setzt also anfangs für j = 1, 2, ..., n:

sumw[j] := w[1] + w[2] + ... + w[j]

und bricht die Rekursion ab, falls RestW > sumw[k] gilt.
Wegen sumw[1] = w[1] und sumw[j] = sumw[j-1] + w[j] lassen sich diese Zahlen sehr schnell berechnen.

Die Abbruchkriterien für die Rekursion lauten also:

k = 0
RestG < g[1] bzw. RestG < ming (siehe unten)
RestW > sumw[k]

wobei der zweite und dritte Fall stets signalisieren, dass ausgehend von der aktuellen Situation keine Lösung durch weitere Rekursion gefunden werden kann. Für den zweiten Fall ist wichtig, dass die Zahlen sortiert sind. Will man dies nicht voraussetzen, so muss g[1] durch das Minimum "ming" der Menge {g[1], g[2], ..., g[n]} ersetzt werden.

Wir entscheiden uns hier dafür, keine Ordnung vorzugeben und ermitteln daher das Minimum ming zu Anfang mit. Diese Berechnungen lauten also:

```
ming:=g[1]; sumw[1] := w[1];
for j := 2 to n do
        if  ming > g[j]  then  ming := g[j]  fi;
        sumw[j] := sum[j-1] + w[j]
od
```

Damit ergibt sich folgendes Programm, das ein spezielles Backtracking, nämlich ein Branch-and-Bound-Verfahren darstellt.

Globale Variablen: n, G, W, ming: natural; g, w, sumw: array [1..n] of natural;
ja: array [1..n] of Boolean;

```
procedure Rucksack (RestG: integer, RestW: integer, k: integer);
var i: natural;
begin
    if  k=0  then
        if  (RestG≥0) and (RestW≤0)  then  {Lösung gefunden}
            for i:=1 to n do  if ja[i] then drucke(i) fi  od;
            {hier: möglicher Abbruch}  fi
    else if (RestG ≥ ming) and (RestW≤ sumw[k]) then
        Rucksack (RestG, RestW, k-1);
        ja[k] := true; Rucksack (RestG-g[k], RestW-w[k], k-1); ja[k] := false
    fi  fi
end;
begin .... < Einlesen der Werte für n, G, W, g, w >;
for j:=1 to n do ja[j] := false od; ming := g[1]; sumw[1] := w[1];
for j := 2 to n do if ming > g[j] then ming := g[j] fi; sumw[j] := sum[j-1] + w[j] od;
Rucksack (G,W,n); ....
end.
```

Nachdem der Algorithmus nun ausformuliert ist, muss man ihn in eine Programmiersprache übertragen. Die Übertragung z.B. nach C, Ada95 oder Java sollte für Sie kein Problem sein.

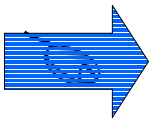
Hinweise: Bei dieser Übertragung sind Eigenarten der Sprachen zu beachten, z.B. die Angabe von oberen Schranken (statt dynamischer Felder) oder die Tatsache, dass manche Compiler große und kleine Buchstaben nicht unterscheiden (G und g haben also dann den gleichen Bezeichner). Nicht aufgeführt sind die genaue Beschreibung der Dateien, aus denen die Eingaben zu lesen oder in die die jeweiligen Ergebnisse zu schreiben sind.

Weiterhin muss man die Eingabeanweisungen sowie Ergebnis- und Kontrollausdrucke einfügen. In der Praxis ist n=35 wegen der Laufzeit eine obere Grenze für die Anzahl der Gegenstände, es sei denn, die Werte für G und W liegen so günstig, dass durch den Branch-and-Bound-Effekt nur ein kleiner Teil des riesigen Baumes durchsucht werden muss.
Ein Pascal-Programm finden Sie auf der nächsten Folie.


```
program Rucksackproblem; { Programm in PASCAL formuliert }
const  max = 40;
var   n, i, j: int;  Gew, Wert, ming: longint;  g, w, sumw: array[1..max] of longint;
      ja: array [1..max] of Boolean;
procedure Rucksack (RestG, RestW, k: longint);
begin  if  k=0  then
            begin if  ((RestG >= 0) and (RestW <= 0))  then  {Lösung gefunden}
                  begin writeln; for i:=1 to n do  if  ja[i]  then write (i) end; writeln
            end
      else if (RestG ≥ ming) and (RestW≤ sumw[k]) then
            begin  Rucksack (RestG, RestW, k-1);
                  ja[k] := true; Rucksack (RestG-g[k], RestW-w[k], k-1); ja[k] := false
                  end;
      if k=n then writeln ('Ende')
end;
begin  read (n);
  if n > max then  writeln ('Eingabe größer als ', max:3, ', Abbruch.')
  else begin read (Gew); read (Wert); writeln; for i:=1 to n do read (g[i], w[i]);
            {Kontrollausgabe:} writeln ('Rucksackproblem mit Gewicht ',Gew:12,' und Wert ',Wert:12,'.');
            writeln ('Es sind ', n:12,' Gegenstände:'); for i:=1 to n do  writeln (g[i]:10, '    ', w[i]:10);
            {Initialisierungen:} for j := 1 to n do ja[j] := false;     ming := g[1]; sumw[1] := w[1];
            for j := 2 to n do begin if ming > g[j] then ming := g[j]; sumw[j] := sum[j-1] + w[j] end;
            Rucksack (Gew, Wert, n)
       end
end.
```

Wir haben das Rucksackproblem behandelt. Die Leser(innen) können durch Weglassen und leichtes Modifizieren leicht Programme für das spezielle Rucksackproblem oder für das Erbschaftsproblem schreiben.
Gibt es weitere Probleme, die man mit Backtracking lösen kann (oder mangels besserer Verfahren lösen muss)?

Am bekanntesten ist das Bin-Packing-Problem (BPP), das wir im nächsten Abschnitt vorstellen werden. Wir werden zugleich zeigen, dass es eine Erweiterung des speziellen Rucksackproblems ist.



Binpacking

Ausgehend vom Füllproblem haben wir als eindimensionalen Sonderfall das Rucksackproblem und zwei Vereinfachungen (spezielles Rucksackproblem, Erbschaftsproblem) vorgestellt, die Lösungsmethode Backtracking herausgearbeitet und als Programm in einer Programmiersprache formuliert. Das Backtracking ist ein sehr allgemeines Lösungsverfahren für viele Probleme. In diesem Abschnitt wenden wir das Verfahren auf das Binpacking-Problem (BPP) an, eines der ältesten Probleme zum Thema Backtracking.

Zum Vorgehen: Zunächst gehen wir wieder von einem anschaulichen Beispiel aus. Als dessen Formalisierung erhalten wir das BPP. Dann stellen wir das Binpacking mit anderen Problemen in Beziehung und lösen es anschließend mit der Backtracking-Methode.

Beispielproblem: Ein Spediteur besitzt **m** Lastwagen des gleichen Typs. Jeder Lastwagen darf mit maximal **G** Gewichtseinheiten beladen werden. Eine Kunde möchte **n** Gegenstände, die jeweils die Gewichte $\mathbf{g_1}$, $\mathbf{g_2}$, ..., $\mathbf{g_n}$ besitzen, zu einem anderen Ort transportieren lassen. Reichen hierfür die m Lastwagen aus? (Hier interessiert nur das Gewicht, nicht die Form der Gegenstände.)

Dieses Problem können wir unmittelbar in eine Definition fassen:

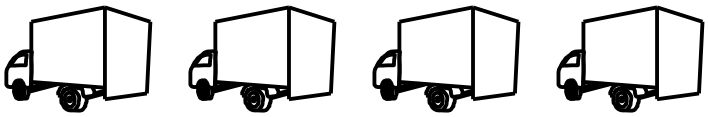
Definition: Binpacking

Gegeben: natürliche Zahlen **G**, **m**, **n**, $\mathbf{g_1}$, $\mathbf{g_2}$, ..., $\mathbf{g_n}$.
Frage: Gibt es eine Aufteilung der n Zahlen g_i auf m Teilmengen, so dass die Summe der Zahlen in jeder Teilmenge kleiner oder gleich G ist?

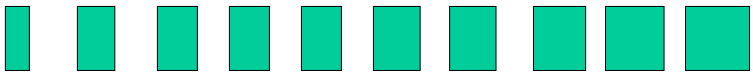
Der Name des Problems kommt vom englischen Wort "bin" = Behälter, Kasten. Diese Behälter sollen optimal mit den n Gegenständen "bepackt" werden.

Schauen wir uns das Problem an Beispielen an.

Beispiel 1: m = 4, G = 20, n = 10, die Gewichte der zehn Gegenstände:
g_1= 3, g_2= 5, g_3= 6, g_4= 6, g_5= 6, g_6= 7, g_7= 7, g_8= 8, g_9= 9, g_{10}= 10.
Die Gegenstände wiegen insgesamt 67 Einheiten.



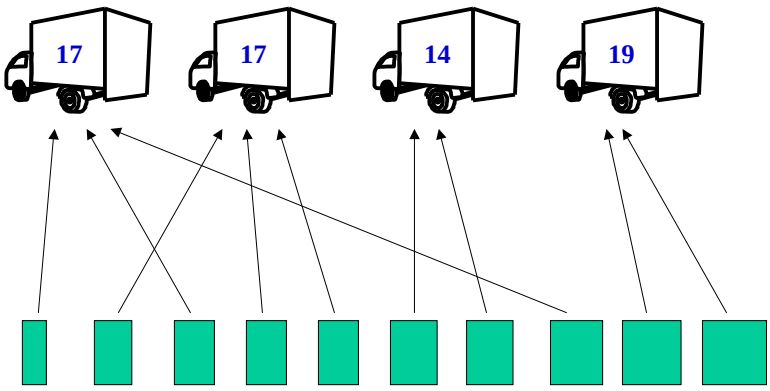
Nun müssen wir die Gegenstände einpacken!



g_1= 3, g_2= 5, g_3= 6, g_4= 6, g_5= 6, g_6= 7, g_7= 7, g_8= 8, g_9= 9, g_{10}= 10

Wir ordnen irgendwie von links nach rechts zu:

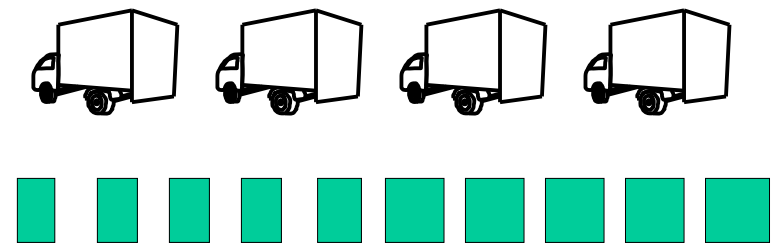
Wir prüfen, ob das Maximalgewicht G=20 irgendwo überschritten wurde:



g_1= 3, g_2= 5, g_3= 6, g_4= 6, g_5= 6, g_6= 7, g_7= 7, g_8= 8, g_9= 9, g_{10}= 10

Also haben wir eine Lösung gefunden. (Es gibt noch viele weitere, selbst suchen.)

Beispiel 2: m = 4, G = 20, n = 10, die Gewichte der zehn Gegenstände:
g_1= 5, g_2= 6, g_3= 6, g_4= 6, g_5= 7, g_6= 9, g_7= 9, g_8= 9, g_9= 9, g_{10}= 10.
Die Gegenstände wiegen insgesamt 76 Einheiten.

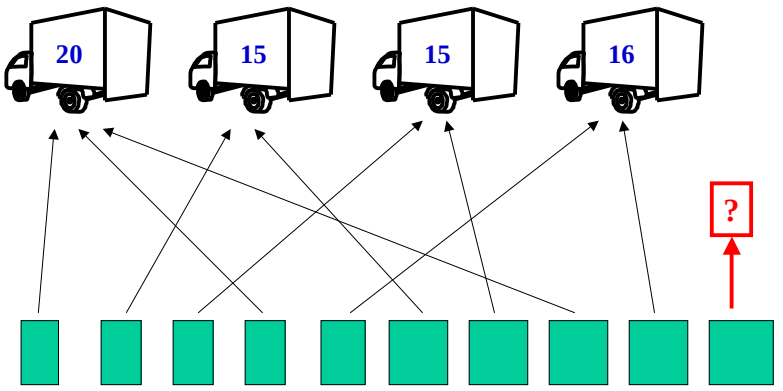


g_1= 5, g_2= 6, g_3= 6, g_4= 6, g_5= 7, g_6= 9, g_7= 9, g_8= 9, g_9= 9, g_{10}= 10

Nun versuchen wir, die Gegenstände einzupacken.

Wir ordnen erneut irgendwie von links nach rechts zu:

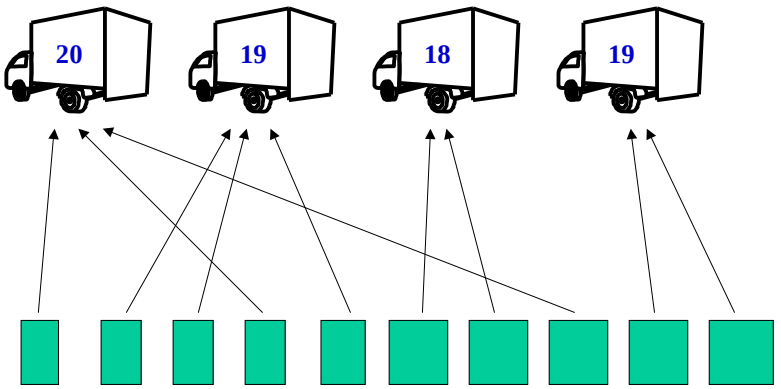
Wir prüfen, ob das Maximalgewicht G=20 irgendwo überschritten wurde:



g_1= 5, g_2= 6, g_3= 6, g_4= 6, g_5= 7, g_6= 9, g_7= 9, g_8= 9, g_9= 9, g_{10}= 10

Die "10" lässt sich hier nicht mehr unterbringen. Aber: Es gibt Lösungen:

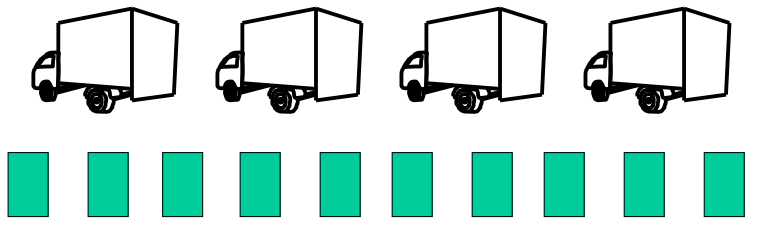
Wir prüfen, ob das Maximalgewicht G=20 irgendwo überschritten wurde:



g_1= 5, g_2= 6, g_3= 6, g_4= 6, g_5= 7, g_6= 9, g_7= 9, g_8= 9, g_9= 9, g_{10}= 10

Somit haben wir eine Lösung für Beispiel 2 gefunden.

Beispiel 3: m = 4, G = 20, n = 10, die Gewichte der zehn Gegenstände:
g_1= 7, g_2= 7, g_3= 7, g_4= 7, g_5= 7, g_6= 7, g_7= 7, g_8= 7, g_9= 7, g_{10}= 7.
Die Gegenstände wiegen insgesamt 70 Einheiten.



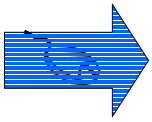
g_1= 7, g_2= 7, g_3= 7, g_4= 7, g_5= 7, g_6= 7, g_7= 7, g_8= 7, g_9= 7, g_{10}= 7

Beim Einpacken werden wir offensichtlich scheitern; denn in jeden Lastwagen passen wegen G = 20 höchstens zwei Gegenstände.

Beispiel 3 hat also keine Lösung.

Zur Einordnung des Problems:

Das BPP verallgemeinert das Rucksackproblem von einem auf viele Rucksäcke. (Hierbei setzen wir den zu erreichenden Wert W auf 0.)
Wir zeigen nun, dass das spezielle Rucksackproblem (vgl. dessen Definition) und das Erbschaftsproblem Sonderfälle des BPP für m=2 Behälter sind.



Hilfssatz 1:

Es seien eine natürliche Zahl G und eine Folge von n Zahlen $g_1, g_2, ..., g_n$ gegeben. Bilde die Summe aller Gewichte $SG = g_1 + g_2 + ... + g_n$. Ohne Beschränkung der Allgemeinheit sei $2 \cdot G \leq SG$. Dann gilt:

Das spezielle Rucksackproblem besitzt für die Zahlen

$G, n, g_1, g_2, ..., g_n$ (mit $2 \cdot G \leq SG$)

genau dann eine Lösung, wenn das Binpackingproblem für

$SG-G, 2, n+1, g_1, g_2, ..., g_n, g_{n+1}$

mit $g_{n+1} = SG- 2 \cdot G$ eine Lösung hat.

Veranschaulichung: Gegeben sei das spezielle Rucksackproblem für die Zahlen

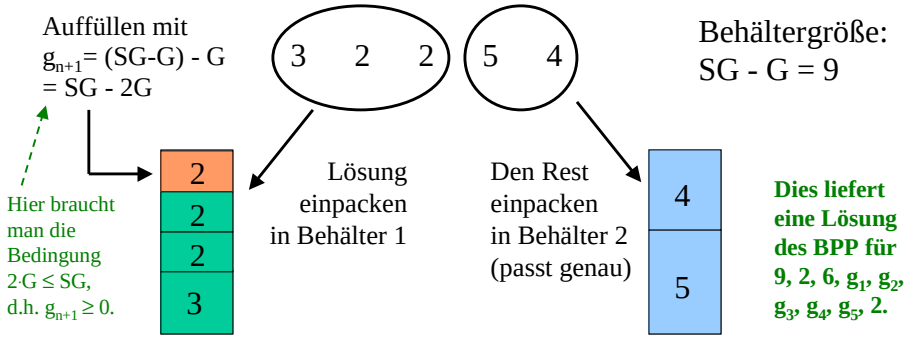
$G=7, n=5, g_1=3, g_2=2, g_3=2, g_4=5, g_5=4.$

Dann ist $SG = 16$ und es gilt $14 = 2 \cdot G \leq SG = 16$.

Eine Lösung ist $\{1,2,3\}$: $3 + 2 + 2 = 7 = G$.

Hieraus entwickeln wir folgende Binpacking-Aufgabe:

Gegebenes spezielles Rucksackproblem:



(Formaler) Beweis:

Wir beweisen zunächst den Hilfssatz und erläutern dann, warum man *ohne Beschränkung der Allgemeinheit* $2 \cdot G \leq SG$ annehmen kann.

Beweisrichtung "⇒":

Das spezielle Rucksackproblem für G, n, $g_1, g_2, ..., g_n$ möge eine Lösung haben, und zwar sei die Gewichtssumme der Teilmenge $L=\{g_{i_1}, g_{i_2}, ..., g_{i_r}\}$ genau G. Wir füllen nun die Gegenstände dieser Teilmenge zusammen mit dem (n+1)-ten Gegenstand in den ersten der beiden Behälter. Ihr Gewicht ist:
$g_{i_1} + g_{i_2} + ... + g_{i_r} + g_{n+1} = G + (SG-2 \cdot G) = SG-G$,
der erste Behälter wird hierdurch also bis zu seinem Maximalgewicht gefüllt.

Alle übrigen Gegenstände füllen wir in den zweiten Behälter. Das Gewicht aller dieser Gegenstände beträgt SG-G. Hierdurch wird also auch der zweite Behälter bis zu seinem Maximalgewicht (SG-G) aufgefüllt.
Diese Aufteilung auf die beiden Behälter ist folglich eine Lösung des Binpackingproblems für SG-G, 2, n+1, $g_1, g_2, ..., g_n, g_{n+1}$ mit $g_{n+1} = SG- 2 \cdot G$, was zu beweisen war.

Beweisrichtung "⇐":
Wenn eine Lösung des BPP für SG-G, 2, n+1, g_1, g_2, ..., g_n, g_{n+1} mit g_{n+1} = SG-2·G vorliegt, dann lassen sich die n+1 Gegenstände auf zwei Behälter mit jeweils maximalem Gewicht SG-G aufteilen. Dieses Maximalgewicht wurde geschickt gewählt, denn es gilt:

$2\cdot(SG-G) = SG - 2\cdot G + SG = g_{n+1} + g_1 + g_2 + ... + g_n$.

Also werden durch die n+1 Gegenstände beide Behälter bis zu ihrem Maximalgewicht gefüllt.

In einem der beiden Behälter muss der (n+1)-te Gegenstand mit dem Gewicht g_{n+1} = SG- 2·G liegen. Da er ganz gefüllt ist, muss das restliche Gewicht in diesem Behälter

$$(SG-G) - g_{n+1} = (SG-G) - (SG-2\cdot G) = G$$

sein. Folglich muss es eine Teilmenge der n Gegenstände geben, deren Gewichtssumme gleich G ist, d.h., das spezielle Rucksackproblem für G, n, g_1, g_2, ..., g_n hat eine Lösung, was zu beweisen war.

Damit ist Hilfssatz 1 bewiesen.

Nachtrag: Wir hatten behauptet, dass man ohne Beschränkung der Allgemeinheit 2·G ≤ SG annehmen dürfe. Begründung hierfür:

Es seien eine natürliche Zahl G und eine Folge von n Zahlen g_1, g_2, ..., g_n gegeben. Bilde die Summe aller Gewichte SG = $g_1 + g_2 + ... + g_n$. Wenn dieses spezielle Rucksackproblem eine Lösung besitzt, dann gilt: Es gibt eine Teilmenge L = $\{i_1, i_2, ..., i_r\}$ der Zahlen {1, 2, ..., n} mit

$$\sum_{j=1}^{r} g_{i_j} = G.$$

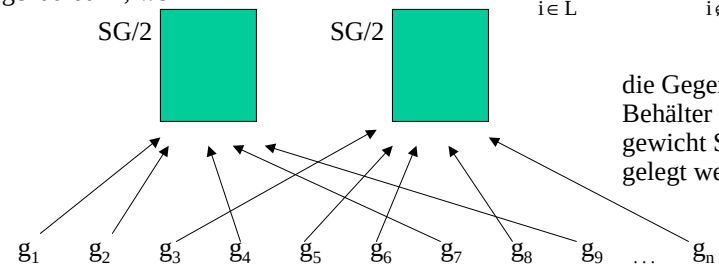
Dann gilt für die anderen Gegenstände: $\sum_{i \notin L} g_i = SG-G.$

Also bildet eine Lösung bzgl. des Gewichts G zugleich eine Lösung bzgl. des Gewichts SG-G, indem man die komplementäre Teilmenge $\{1, 2, ..., n\} - \{i_1, i_2, ..., i_r\}$ verwendet. Wenn daher 2·G > SG ist, so löse man stattdessen das Problem mit dem Gewicht SG-G, denn hierfür gilt:
2·(SG-G) ≤ SG+SG-2·G < SG, d.h., dann ist die geforderte Bedingung erfüllt.

Bemerkung:

Beim Erbschaftsproblem ist sofort klar, dass es ein Binpackingproblem mit 2 Behältern und der Behältergröße SG/2 ist. Denn hier wird festgestellt, ob sich eine Folge von n Zahlen so in zwei Teilfolgen zerlegen lässt, dass deren Summen gleich sind. Formal: Sei für eine Folge von n Zahlen $g_1, g_2, ..., g_n$ die Summe aller Gewichte $SG = g_1 + g_2 + ... + g_n$, so hat das Erbschaftsproblem genau dann eine Lösung, wenn das Binpackingproblem für SG/2, 2, n, $g_1, g_2, ..., g_n$ eine Lösung hat. Denn: Es gibt eine Teilmenge $L = \{i_1, i_2, ..., i_r\}$ der Zahlen {1, 2, ..., n} mit genau dann, wenn

$$\sum_{i \in L} g_i = SG/2 = \sum_{i \notin L} g_i$$



die Gegenstände in 2 Behälter mit Maximal-gewicht SG/2 gelegt werden können.

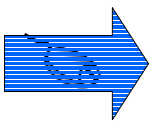
Was besagt der Hilfssatz 1 ?

Er vergleicht die beiden Probleme "spezielles Rucksackproblem" und "Binpacking mit 2 Behältern", wobei er nachweist, dass das Binpacking mit 2 Behältern mindestens so schwierig wie das spezielle Rucksackproblem ist.

Anders ausgedrückt: Wenn es ein Verfahren gibt, das das Binpacking für zwei Behälter schnell löst, so lässt sich das spezielle Rucksackproblem ebenfalls mit diesem Zeitaufwand lösen.

Der Beweis zeigt zugleich, wie sich jedes spezielle Rucksackproblem in das Binpackingproblem einbetten lässt. (Sog. "Reduktion" des einen Problems auf das andere; vgl. Kurse über Komplexitätstheorie.)

Wir wenden uns nun der Programmierung des Backtrackingverfahrens für das Binpacking zu.



Hierzu müssen wir eine möglichst präzise Definition haben. Die Definition des Binpackings lautete: Finde eine Aufteilung von n Zahlen auf höchstens m Teilmengen, deren Summe jeweils kleiner oder gleich G ist. Wir formulieren dies nun nochmals mathematisch:

Definition: Binpacking

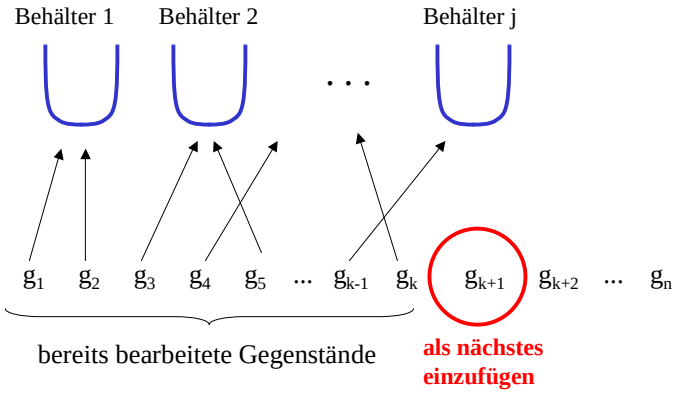
Gegeben: natürliche Zahlen **G**, **m**, **n**, $\mathbf{g_1}$, $\mathbf{g_2}$, ..., $\mathbf{g_n}$.
Gesucht: eine Abbildung f: {1, 2, ..., n} → {1, 2, ..., m},
so dass für jedes i (1≤ i≤ m) gilt:

$$\sum_{f(j)=i} g_j \leq G.$$

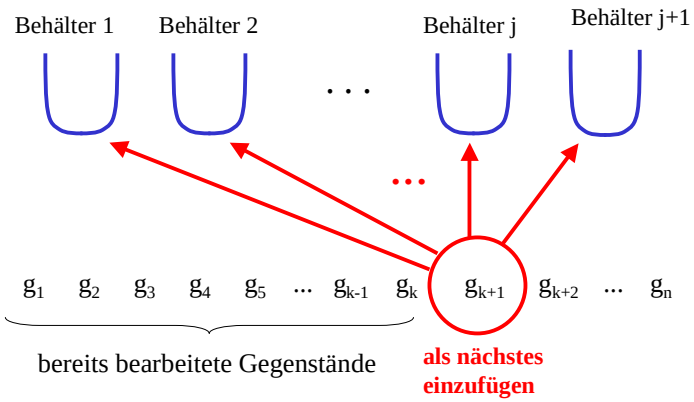
f(j) = i bedeutet also:
Der j-te Gegenstand wird in den i-ten Behälter gelegt.

Um eine Lösung zu finden, probieren wir alle Abbildungen durch, sofern sie eine Lösung bilden könnten. Hierfür verwenden wir das Backtracking.

Backtracking geht stets von einer vorhandenen Situation aus und reduziert das Problem auf einfachere Probleme. Eine typische Situation beim Füllen der Behälter ist: Man hat die ersten k Gegenstände bereits konfliktfrei in j Behälter gefüllt und muss nun den (k+1)-ten Gegenstand einfügen.



Wir können den (k+1)-ten Gegenstand nun einem der bereits vorhandenen Behälter 1, 2, ..., j zuordnen oder einen neuen Behälter (j+1) verwenden, sofern j+1 ≤ m, d.h., j < m ist.



Alle diese j+1 Fälle probieren wir im Backtracking durch. Die Prozedur folgt genau der obigen Rekursion.

Schema der Prozedur (es fehlen noch einige Details):

```
procedure BTBPP (k, j: natural);
var i: natural;
begin
      if  k = n  then  ..... {Lösung gefunden} .....
      else for i:=1 to j do
                  if  Gegenstand (k+1) passt noch in Behälter i
                                    then Aufruf von BTBPP (k+1, j)  fi   od;
            if j < m then Aufruf von BTBPP (k+1, j+1) fi
      fi
end;
```

Um zu überprüfen, ob ein Gegenstand (k+1) noch in den Behälter i passt, verwenden wir ein globales Feld GBeh: array[1..m] of natural, in welchem wir die Summe der Gewichte aller Gegenstände, die aktuell im jeweiligen Behälter liegen, notieren. Ein Gegenstand (k+1) darf in den Behälter i nur gelegt werden, wenn dadurch das Maximalgewicht nicht überschritten wird, also nur, wenn GBeh[i]+g[k+1] ≤ G ist.

So erhalten wir die Verfeinerung des Prozedurschemas (man beachte, dass vor und nach jedem rekursiven Aufruf das Gewicht des jeweiligen Behälters aktualisiert werden muss):

```
procedure BTBPP (k, j: natural);
var i: natural;
begin
      if  k = n  then  ..... {Lösung gefunden} .....
      else for i:=1 to j do
                  if  GBeh[i] + g[k+1] ≤ G  then
                        GBeh[i]:=GBeh[i] + g[k+1]; BTBPP (k+1, j);
                        GBeh[i]:=GBeh[i] - g[k+1]  fi  od;
            if j < m then GBeh[j+1]:=g[k+1];
                        BTBPP (k+1, j+1); GBeh[j+1]:=0  fi
      fi
end;
```


Diese Prozedur stellt bisher nur fest, ob eine Lösung existiert, aber sie gibt keine mögliche Zuordnung zu den Behältern aus. Hierfür müssen wir uns noch zu jedem Gegenstand merken, in welchen Behälter er gelegt wurde. Dies geschieht in einem globalen Feld f: array [1..n] of natural. Wir nennen dieses Feld f, weil es genau die Abbildung f aus der Definition beschreibt.

Wir brauchen nicht alle möglichen Abbildungen f durchzuprobieren; denn wir können annehmen, dass der Gegenstand mit der Nummer 1 stets im Behälter 1 liegt, der Gegenstand mit der Nummer 2 stets in einem der Behälter 1 oder 2 usw. Durch Umnummerieren der Behälter lässt sich also stets erreichen, dass $f[k] \leq k$ für alle $k = 1, 2, ..., n$ gilt. Wir werden daher f[1]:=1 setzen. In der Prozedur stellen wir automatisch sicher, dass $f[k] \leq k$ für alle weiteren k gilt, indem für den Gegenstand k+1 neben den bereits betrachteten Behältern nur der Behälter j+1 (und nicht j+2, j+3 usw.) ausprobiert wird. Weil m die höchste Nummer eines Behälters ist, brauchen wir also nur Abbildungen f zu betrachten mit: $f(k) \leq \text{Min}(k, m)$,
wobei Min(k, m) das Minimum der beiden Zahlen k und m ist.

So erhalten wir als Backtrackingprozedur für das Binpackingproblem:

```
procedure BTBPP (k, j: natural);
var i: natural;
begin
      if  k = n  then  < Lösung gefunden: drucke das Feld f aus >
      else for i:=1 to j do
                  if  GBeh[i] + g[k+1] ≤ G  then
                        f[k+1]:=i; GBeh[i]:=GBeh[i] + g[k+1]; BTBPP (k+1, j);
                        GBeh[i]:=GBeh[i] - g[k+1]; f[k+1]:=0  fi  od;
            if j < m then f[k+1]:=j+1; GBeh[j+1]:=g[k+1];
                        BTBPP (k+1, j+1); GBeh[j+1]:=0; f[k+1]:=0  fi
      fi
end;
```

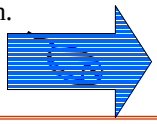

Globale Variablen sind:

```
G, m, n: natural;
g, f: array [1..n] of natural;
GBeh: array [1..m] of natural;
```

Der Aufruf der Prozedur BTBPP lautet, sofern bereits alle Daten in die globalen Variablen G, m, n und g eingelesen wurden:

```
if (n<1) or (m<1) then ...{Abbruch, da keine Lösung zu suchen ist}... fi;
for i:=1 to n do if g[i]>G then ...{dieses g[i] kann man weglassen}... fi od;
for i:=1 to n do f[i]:=0; GBeh[i]:=0 od;
GBeh[1]:=g[1]; f[1]:=1; BTBPP(1,1);
```

Manche der obigen Anweisungen erscheinen überflüssig (z.B. f[k+1]:=0 oder for i:=1 to n do f[i]:=0; GBeh[i]:=0 od;). Wir haben sie dennoch aufgeführt, da sie eventuell hilfreich werden können, wenn Fehler auftreten oder wenn die Prozedur noch von anderen Programmen aufgerufen wird. Aber sicher kann man eine konkrete Implementierung noch effizienter formulieren.



Eigenschaften der Prozedur BTBPP

Wir haben bereits gesehen, dass nur Lösungen mit

$$f(k) \leq \text{Min}(k,m) \quad \text{für } k = 1, ..., n$$

untersucht werden.

Weiterhin werden wir erwarten, dass die Prozedur die Behälter tatsächlich stets von links nach rechts auffüllt, dass sich unter den ersten j Behältern (sofern der Behälter j nicht leer ist) also niemals ein leerer Behälter befindet; denn sonst würden wir Situationen mehrmals testen, die bis auf Umnummerierung der Behälter gleich sind. Ist durch unsere Prozedur sicher gestellt, dass unter den ersten j Behältern keine leeren auftreten?

Dies ist tatsächlich der Fall, wie folgende Überlegung zeigt.

Zu Beginn wird in Behälter 1 durch GBeh[1]:=g[1] der erste Gegenstand gelegt. Er wird hieraus nicht wieder entfernt. Wir nehmen nun an, dass die ersten k Gegenstände in den Behältern von 1 bis j liegen und dass keiner dieser j Behälter leer ist (Induktionsannahme). Diese Annahme ist für k=1 richtig, wobei hier automatisch j=1 ist. Die Prozedur versucht nun, den nächsten Gegenstand k+1 in einen bereits existierenden Behälter zu legen (for i:=1 to j do ... od). Dabei kann kein vorhandener Behälter geleert werden; in diesem Fall bleibt daher die Induktionsannahme richtig. Anschließend wird der Gegenstand k+1 in den neuen Behälter (j+1) gelegt, sofern j < m ist. Dieser (j+1)-te Behälter ist also nicht leer, so dass die Induktionsannahme auch für den Fall k+1 richtig bleibt. (Nach der Rückkehr aus diesem letzten Rekursionsfall wird der Behälter j+1 geleert, aber j+1 wieder durch j ersetzt, so dass die Induktionsannahme nicht verletzt wird.) Durch Induktion folgt daher, dass sich unter den j ersten Behältern keine leeren befinden können.

Zur Analyse der Laufzeit:

Beim ersten Gegenstand gibt es nur eine Möglichkeit, beim zweiten zwei Möglichkeiten (lege den Gegenstand in den Behälter 1 oder in den Behälter 2), beim dritten drei usw. Ab dem m-ten Gegenstand hat man für jeden Gegenstand nur noch höchstens m Möglichkeiten.

Somit werden im ungünstigsten Fall

$$1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot m \cdot m \cdot \ldots \cdot m = m! \cdot m^{(n-m)}$$

Möglichkeiten durchprobiert (m! ist die Fakultätsfunktion). Wir müssen also größenordnungsmäßig mit m^{n-1}/e^m Schritten (siehe Stirlingsche Formel für die Fakultät, e ≈ 2,71828...) rechnen. Dies war zu erwarten, da BPP, wie gezeigt, mindestens so aufwändig wie das spezielle Rucksackproblem ist, dessen Backtracking-Bearbeitung bereits 2^n Schritte benötigt. In der Praxis ist das Verfahren BTBPP also nur für wenige Gegenstände einsetzbar.

Zur Analyse des Speicherplatzes:

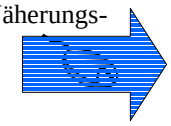
Zusätzlich zu den globalen Variablen wird weiterer Speicherplatz nur durch die Rekursion erforderlich. Da die Rekursionstiefe, also die maximale Zahl ineinander geschachtelter Prozeduraufrufe, höchstens n ist (denn dann erfolgt der Abbruch durch "if k = n ..."), brauchen wir daher nur "linear viel zusätzlichen Speicherplatz", d.h., zusätzliche c·n Speicherplätze für eine geeignete Konstante c.

Diese Konstante hängt auch von der Implementierung der Rekursion in der jeweiligen Programmiersprache ab, so dass sie nicht ohne eine konkrete Implementierung angegeben werden kann. Dennoch kann man grob geschätzt sagen, dass man etwa den gleichen Speicherplatz, der für die Eingabe erforderlich ist, nochmals benötigt. Dies ist aber bei heutigen Rechenanlagen unproblematisch.

Die Analyse von Speicherplatz und Laufzeit zeigt: Für praktisch relevante Aufgabenstellungen ist das Backtracking ungeeignet. Wir brauchen also schnellere Verfahren.

Wir kennen zur Zeit keine Verfahren, die eine Lösung des Binpackings, sofern sie existiert, zuverlässig finden und zugleich größenordnungsmäßig weniger Zeit als d^n Schritte (für eine Konstante $d > 1$) benötigen; man vermutet, dass es solche Verfahren nicht gibt. Daher hat man Näherungsverfahren entwickelt, die schnell sind und häufig eine Lösung, sofern sie existiert, finden; wenn solch ein Verfahren aber keine Lösung entdeckt, so kann es trotzdem eine Lösung geben.

Wie sehen solche Näherungsverfahren aus? Vermutlich haben Sie eines schon unbewusst benutzt, als Sie Lösungen zu den vorgestellten Beispielen gesucht haben: Man sortiere zunächst die Gegenstände nach ihrer Größe (beginnend mit dem größten) und platziere den jeweils nächsten Gegenstand in den Behälter mit der kleinsten Nummer, in den er noch passt. Auf Näherungsverfahren werden wir aber in diesem Modul nicht eingehen.



Schlussbemerkung

Backtracking ist ein zentrales Prinzip in der Informatik: Es ist die Technik des systematischen Aufzählens von Bereichen, deren Lösungen sich baumartig aus Teillösungen ermitteln lassen. Hierbei wird der Baum rekursiv durchlaufen, wodurch die algorithmische Formulierung recht einfach wird. Gut sichtbar wird dies beim speziellen Rucksackproblem: Man baut die Lösung systematisch als 0-1-Vektor auf, dessen i-te Komponente x_i festlegt, ob der i-te Gegenstand in den Rucksack gelegt wird ($x_i=1$) oder nicht ($x_i=0$).
Die Anwendungsmöglichkeiten des Backtracking sind schier unbegrenzt.

In der Regel wächst die Laufzeit von Backtracking-Verfahren exponentiell mit der Anzahl der Gegenstände oder Positionen. Für kleine Anzahlen n ist dieses Verfahren noch ausführbar, für größere nicht. Da es sich relativ leicht programmieren lässt und auf jeden Fall das Optimum findet, lassen sich hiermit für einige Werte die exakten Lösungen berechnen, die dann als Test für kompliziertere oder für Näherungsverfahren verwendet werden können.

Für größere Anzahlen versucht man, den Baum zu beschneiden, indem man Unterbäunme, die nicht mehr zu einer Lösung führen können, nicht besucht (Branch-and-Bound-Technik). Hierbei muss man aber stets Eigenschaften des Problems in das Verfahren einfließen lassen.

In der Literatur bezeichnet man Backtracking auch als "Versuch-und-Irrtum"-Methode (*Trial-and-Error-Verfahren*), da es einen Lösungskandidaten zu einer Lösung auszubauen sucht ("Versuch") und jedes Mal, wenn es hierbei in eine Sackgasse ("Irrtum") gerät, den nächsten Versuch startet.

Standardbeispiele für Probleme, deren Lösung man mittels Backtracking leicht beschreiben kann, sind:
- Rucksackproblem, spezielles Rucksackproblem, Erbschaft
- Bin Packing
- Springerproblem, Acht-Damen-Problem
- Travelling Salesman Problem (TSP; Handlungsreisendenproblem)
- Lösungssuche in der Logikprogrammierung (z.B. in PROLOG)
- Mautstraßenproblem, Rekombinationsprobleme
- Schaltkreisoptimierungen
- Stundenpläne, Zuteilungsprobleme

Ende des Moduls Backtracking (Rucksackproblem)