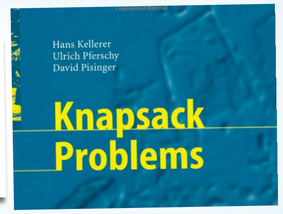


$$\text{maximize} \quad \sum_{j=1}^{n} p_j x_j$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \le c,$$

$$x_j = 0 \text{ or } 1, \qquad j = 1, \ldots, n,$$



Algorithmen und Datenstrukturen 2

Sommer 2026

Prof. Dr. Sándor Fekete

Startseite

Algorithmen und Datenstrukturen 2

Die Vorlesung **Algorithmen und Datenstrukturen 2** ist eine Wahlpflichtveranstaltung für Studierende der Informatik, Wirtschaftsinformatik, Informations- und Systemtechnik; außerdem ist sie offen für interessierte Studierende anderer Studiengänge.

Algorithmen sind das methodische Herz der theoretischen und praktischen Informatik; Datenstrukturen ermöglichen die effiziente Umsetzung von Algorithmen und den effizienten Zugriff auf Input- und Outputdaten. In dieser weiterführenden Vorlesung werden die folgenden

Neuigkeiten

Letzte Veranstaltungen

Vorlesung 12

Übung 6

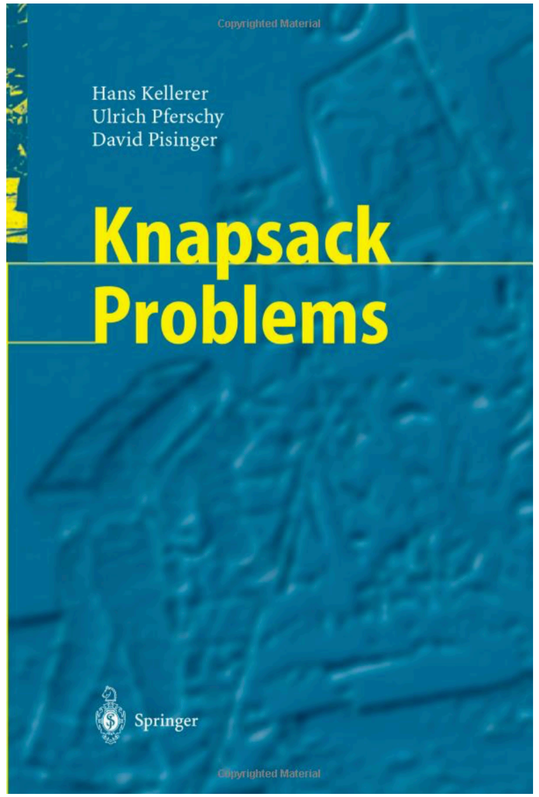
Vorlesung 11

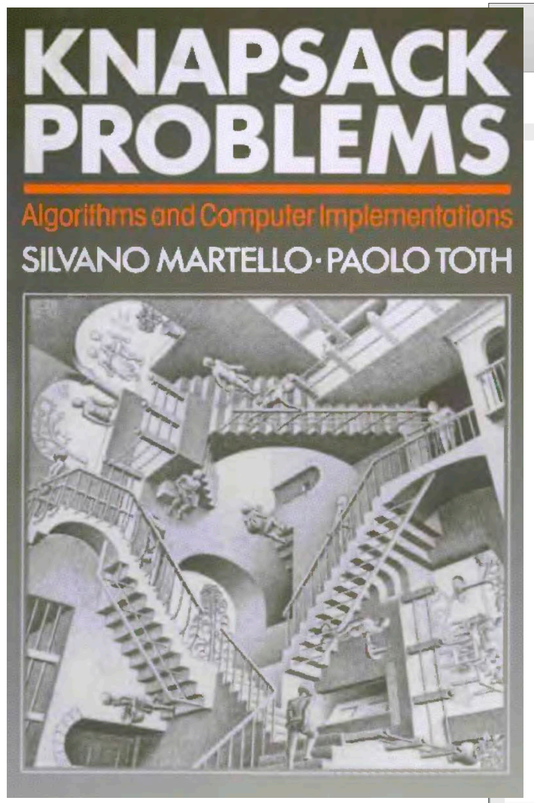
Vorlesung 10

Zeiten

Vorlesung

Literatur





KnapsackProblems.pdf (Seite 5 von 306)

Hervorhebung Drehen Markierungen Suchen Suchen nach

Contents

Preface xi

1 Introduction 1
1.1 What are knapsack problems? 1
1.2 Terminology 2
1.3 Computational complexity 6
1.4 Lower and upper bounds 9

2 0-1 Knapsack problem 13
2.1 Introduction 13
2.2 Relaxations and upper bounds 16
2.2.1 Linear programming relaxation and Dantzig's bound 16
2.2.2 Finding the critical item in $O(n)$ time 17
2.2.3 Lagrangian relaxation 19
2.3 Improved bounds 20
2.3.1 Bounds from additional constraints 20
2.3.2 Bounds from Lagrangian relaxations 23
2.3.3 Bounds from partial enumeration 24
2.4 The greedy algorithm 27

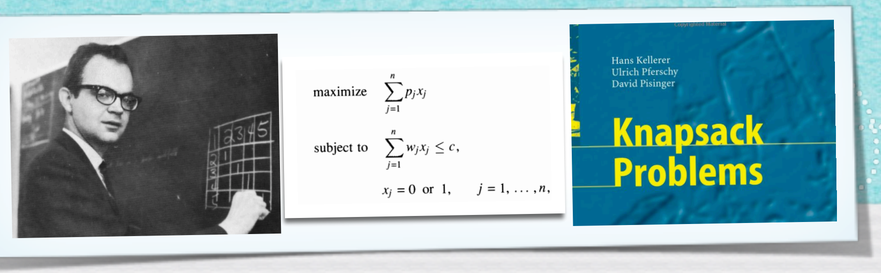
Literatur

Algorithmen und Datenstrukturen 2

Sándor P. Fekete

LaTeX Version: Arne Schmidt

21. Juni 2021



maximize $\sum_{j=1}^{n} p_j x_j$

subject to $\sum_{j=1}^{n} w_j x_j \leq c,$

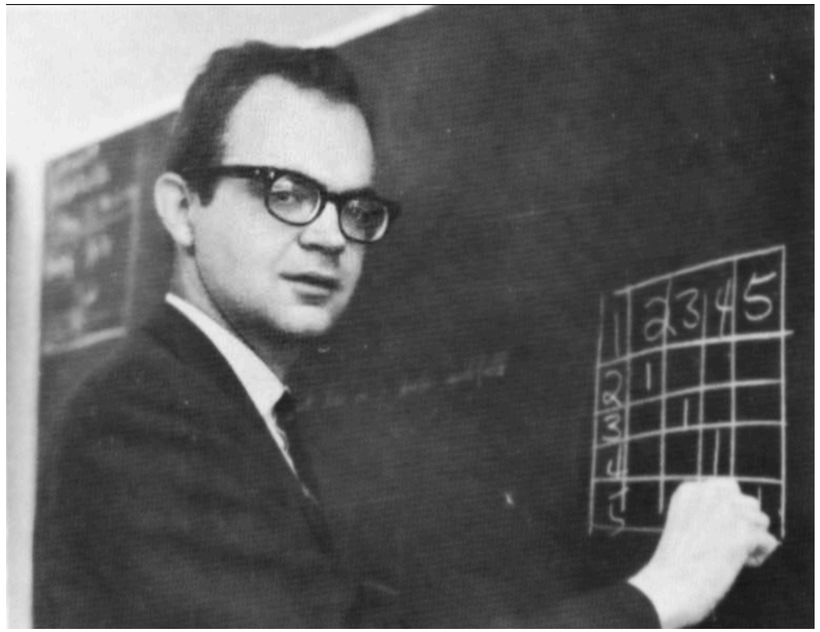
$x_j = 0 \text{ or } 1, \quad j = 1, \ldots, n,$

1 Einführung: Knapsack-Probleme

Algorithmen und Datenstrukturen 2
Sommer 2026

Prof. Dr. Sándor Fekete

Die Zeit läuft!



Knut Donald, erster Studierender der Informatik



- 150 Minuten
- 20 Aufgaben
- 100 Punkte
- 50 Punkte zum Bestehen
- Die Zeit läuft!

30 Minuten später:

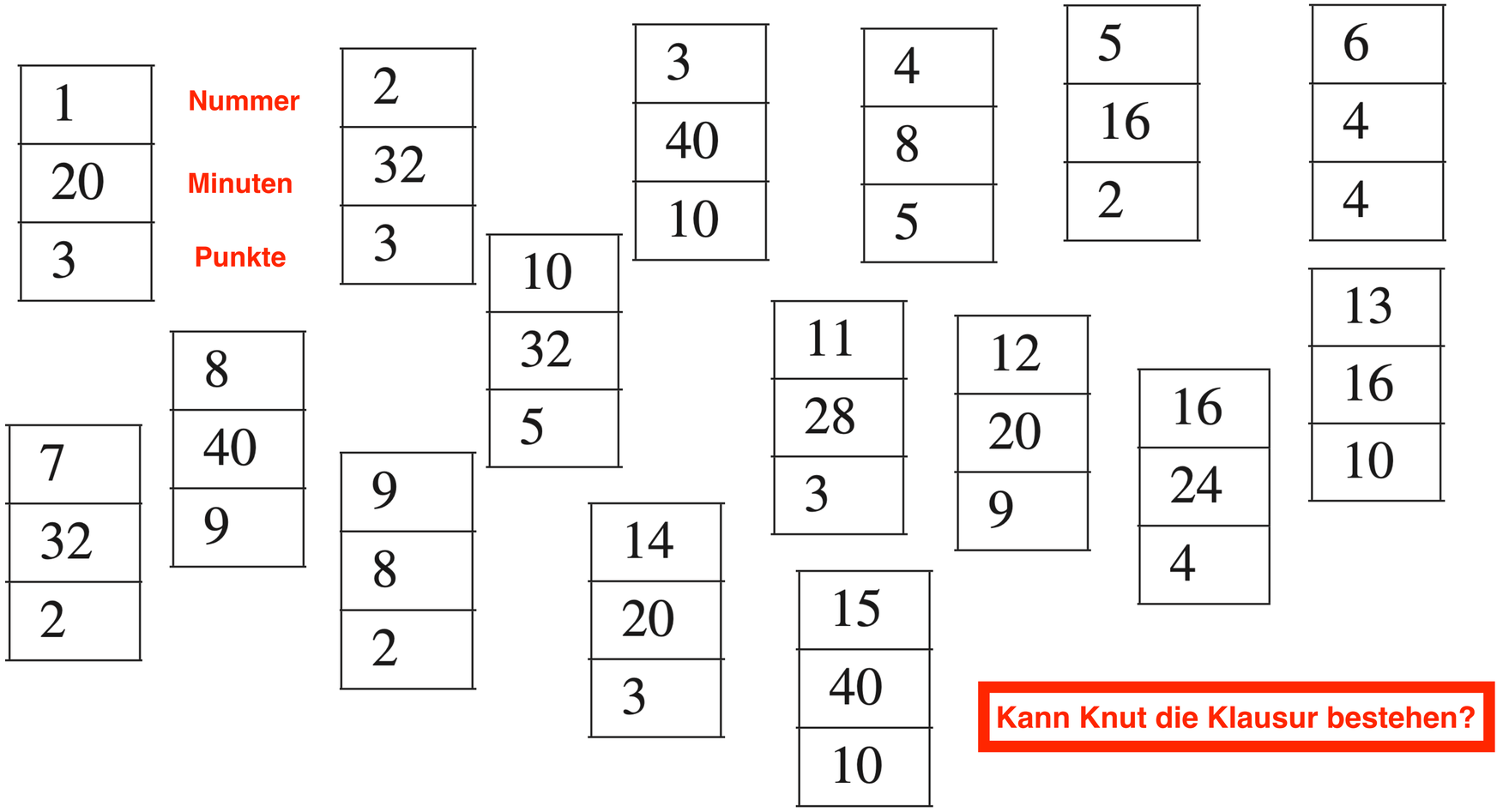






- 6 sichere Punkte
- 10 hoffnungslose Punkte
- Noch 44 Punkte zum Bestehen
- Restliche Aufgaben…

Restliche Aufgaben



Kann Knut die Klausur bestehen?

Sortieren?

nach Wert $\left(\frac{z_i}{p_i}\right)$ sortieren:

i	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
z_i	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
p_i	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

$$\sum_{i=1}^{n} z_i x_i = 120$$

$$\sum_{i=1}^{n} p_i x_i = 44$$

Knut kann die Klausur bestehen!

Optimierungsvariante

Problem 1.2′ (MAXIMUM KNAPSACK).

Gegeben:

- *n Objekte* $1, ..., n$ *mit jeweils Größe* z_i *Gewinn* p_i
- *Größenschranke* Z

Gesucht:

Eine Menge

$$S \subseteq \{1, ..., n\}$$

mit

$$\sum_{i \in S} z_i \leq Z$$

und

$$\sum_{i \in S} p_i = Maximal$$

Teilaufgaben?!

Problem 1.3 (FRACTIONAL KNAPSACK).

Gegeben:

- *n Objekte* $1, ..., n$ *mit jeweils Größe* $z_i > 0$ *Gewinn* $p_i > 0$
- *Größenschranke* Z

Gesucht:

Für jedes Objekt ein Wert

$$x_i \in [0, 1]$$

sodass

$$\sum_{i=1}^{n} z_i x_i \leq Z$$

und

$$\sum_{i=1}^{n} p_i x_i = Maximal$$

Teilaufgaben?!

nach Wert $\left(\frac{z_i}{p_i}\right)$ sortieren:

i	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
z_i	4	8	16	20	8	40	24	40	24	32	20	20	16	28	32	32
p_i	4	5	10	9	2	10	6	9	4	5	3	3	2	3	3	2

$$\sum_{i=1}^{n} z_i x_i = 120$$

$$\sum_{i=1}^{n} p_i x_i = 46$$

$x_{15} = 0{,}6$

Algorithmus

Algorithmus 1.4. *(Greedy-Algorithmus)*

Eingabe: $z_i, ..., z_n, Z, p_i, ..., p_n$

Ausgabe: $x_i, ..., x_n \in [0, 1]$

mit

$$\sum_{i=1}^{n} z_i x_i \leq Z$$

und

$$\sum_{i=1}^{n} p_i x_i = Maximal$$

1: *Sortiere {1,...,n} nach* $\frac{z_i}{p_i}$ *aufsteigend;*
Dies ergibt die Permutation $\pi(1), ..., \pi(n)$.
Setze $j = 1$.

2: ***while*** $\left(\sum_{i=1}^{j} z_{\pi(i)} \leq Z\right)$ ***do***

3: $\quad x_{\pi(j)} := 1$

4: $\quad j := j + 1$

5: *Setze* $x_{\pi(j)} := \dfrac{Z - \sum_{i=1}^{j-1} z_{\pi(i)}}{z_{\pi(j)}}$

6: ***return***

Das klappt immer!

Satz 1.5. *Algorithmus 1.4 liefert eine optimale Lösung für Problem 1.3.*

Beweis:
O.B.d.A. sei $\sum_{i=1}^{n} z_{\pi(i)} > Z$ - sonst nimmt man einfach alles.

(1) Wir bekommen eine zulässige Lösung.
(2) Wir bekommen eine optimale Lösung.

(1) Zulässigkeit

Setze $j = 1$.
2: ***while*** $\left(\sum_{i=1}^{j} z_{\pi(i)} \leq Z\right)$ ***do***
3: $\quad x_{\pi(j)} := 1$
4: $\quad j := j + 1$
5: *Setze* $x_{\pi(j)} := \dfrac{Z - \sum_{i=1}^{j-1} z_{\pi(i)}}{z_{\pi(j)}}$
6: ***return***

i	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
z_i	4	8	16	20	8	40	24	40	24	32	20	20	16	28	32	32
p_i	4	5	10	9	2	10	6	9	4	5	3	3	2	3	3	2

$\sum_{i=1}^{n} z_i x_i$ = 120

$\sum_{i=1}^{n} p_i x_i$ = 46

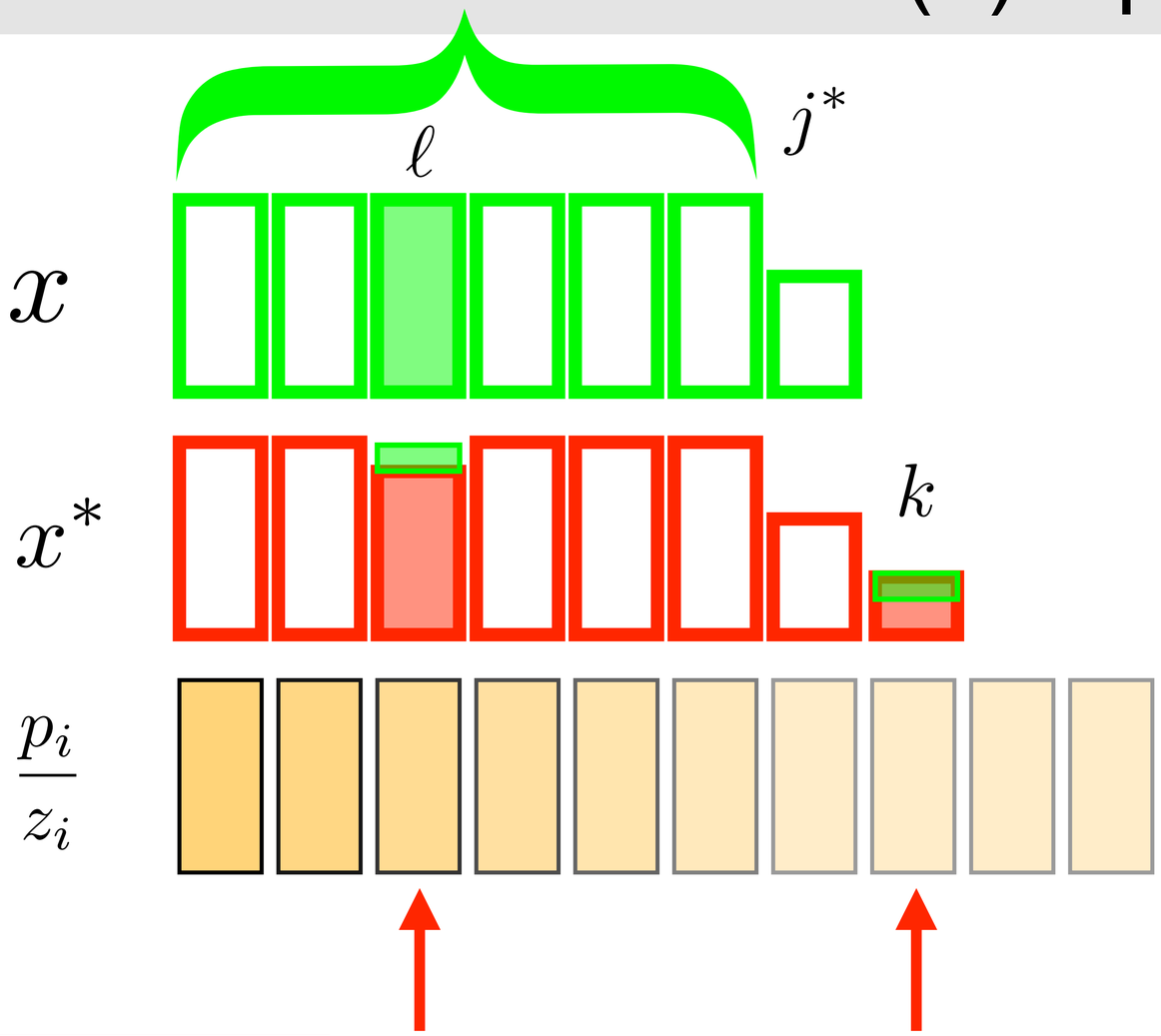
x_{15} = 0,6

Sei j^* der letzte Index, für den die **while**-Bedingung getestet wird.

Dann ist $\sum_{i=1}^{j^*-1} z_{\pi(i)} \leq Z$ und $\sum_{i=1}^{j^*} z_{\pi(i)} > Z$

Also ist $x_{\pi(1)}, \ldots, x_{\pi(j^*-1)} = 1$ und $x_{\pi(j^*)} \in [0, 1[$ und $\sum_{i=1}^{j^*} x_{\pi(i)} z_{\pi(i)} = Z$

(2) Optimalität



$$\sum_{i=1}^{n} x_i p_i < \sum_{i=1}^{n} x_i^* p_i$$

$$p_i > 0 :$$

$$\Longrightarrow \exists\, k \in \{1, \ldots, n\} : \; x_{\pi(k)} < x^*_{\pi(k)}$$

$$\Longrightarrow \exists\, k \in \{j^*, \ldots, n\} : \; x_{\pi(k)} < x^*_{\pi(k)}$$

$$\Longrightarrow \exists\, \ell \in \{1, \ldots, j^*\} : \; x_{\pi(\ell)} > x^*_{\pi(\ell)}$$

x^* nicht optimal!

Verallgemeinerung: Mehrfachverwendung

Problem 1.6 (INTEGER KNAPSACK).

Gegeben:

- *n Objekte* $1, ..., n$ *mit jeweils Größe* z_i *Gewinn* p_i
- *Größenschranke* Z

Gesucht:

Für jedes Objekt ein Wert

$$x_i \in \mathbb{N}$$

mit

$$\sum_{i=1}^{n} z_i x_i \leq Z$$

und

$$\sum_{i=1}^{n} p_i x_i = Maximal$$

Spezialfall: Alle Gewinndichten gleich

Problem 1.7.

Gegeben:

- n Objekte $1, ..., n$ mit jeweils Größe z_i
- Größenschranke Z

Gesucht:

Eine Menge

$$S \subseteq \{1, ..., n\}$$

mit

$$\sum_{i \in S} z_i \leq Z$$

und

$$\sum_{i \in S} z_i = Maximal$$

Spezialfall des Spezialfalls: Größenschranke treffen

Problem 1.8 (SUBSET SUM).

Gegeben:

- *n Objekte* $1, ..., n$ *mit jeweils Größe* z_i
- *Zielgröße* Z

Gesucht:

Eine Menge

$$S \subseteq \{1, ..., n\}$$

mit

$$\sum_{i \in S} z_i = Z$$

Spezialfall^3: Partition

Problem 1.9 (PARTITION).

Gegeben:

- *n Objekte* $1, ..., n$ *mit jeweils Größe* z_i

Gesucht:

Eine Menge

$$S \subseteq \{1, ..., n\}$$

mit

$$\sum_{i \in S} z_i = \sum_{i \notin S} z_i$$

Beispiel

Beispiel 1.10.

Partition für $\{z_1, ..., z_9\} = \{7, 13, 17, 20, 29, 31, 31, 35, 57\}$

Gesamtsumme: 240

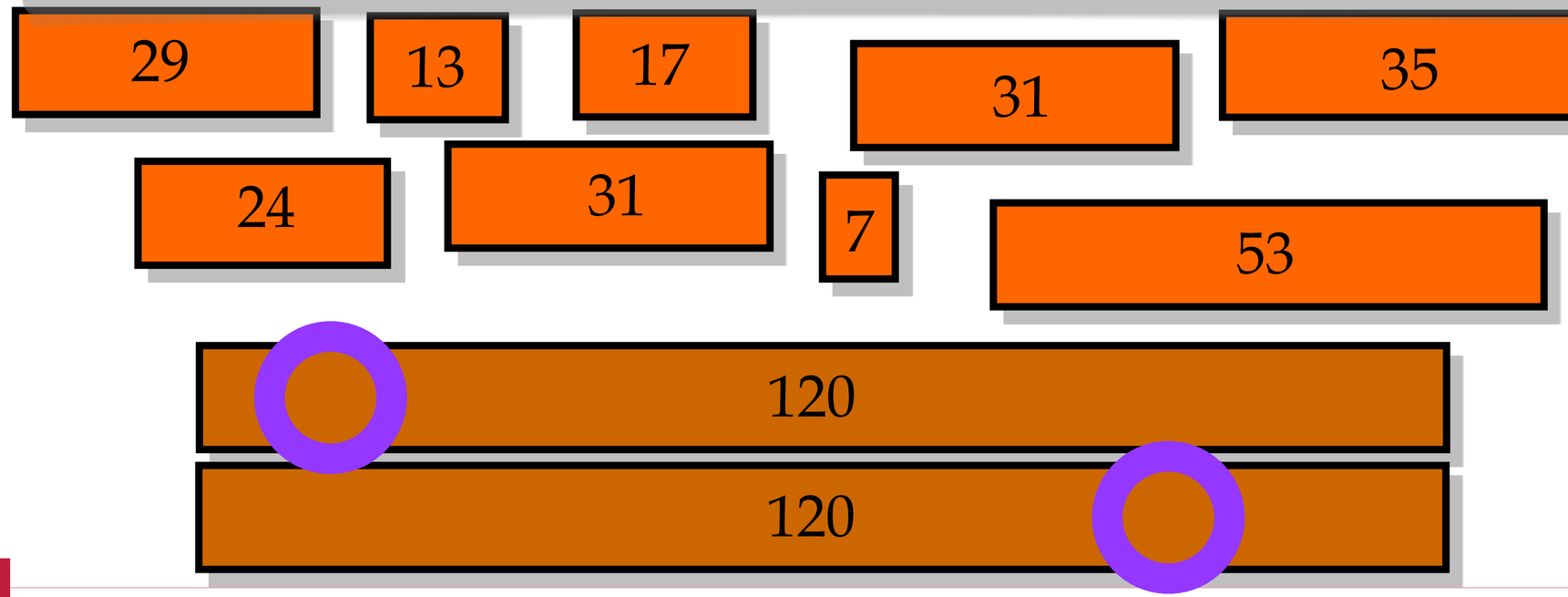
Finde Aufteilung in zwei gleiche Teilmengen!

Ein Bild hilft

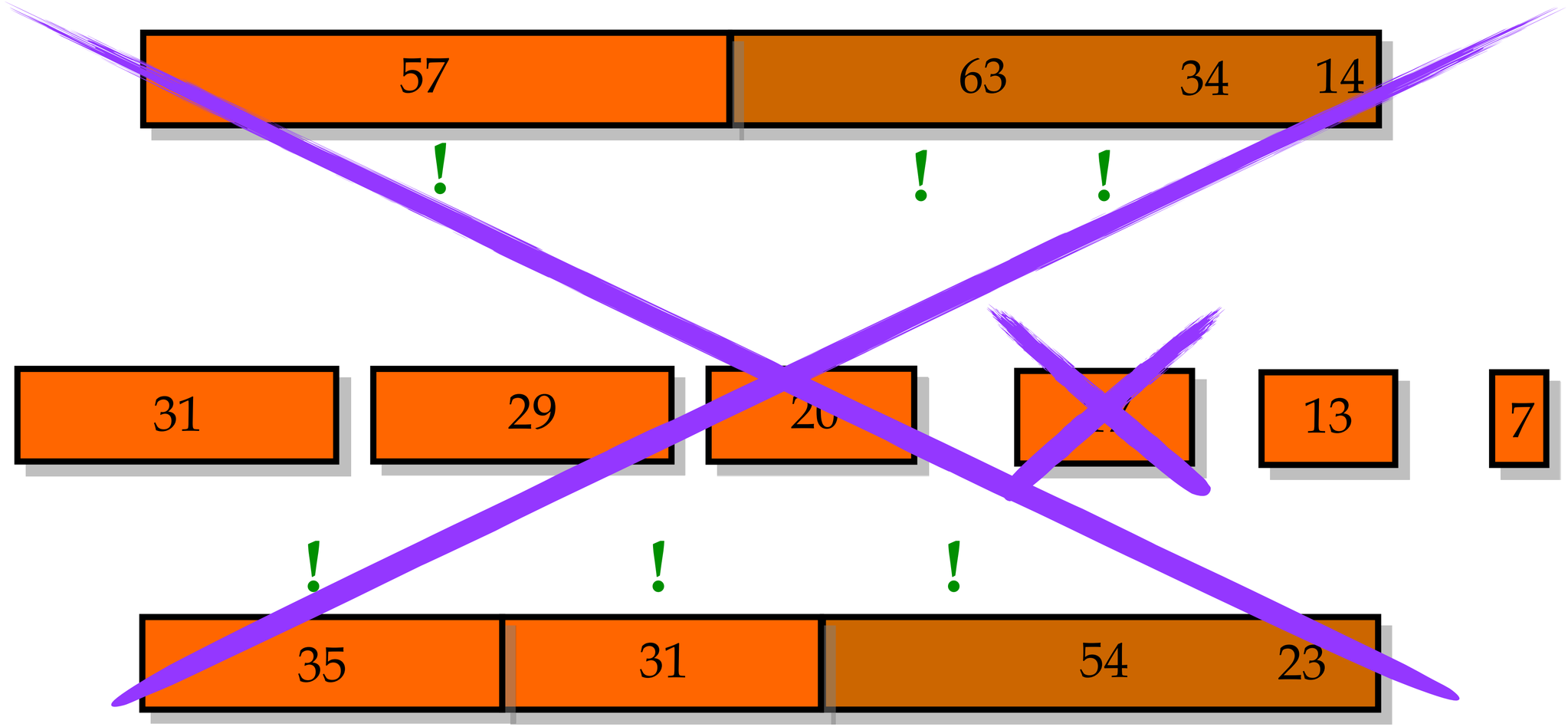
Beispiel 1.10.

Partition für $\{z_1, ..., z_9\} = \{7, 13, 17, 20, 29, 31, 31, 35, 57\}$

Gesamtsumme: 240

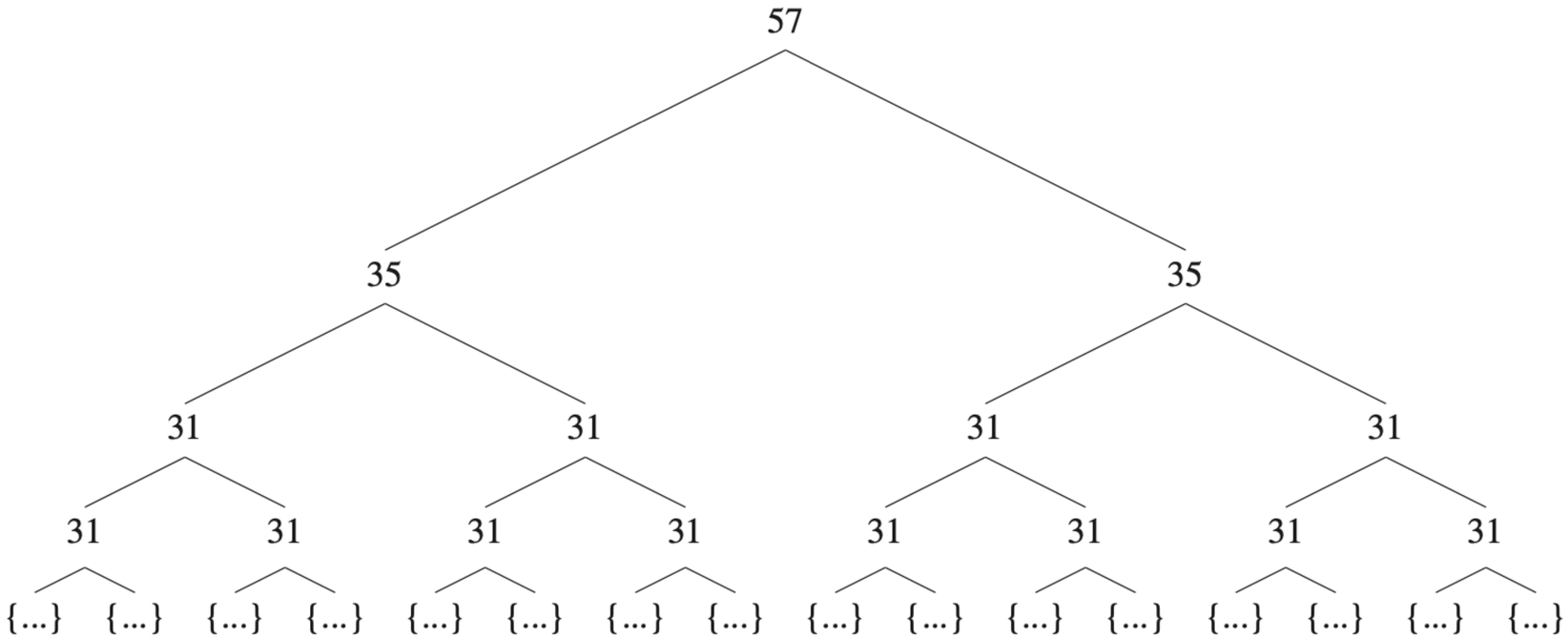


Keine Lösung?!



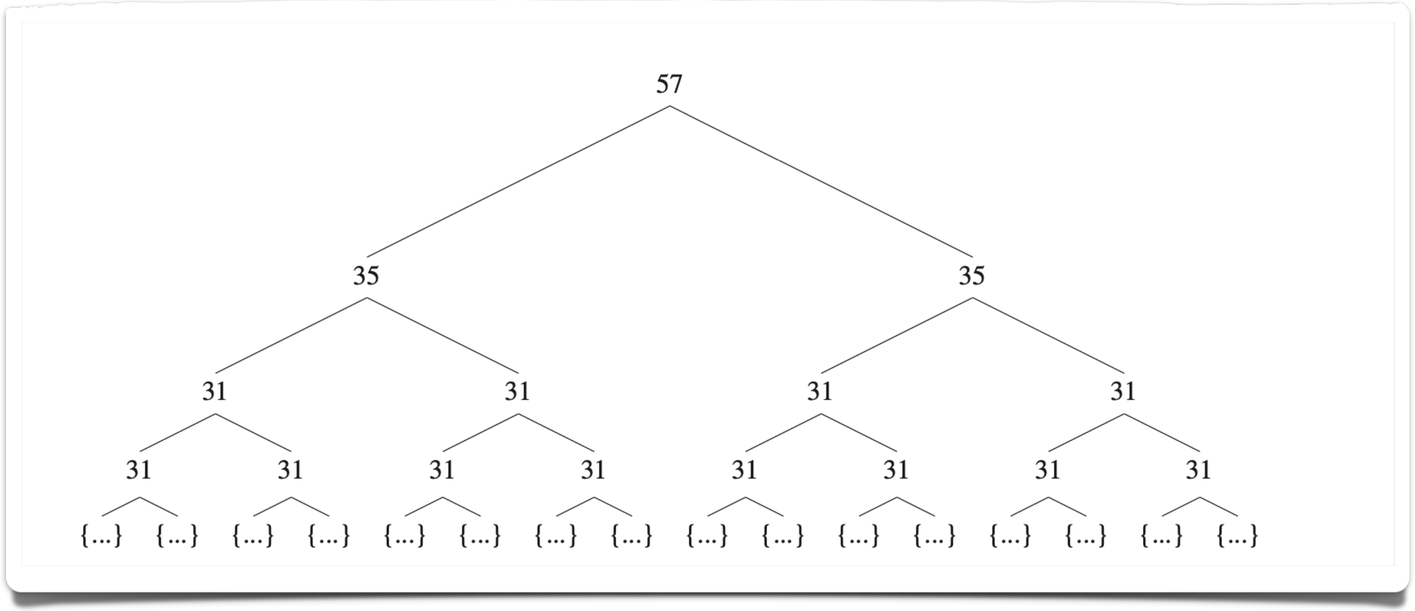
Enumerationsprinzip

120



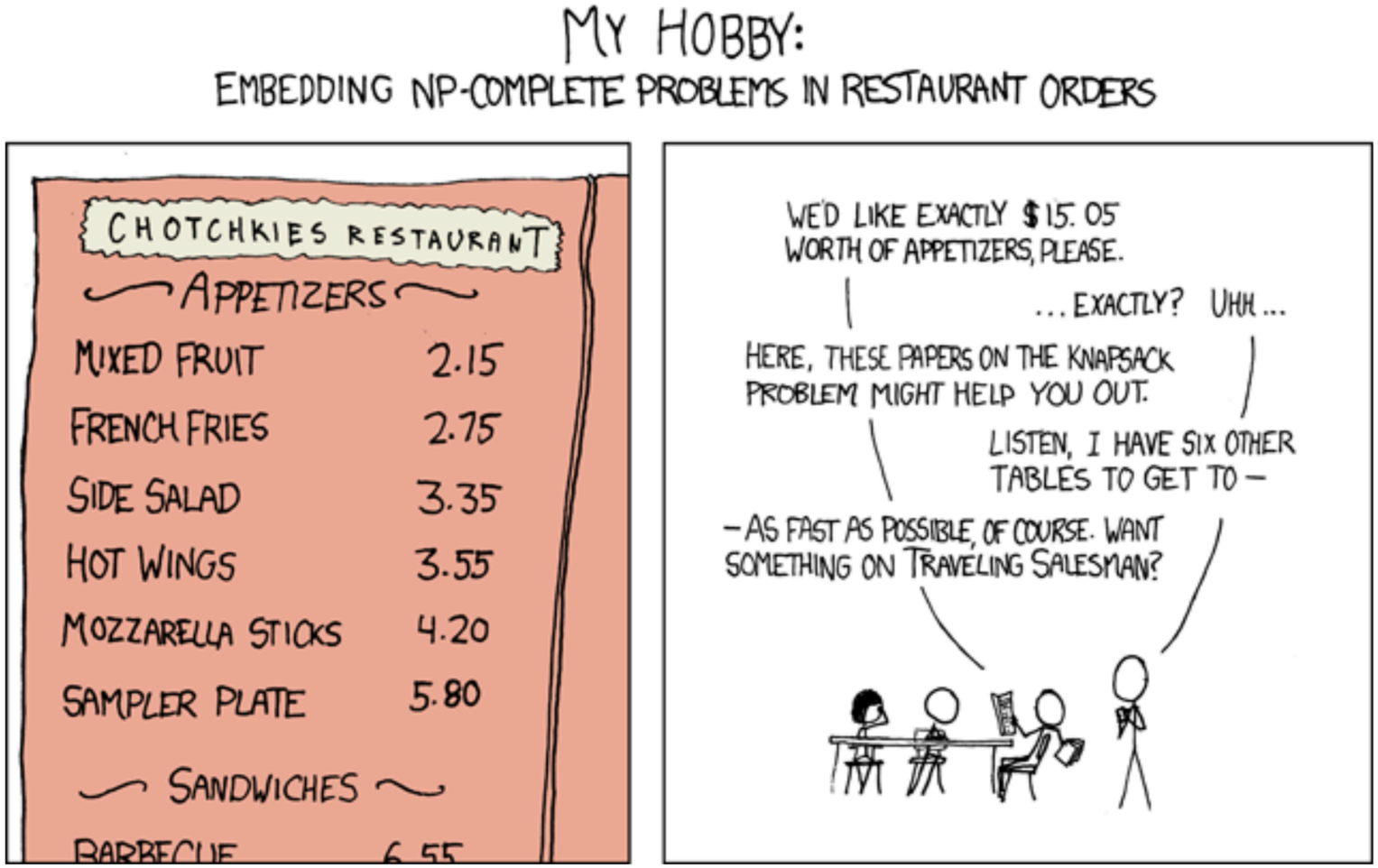
120

Enumerationsprinzip



- Exponentiell viele Fälle!
- Wie geht das systematisch?
- Wo kann man Arbeit sparen?

Alltagsanwendung: xkcd #287





$$P(x,i) = \begin{cases} P(x,i-1), \\ max\{P(x,i- \end{cases}$$

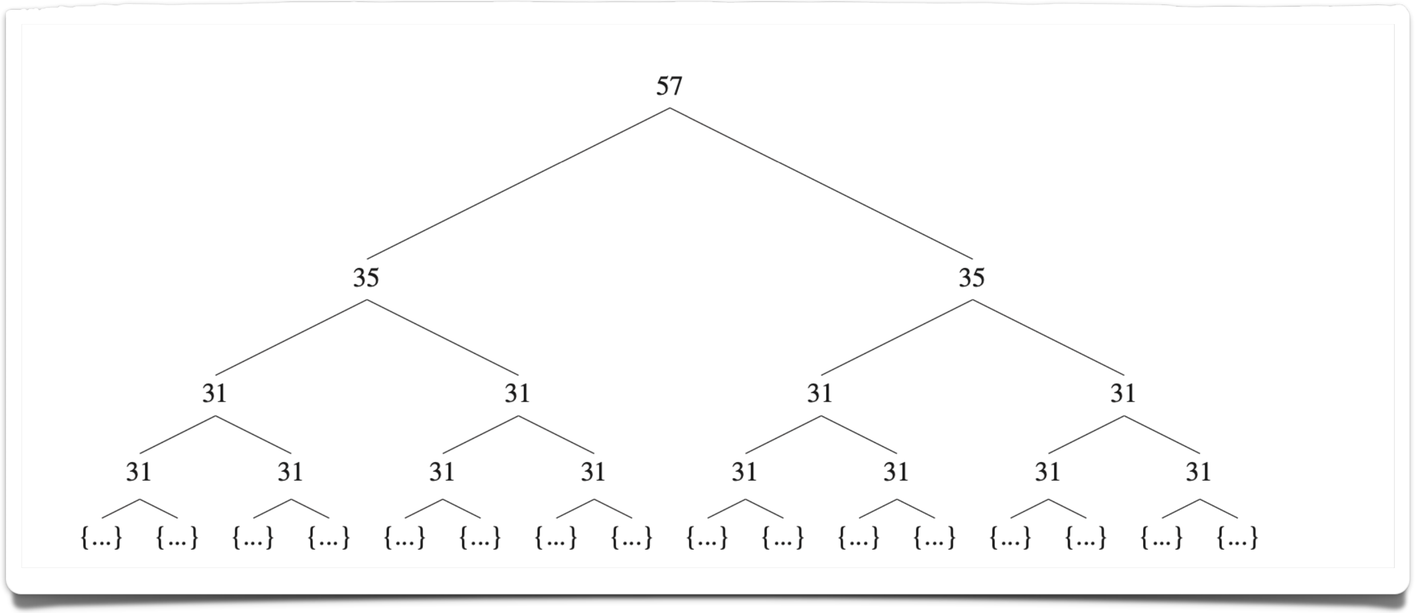
i \ x	0	1	2	3	3	5
0	1	0	0	0	0	0
1	1	0	0	0	0	0
2	1	0	0	0	+13	
3						

2 Dynamic Programming

Algorithmen und Datenstrukturen 2
Sommer 2026

Prof. Dr. Sándor Fekete

Dynamic Programming



- Bestimme erreichbare Zahlen
- Vorwärts: Nimm Zahlen hinzu

- Zeige Unerreichbarkeit von Zahlen
- Rückwärts: Schließe Zahlen aus

- Wieviel ist das?

  1+1+1+1+1+1+1+1+1+1+1+1+1 =13

- Und das?

  1+1+1+1+1+1+1+1+1+1+1+1+1+1 = 13+1 = 14

- Dynamic Programming:
  - Nicht von vorn lösen
  - Zwischenergebnisse merken

Beispiel 2.1



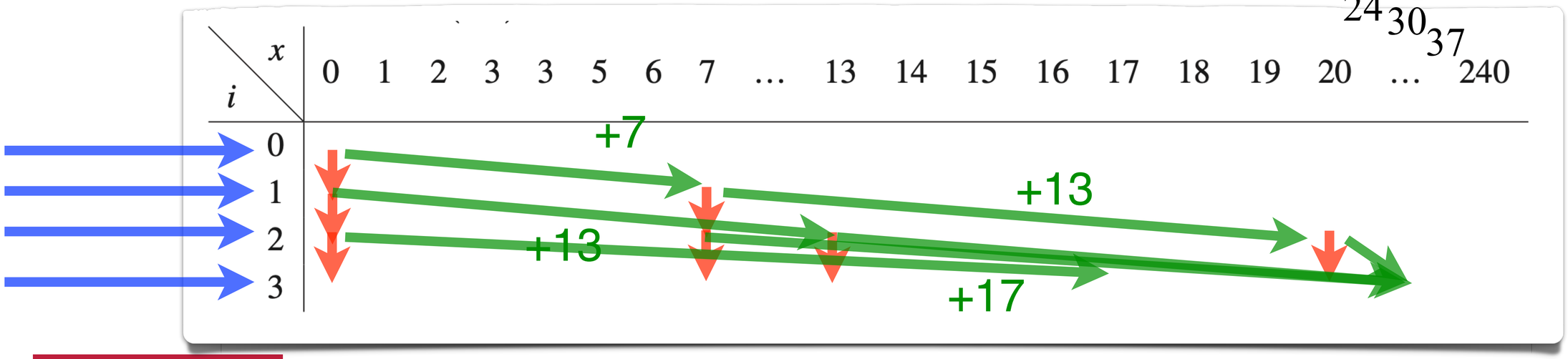
$$\{z_1, \ldots, z_9\} = \{7, 13, 17, 20, 29, 31, 31, 35, 57\}$$

Bestimme alle Zahlen, die als Teilsummen erreichbar sind!

- Bestimme erreichbare Zahlen
- Vorwärts: Nimm Zahlen hinzu

$$\{z_1, \ldots, z_9\} = \{7, 13, 17, 20, 29, 31, 31, 35, 57\}$$

$$\mathcal{S}(x,i) = \begin{cases} 1, & \text{falls x mit } z_1, \ldots, z_i \text{ erzeugt werden kann} \\ 0, & \textit{sonst} \end{cases}$$



$$\mathscr{S}(x,i) = \begin{cases} 1, & \text{falls x mit } z_1, ..., z_i \text{ erzeugt werden kann} \\ 0, & \textit{sonst} \end{cases}$$

$$\mathscr{S}(x,0) = 0, \text{ für alle } x \in \{1, ..., Z\}; \mathscr{S}(0,0) = 1$$

i \ x	0	1	2	3	3	5	6	7	…	13	14	15	16	17	18	19	20	…	240
0	1	0	0	0	0	0	0	0	…	0	0	0	0	0	0	0	0	…	0
1	1	0	0	0	0	0	0	1	…	0	0	0	+13	0	0	0	0	…	0
2	1	0	0	0	+13	0	1	…	1	0	0	0	0	0	0	1	…	0	
3																			

$$\mathscr{S}(x,i-1) = 1 \Rightarrow \mathscr{S}(x,i) = 1$$

$$\mathscr{S}(x - z_i, i-1) = 1 \Rightarrow \mathscr{S}(x,i) = 1$$

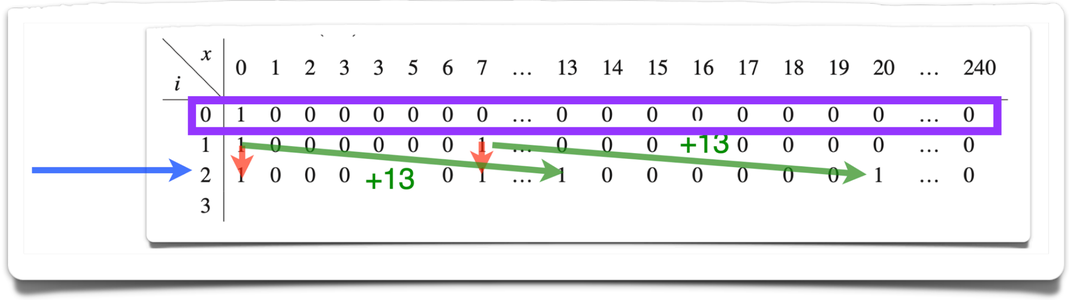
Algorithmus 2.2 . (Dynamic Programming für SUBSET SUM).

Eingabe: *Zahlen* $z_1, ..., z_n$ *mit* $\sum_{i=1}^{n} z_i = Z$

Ausgabe: *Boolesche Funktion* $\mathscr{S} : \{0, ..., Z\} \times \{0, ..., n\} \to \{0, 1\}$

$$\text{mit } \mathscr{S}(x,i) = \begin{cases} 1, & \text{falls } x \text{ mit } z_1, ..., z_i \text{ erzeugt werden kann} \\ 0, & \text{sonst} \end{cases}$$

```
1:  S(0,0) := 1
2:  for (x=1) to Z do
3:      S(x,0) := 0;
4:  for (i = 1) to n do
5:      for (x = 0) to z_i − 1 do
6:          S(x,i) := S(x,i−1);
7:      for (x = z_i) to Z do
8:          if ( S(x,i−1) = 1 OR (S((x−z_i),i−1) = 1) then
9:              S(x,i) := 1;
10:         else
11:             S(x,i) := 0;
12: return
```



Satz 2.3 *Algorithmus 2.2 löst das Problem* SUBSET SUM. *Die Laufzeit ist* $O(Z \cdot n)$.

Beweis. Induktion über die Anzahl der Elemente n.

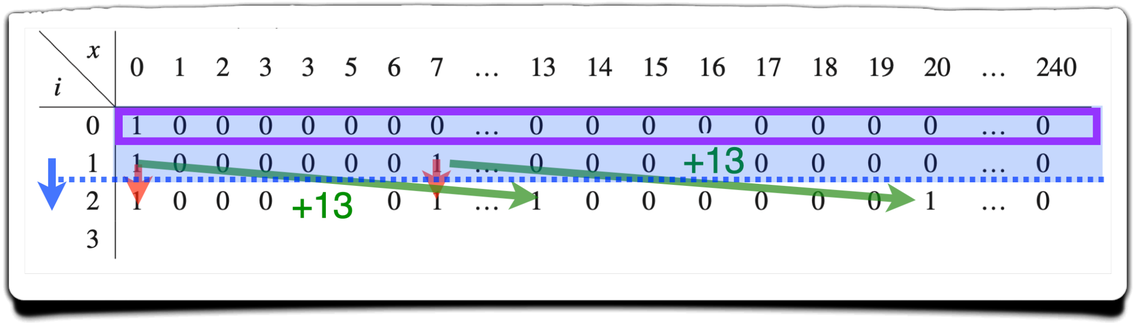
Induktionsanfang: Initialisierung korrekt für $n=0$.

Induktionssannahme: $\mathcal{S}(x, i-1)$ *korrekt für alle i.*

Induktionsschritt: Erzeugung von x mit $z_1, ..., z_i$.

(I) z_i wird nicht verwendet $\Rightarrow$ x kann mit $z_1, ..., z_{i-1}$ erzeugt werden.

(II) z_i wird verwendet $\Rightarrow$ $x - z_i$ kann mit $z_1, ..., z_{i-1}$ erzeugt werden.



Algorithmus 2.2 . (Dynamic Programming für SUBSET SUM).

Eingabe: *Zahlen* $z_1, ..., z_n$ *mit* $\sum_{i=1}^{n} z_i = Z$

Ausgabe: *Boolesche Funktion* $\mathscr{S} : \{0, ..., Z\} \times \{0, ..., n\} \to \{0, 1\}$

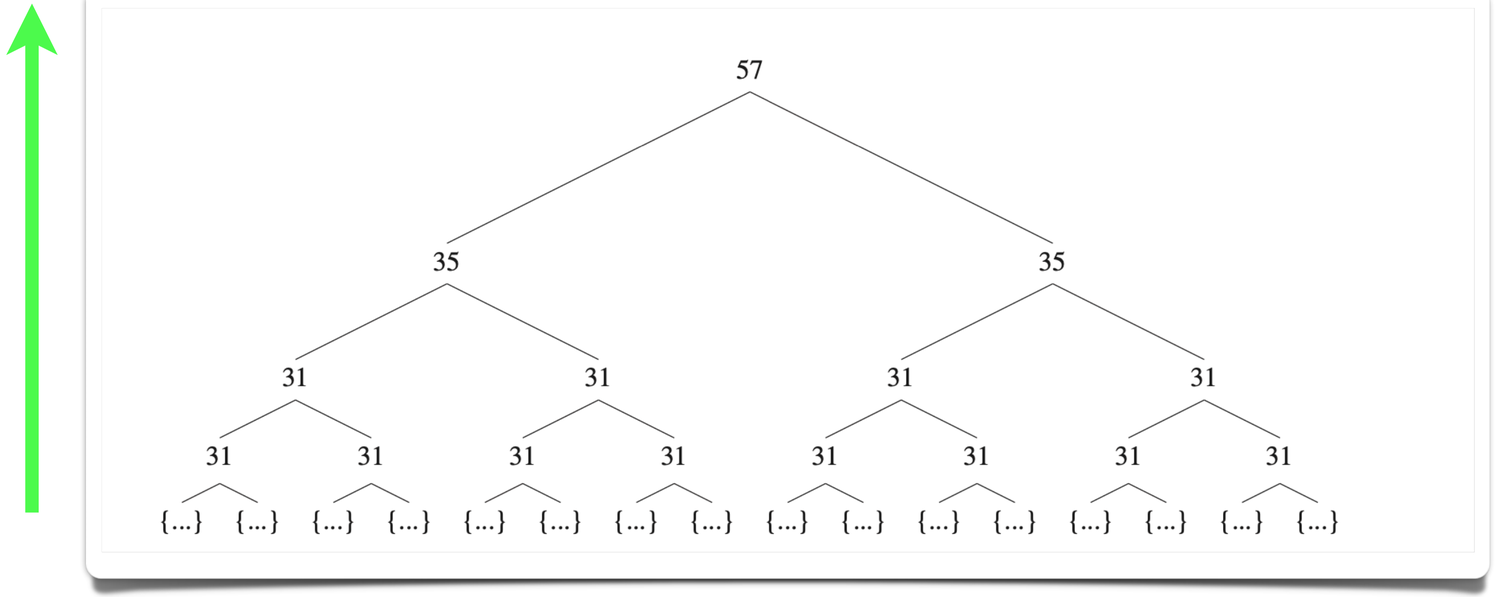
$$\text{mit } \mathscr{S}(x,i) = \begin{cases} 1, & \text{falls } x \text{ mit } z_1, ..., z_i \text{ erzeugt werden kann} \\ 0, & \text{sonst} \end{cases}$$

```
1:  S(0,0) := 1
2:  for (x=1) to Z do
3:      S(x,0) := 0;
4:  for (i = 1) to n do
5:      for (x = 0) to z_i − 1 do
6:          S(x,i) := S(x,i−1);
7:      for (x = z_i) to Z do
8:          if ((S(x,i−1) = 1) OR (S((x−z_i),i−1) = 1)) then
9:              S(x,i) := 1;
10:         else
11:             S(x,i) := 0;
12: return
```

$O(nZ)$

Dynamic Programming

$O(nZ)$



2^n

- Exponentiell viele Fälle!
- Wo sparen wir Arbeit?

Z klein im Vergleich zu 2^n : viele Duplikate gespart

Z groß im Vergleich zu 2^n : viel unnötige Arbeit

- Wie lässt sich das Prinzip verallgemeinern?
- Wie geht das für Knapsack?
- Wie geht das für andere Probleme?



$$P(x,i) = \begin{cases} P(x,i-1), \\ max\{P(x,i- \end{cases}$$

i \ x	0	1	2	3	3	5
0	1	0	0	0	0	0
1	1	0	0	0	0	0
2	1	0	0	0	+13	
3						

2.2 Dynamic Programming für Knapsack

Algorithmen und Datenstrukturen 2
Sommer 2026

Prof. Dr. Sándor Fekete

- Wie lässt sich das Prinzip verallgemeinern?
- Wie geht das für Knapsack?
- Wie geht das für andere Probleme?

- Objekte $1, \ldots, n$
- Kosten $z_1, \ldots, z_n$
- Nutzen $p_1, \ldots, p_n$

$$\mathcal{S}(x,i) = \begin{cases} 1, & \text{falls x mit } z_1, ..., z_i \text{ erzeugt werden kann} \\ 0, & \textit{sonst} \end{cases}$$

i \ x	0	1	2	3	3	5	6	7	…	13	14	15	16	17	18	19	20	…	240
0	1	0	0	0	0	0	0	0	…	0	0	0	0	0	0	0	0	…	0
1	1	0	0	0	0	0	0	1	…	0	0	0	0	0	0	0	0	…	0
2	1	0	0	0	0	0	0	1	…	1	0	0	0	0	0	0	1	…	0
3	1	0	0	0	0	0	0	1	…	1	0	0	0	1	0	0	1	…	0

$P(x, i)$

der höchste Nutzen

der sich mit den Objekten

$1, \ldots, i$ erzielen lässt

wenn die Kosten auf x begrenzt sind

Beispiel 2.4 (Knapsackproblem).

Sei $Z = 9$ und seien folgende sieben Objekte gegeben:

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

$P(x, i)$

der höchste Nutzen

der sich mit den Objekten $1, \ldots, i$ erzielen lässt

wenn die Kosten auf x begrenzt sind

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

1. Objekt i ist zu teuer für Kostenschranke x, falls $z_i > x$

2. Objekt i ist nicht zu teuer, bringt aber keine Verbesserung

3. Objekt i ist nicht zu teuer und bringt eine Verbesserung

i \ x	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	12	14	15
6	0	0	6	6	6	11	11	12	14	15
7	0	0	6	6	6	11	11	12	14	15

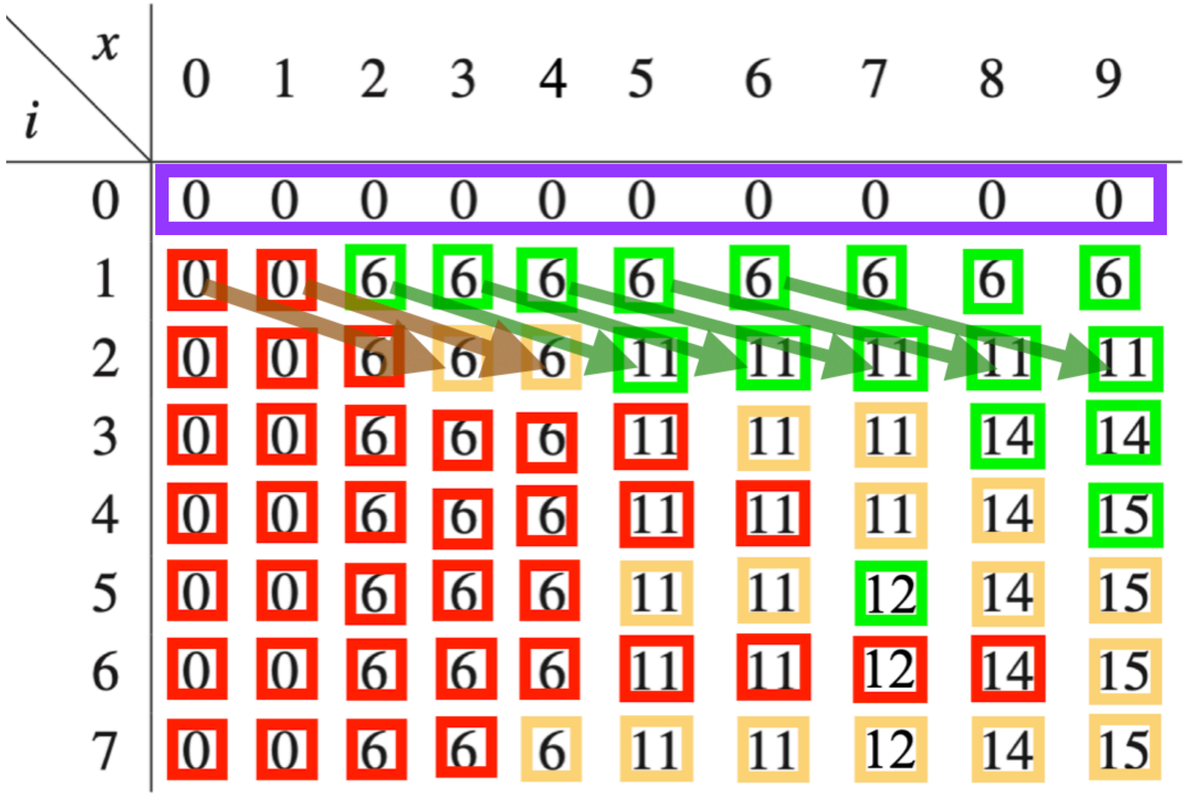
„Bellman-Rekursion“

$$P(x,i) = \begin{cases} P(x,i-1), & \text{falls } z_i > x \\ \max\{P(x,i-1),\ P(x-z_i,i-1)+p_i\}, & \text{sonst} \end{cases}$$

1. Objekt i ist zu teuer für Kostenschranke x, falls $z_i > x$

2. Objekt i ist nicht zu teuer, bringt aber keine Verbesserung

3. Objekt i ist nicht zu teuer und bringt eine Verbesserung



i \ x	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	12	14	15
6	0	0	6	6	6	11	11	12	14	15
7	0	0	6	6	6	11	11	12	14	15



Richard Bellman (1920-1984)

DYNAMIC
PROGRAMMING

BY

RICHARD BELLMAN

1957

PRINCETON UNIVERSITY PRESS
PRINCETON, NEW JERSEY

Preface

The purpose of this work is to provide an introduction to the mathematical theory of multi-stage decision processes. Since these constitute a somewhat formidable set of terms we have coined the term "dynamic programming" to describe the subject matter. Actually, as we shall see, the distinction involves more than nomenclature. Rather, it involves a certain conceptual framework which furnishes us a new and versatile mathematical tool for the treatment of many novel and interesting problems both in this new discipline and in various parts of classical analysis. Before expanding upon this theme, let us present a brief discussion of what is meant by a multi-stage decision process.

The functional equations which arise in this way are of a novel type, completely different from any of the functional equations encountered in classical analysis. The particular one we shall employ for purposes of discussion in this chapter is

$$(1) \qquad f(x) = \operatorname*{Max}_{0 \le y \le x} [g(y) + h(x-y) + f(ay + b(x-y))].$$

where g and h are known functions and a and b are known constants, satisfying the condition $0 \le a, b < 1$.

Algorithmus 2.5 *(Dynamic Programming für Knapsack)*

Eingabe: $z_1, ..., z_n, Z, p_1, ...p_n$

Ausgabe: *Funktion* $P : \{1, ..., Z\} \times \{1, ..., n\} \to \mathbb{R};\ (x, i) \mapsto P(x, i)$

$$\text{mit } P(x,i) = max \sum_{i=1}^{i} p_j y_j \text{ mit } \sum_{j=1}^{i} z_j y_j \leq x, \text{ für } y_j \in \{0,1\}$$

```
1: for (x = 0) to Z do
2:     P(x,0) := 0
3: for (i = 1) to n do
4:     for (x = 0) to (z_i − 1) do
5:         P(x,i) := P(x,i−1)
6:     for (x = z_i) to Z do
7:         if ((P(x − z_i, i−1) + p_i) > P(x,i−1)) then
8:             P(x,i) := P(x − z_i, i−1) + p_i
9:         else
0:             P(x,i) := P(x,i−1)
```

i \ x	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	12	14	15
6	0	0	6	6	6	11	11	12	14	15
7	0	0	6	6	6	11	11	12	14	15

Satz 2.6 *Algorithmus 2.5 berechnet besten Lösungswert für das Rucksackproblem in einer Laufzeit $O(nZ)$.*

Algorithmus 2.5 *(Dynamic Programming für Knapsack)*

Eingabe: $z_1, ..., z_n, Z, p_1, ...p_n$

Ausgabe: Funktion $P : \{1, ..., Z\} \times \{1, ..., n\} \to \mathbb{R};\ (x, i) \mapsto P(x, i)$

$$\text{mit } P(x,i) = max \sum_{i=1}^{i} p_j y_j \text{ mit } \sum_{j=1}^{i} z_j y_j \leq x, \text{ für } y_j \in \{0,1\}$$

```
1: for (x = 0) to Z do
2:     P(x, 0) := 0
3: for (i = 1) to n do
4:     for (x = 0) to (z_i − 1) do
5:         P(x, i) := P(x, i − 1)
6:     for (x = z_i) to Z do
7:         if ((P(x − z_i, i − 1) + p_i) > P(x, i − 1)) then
8:             P(x, i) := P(x − z_i, i − 1) + p_i
9:         else
0:             P(x, i) := P(x, i − 1)
```

$O(nZ)$

i \ x	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	12	14	15
6	0	0	6	6	6	11	11	12	14	15
7	0	0	6	6	6	11	11	12	14	15

Induktion!

1. Objekt i ist zu teuer für Kostenschranke x, falls $z_i > x$
2. Objekt i ist nicht zu teuer, bringt aber keine Verbesserung
3. Objekt i ist nicht zu teuer und bringt eine Verbesserung

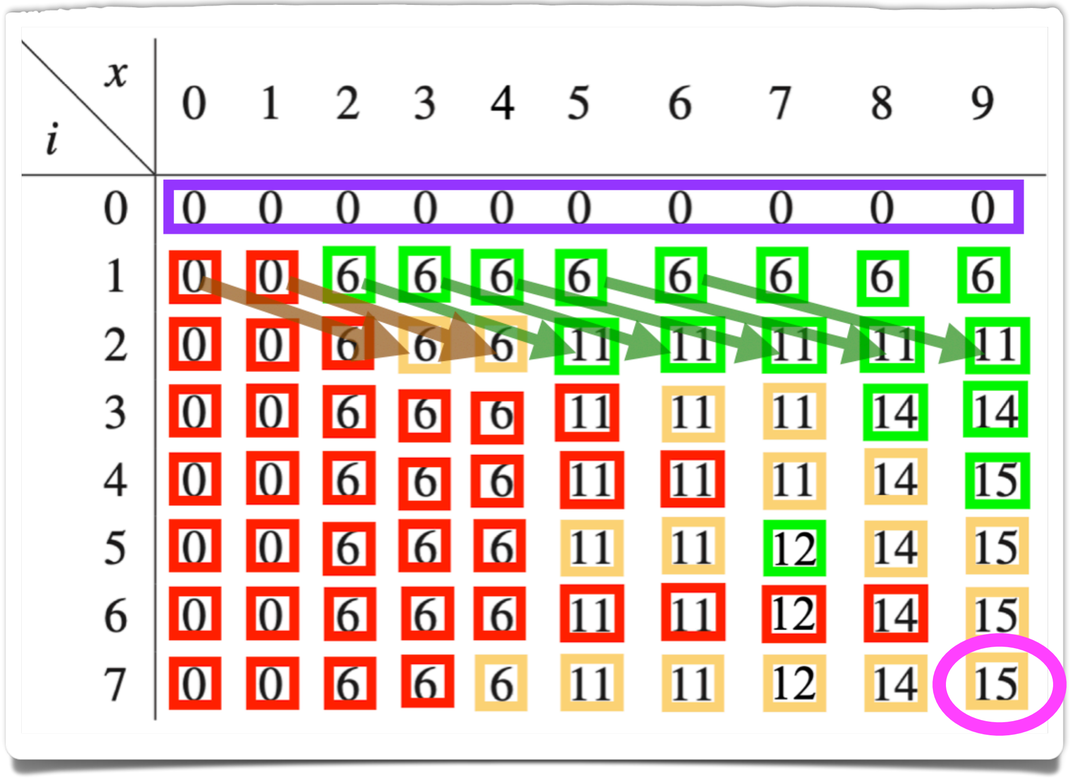
Limitierender Faktor bei Dynamic Programming?

- Laufzeit?
- Speicherplatz!

$O(nZ)$

Idee:

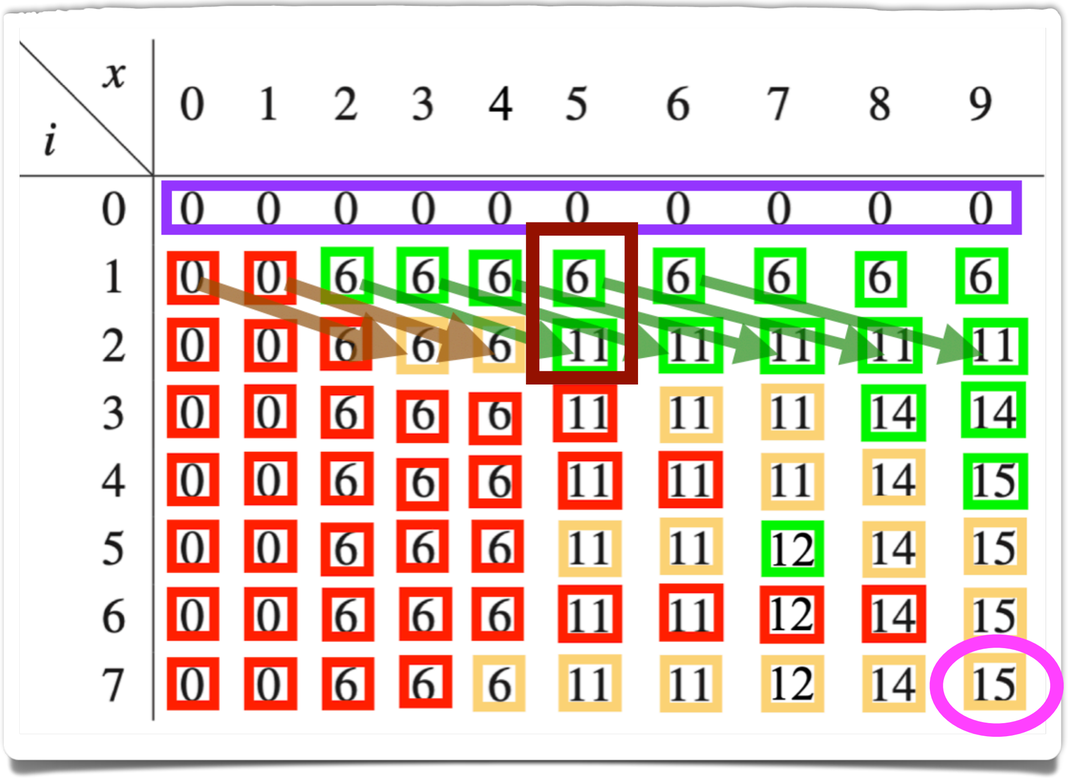
- Nicht ganze Tabelle speichern
- Nur die relevante Zeile



i \ x	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	12	14	15
6	0	0	6	6	6	11	11	12	14	15
7	0	0	6	6	6	11	11	12	14	15

Ideen:

- Nur die jeweils letzte Zeile wird benötigt
- Updates von rechts nach links, damit nur erledigte Werte überschrieben werden



i \ x	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	12	14	15
6	0	0	6	6	6	11	11	12	14	15
7	0	0	6	6	6	11	11	12	14	15

Algorithmus 2.7 *(Dynamic Programming für Knapsack - sparsamer)*

Eingabe: $z_1, ..., z_n, Z, p_1, ...p_n$

Ausgabe: *Funktion* $P : \{1, ..., Z\} \times \{1, ..., n\} \to \mathbb{R};\ (x, i) \mapsto P(x, i)$

$$\text{mit } P(x,i) = max \sum_{j=1}^{i} p_j y_j \text{ mit } \sum_{j=1}^{i} z_j y_j \leq x, \text{ für } y_j \in \{0,1\}$$

```
1: for (x = 0) to Z do
2:     P(x, 0) := 0
3: for (i = 1) to n do
4:     for (x = Z) downto z_i do
5:         if ((P(x − z_i) + p_i) > P(x)) then
6:             P(x, i) := P(x − z_i) + p_i
7: return p* := P(Z)
```

Satz 2.8 *Algorithmus 2.7 berechnet besten Lösungswert für das Rucksackproblem in einer Laufzeit O(nZ).*

Algorithmus 2.5 *(Dynamic Programming für Knapsack)*

Eingabe: $z_1, ..., z_n, Z, p_1, ...p_n$

Ausgabe: *Funktion* $P : \{1, ..., Z\} \times \{1, ..., n\} \to \mathbb{R};\ (x, i) \mapsto P(x, i)$

$$\text{mit } P(x,i) = max \sum_{i=1}^{i} p_j y_j \text{ mit } \sum_{j=1}^{i} z_j y_j \leq x, \text{ für } y_j \in \{0,1\}$$

```
1:  for (x = 0) to Z do
2:      P(x,0) := 0
3:  for (i = 1) to n do
4:      for (x = 0) to (z_i − 1) do
5:          P(x,i) := P(x,i−1)
6:      for (x = z_i) to Z do
7:          if ((P(x − z_i, i−1) + p_i) > P(x,i−1)) then
8:              P(x,i) := P(x − z_i, i−1) + p_i
9:          else
10:             P(x,i) := P(x,i−1)
```

$O(nZ)$

Algorithmus 2.7 *(Dynamic Programming für Knapsack - sparsamer)*

Eingabe: $z_1, ..., z_n, Z, p_1, ...p_n$

Ausgabe: *Funktion* $P : \{1, ..., Z\} \times \{1, ..., n\} \to \mathbb{R};\ (x, i) \mapsto P(x, i)$

$$\text{mit } P(x,i) = max \sum_{i=1}^{i} p_j y_j \text{ mit } \sum_{j=1}^{i} z_j y_j \leq x, \text{ für } y_j \in \{0,1\}$$

```
1:  for (x = 0) to Z do
2:      P(x,0) := 0
3:  for (i = 1) to n do
4:      for (x = Z) downto z_i do
5:          if ((P(x − z_i) + p_i) > P(x)) then
6:              P(x,i) := P(x − z_i) + p_i
7:  return p* := P(Z)
```



$$P(x,i) = \begin{cases} P(x,i-1), \\ max\{P(x,i- \end{cases}$$

i \ x	0	1	2	3	3	5
0	1	0	0	0	0	0
1	1	0	0	0	0	0
2	1	0	0	0	+13	
3						

2.3 Andere Beispiele für Dynamic Programming

Algorithmen und Datenstrukturen 2
Sommer 2026

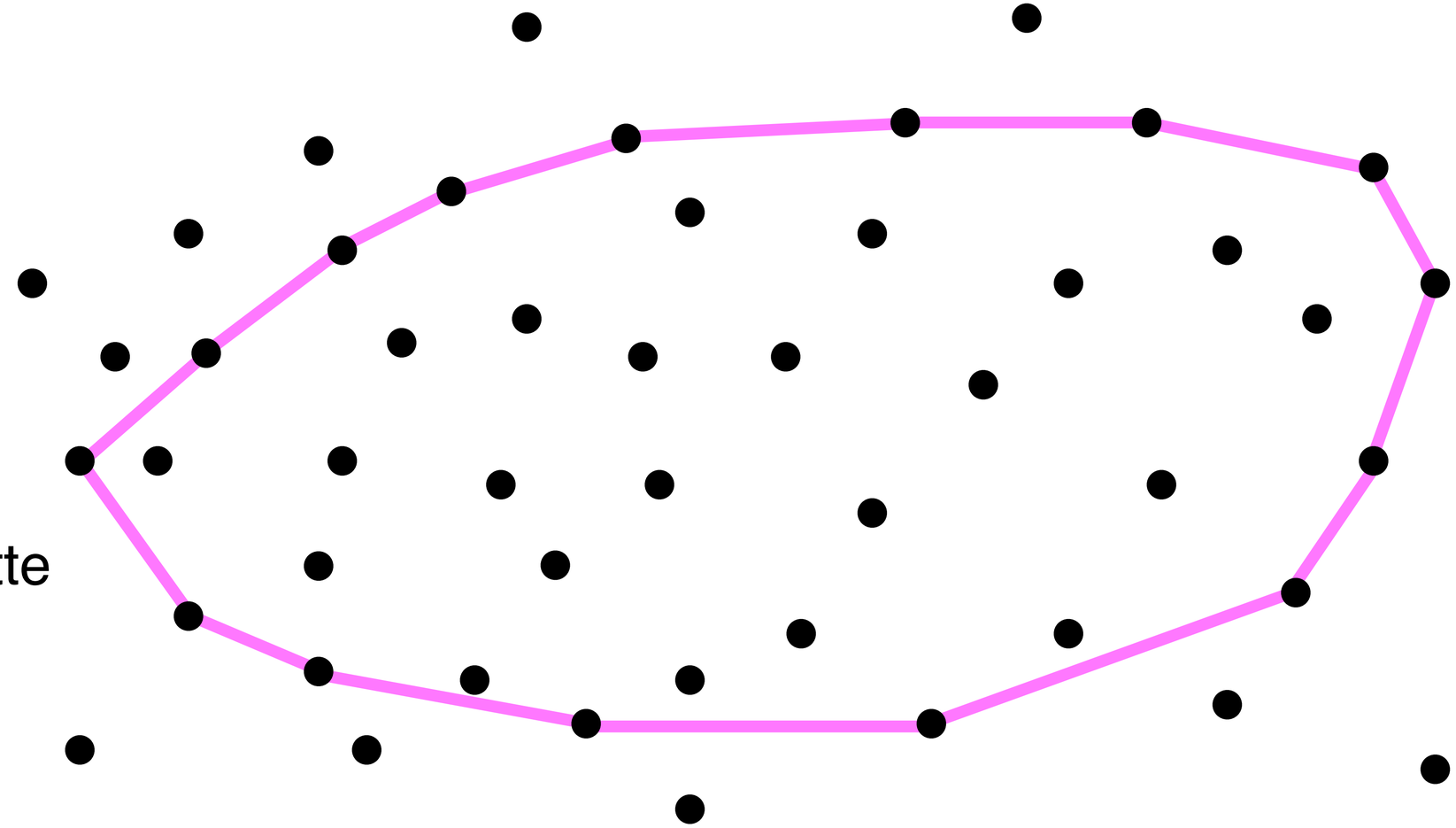
Prof. Dr. Sándor Fekete

Gegeben:

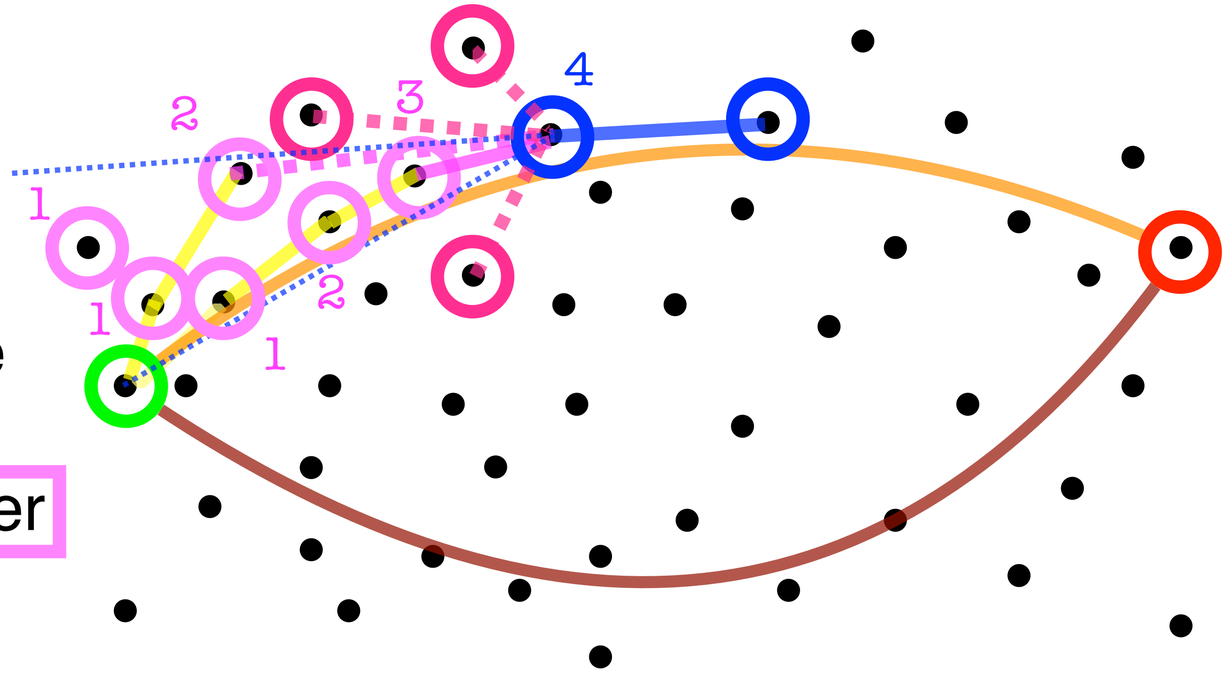
Punktmenge

Gesucht:

Längste konvexe Kette



1. Enumeriere Startpunkt
    1.1 Enumeriere Endpunkt
        1.1.1.Betrachte obere und untere Kette
            1.1.1.1 Für jedes Punktepaar:
                1.1.1.1.1.Ermittle besten Vorgänger

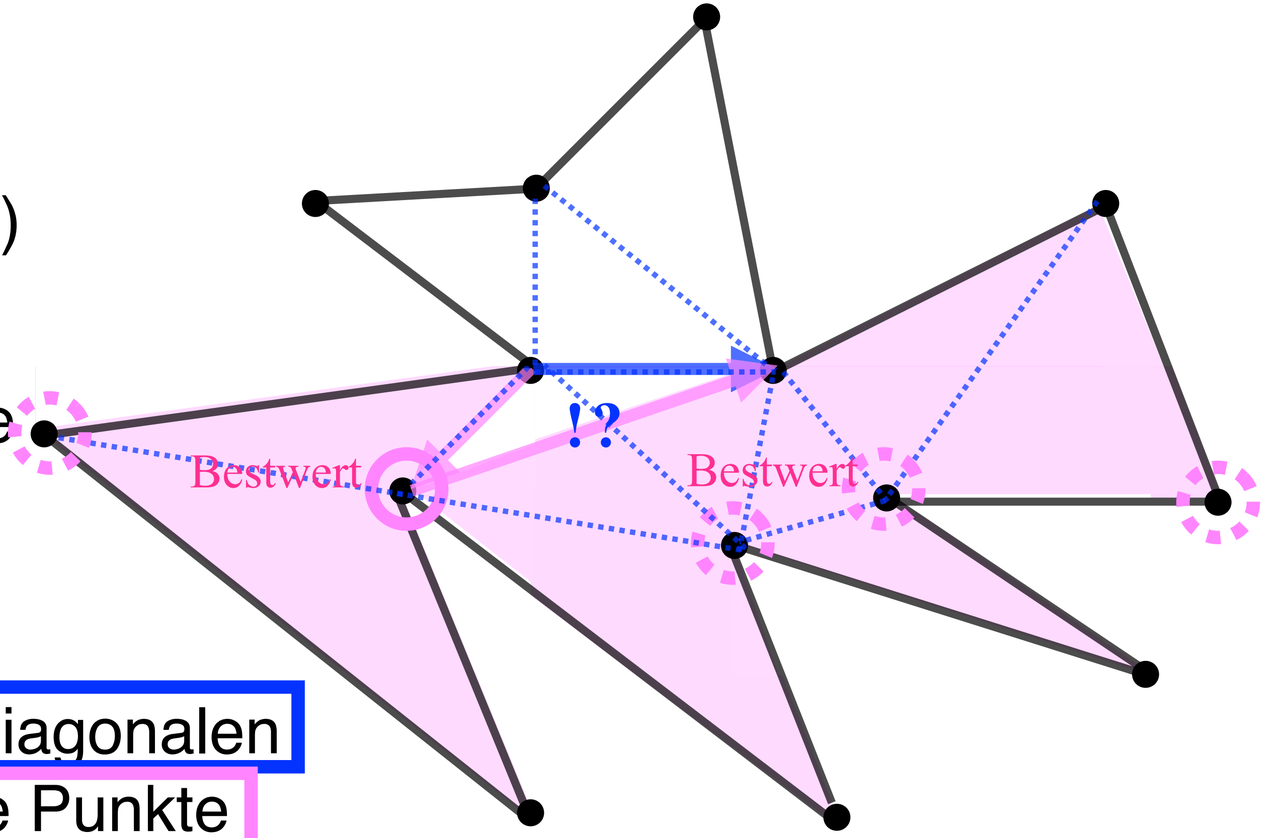


Gegeben:

„Einfaches“ Polygon
(mit zusammenhängendem Rand)

Gesucht:

Optimale Unterteilung in Dreiecke
durch „Diagonalen“



Idee:

1.Berechne für alle (gerichtete) Diagonalen

1.1.Optimiere über möglich dritte Punkte

On the Chromatic Art Gallery Problem

Sándor P. Fekete* Stephan Friedrichs* Michael Hemmer* Joseph B. M. Mitchell†
Christiane Schmidt*

Abstract

For a polygonal region P with n vertices, a *guard cover* S is a set of points in P, such that any point in P can be seen from a point in S. In a *colored guard cover*, every element in a guard cover is assigned a color, such that no two guards with the same color have overlapping visibility regions. The Chromatic Art Gallery Problem (CAGP) asks for the minimum number of colors for which a colored guard cover exists.

We discuss the CAGP for the case of only two colors. We show that it is already *NP*-hard to decide whether two colors suffice for covering a polygon with holes, even when arbitrary guard positions are allowed. For simple polygons with a discrete set of possible guard locations, we give a polynomial-time algorithm for deciding whether a two-colorable guard set exists. This algo-

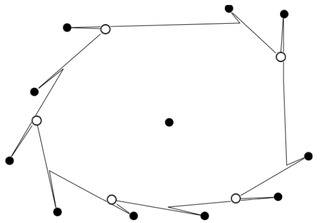


Figure 1: An example polygon with $n = 20$ vertices. A minimum-cardinality guard cover with $n/4$ guards (shown in white) requires $n/4$ colors, while a minimum-color guard cover (shown in black) has $n/2 + 1$ guards and requires only 3 colors.

Theorem 4.1 *For a simple polygon P with n vertices and ℓ discrete guard locations L, it can be decided in polynomial time whether there is a 2-colorable guard set.*

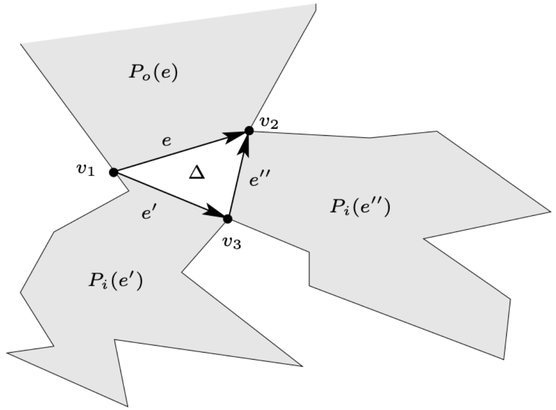


Figure 7: Solving a subproblem (e, Γ) by combining solutions for subproblems (e', Γ') and (e'', Γ'').



$$\sum_{i=1}^{n} x_i z_i \leq Z$$

$$x_i \in [0,1]$$



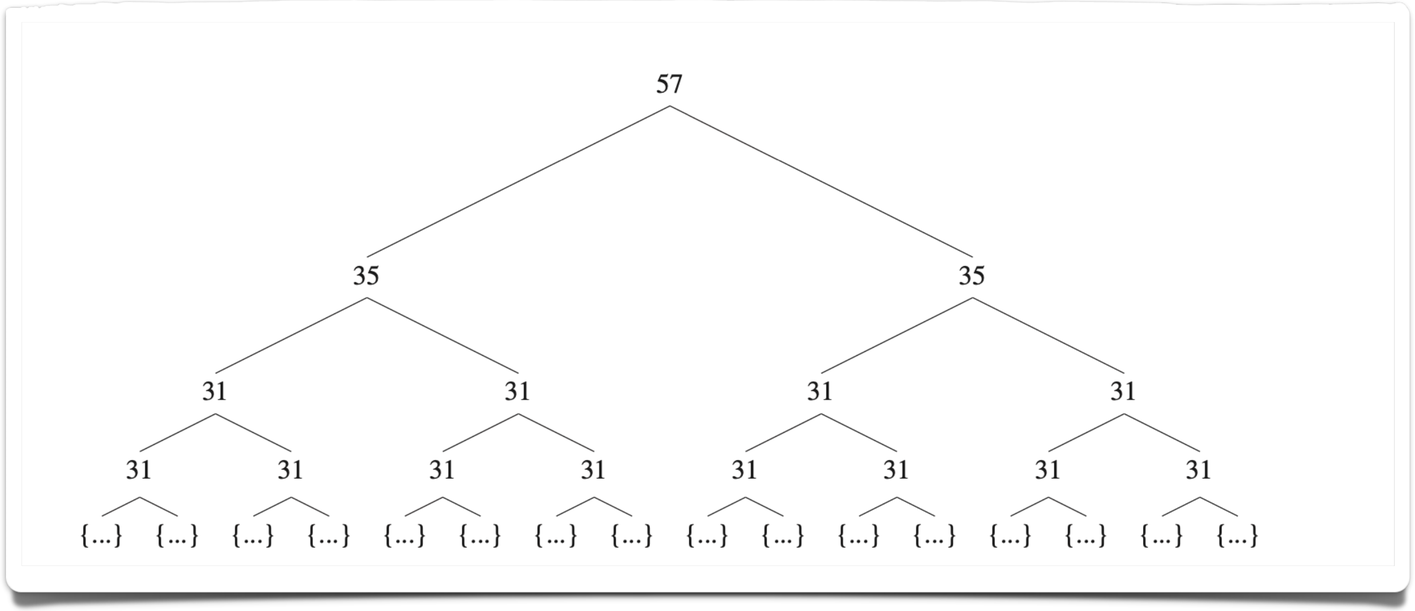
3 Branch-and-Bound

Algorithmen und Datenstrukturen 2
Sommer 2026

Prof. Dr. Sándor Fekete

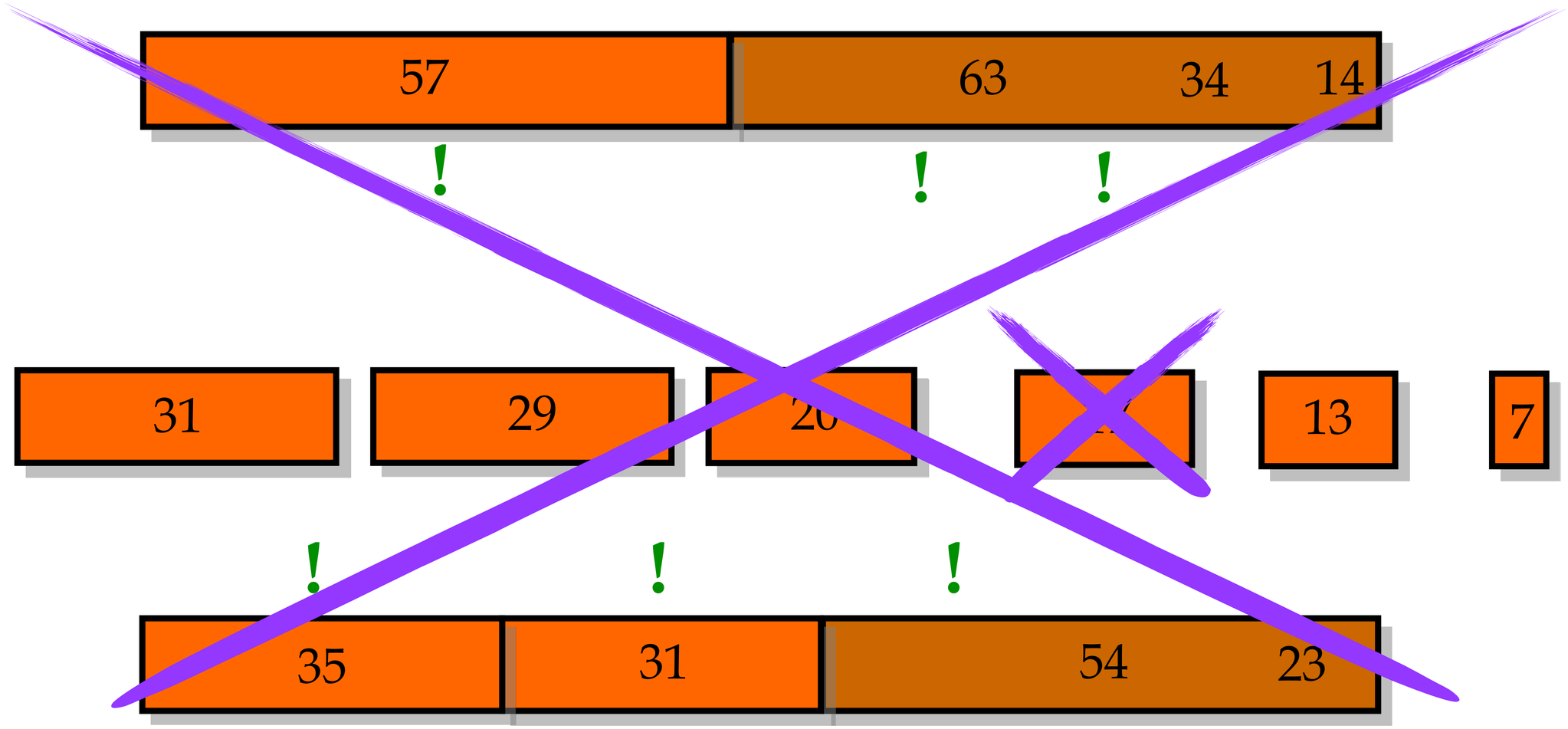
3.1 Motivation

Dynamic Programming



Branch and Bound

- Exponentiell viele Fälle!
- Wie geht das systematisch?
- Wo kann man Arbeit sparen?


57
63
34
14
!
!
!
31
29
13
7
!
!
!
35
31
54
23

Satz 2.6 *Algorithmus 2.5 berechnet besten Lösungswert für das Rucksackproblem in einer Laufzeit $O(nZ)$.*

Algorithmus 2.5 *(Dynamic Programming für Knapsack)*

Eingabe: $z_1, ..., z_n, Z, p_1, ...p_n$

Ausgabe: Funktion $P : \{1, ..., Z\} \times \{1, ..., n\} \to \mathbb{R};\ (x, i) \mapsto P(x, i)$

$$\text{mit } P(x,i) = max \sum_{i=1}^{i} p_j y_j \text{ mit } \sum_{j=1}^{i} z_j y_j \leq x, \text{ für } y_j \in \{0,1\}$$

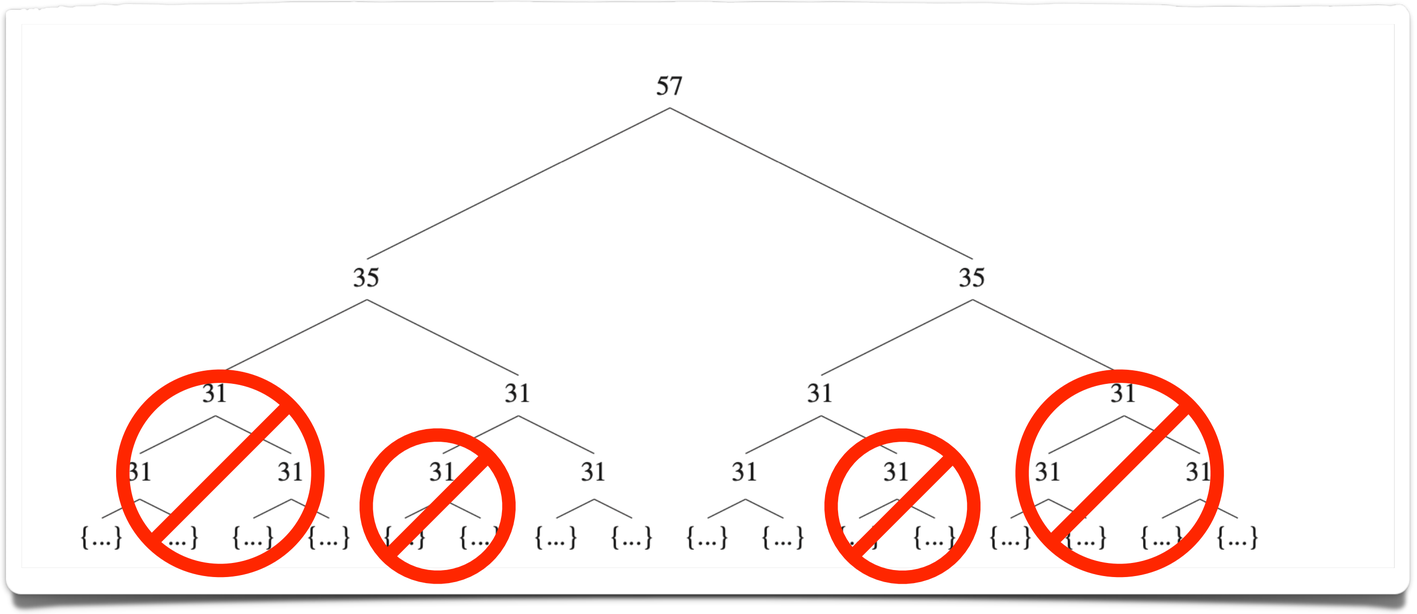
```
1: for (x = 0) to Z do
2:     P(x,0) := 0
3: for (i = 1) to n do
4:     for (x = 0) to (z_i − 1) do
5:         P(x,i) := P(x,i−1)
6:     for (x = z_i) to Z do
7:         if ((P(x − z_i, i−1) + p_i) > P(x,i−1)) then
8:             P(x,i) := P(x − z_i, i−1) + p_i
9:         else
0:             P(x,i) := P(x,i−1)
```

$O(nZ)$

i \ x	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6
2	0	0	6	6	6	11	11	11	11	11
3	0	0	6	6	6	11	11	11	14	14
4	0	0	6	6	6	11	11	11	14	15
5	0	0	6	6	6	11	11	12	14	15
6	0	0	6	6	6	11	11	12	14	15
7	0	0	6	6	6	11	11	12	14	15

Induktion!

1. Objekt i ist zu teuer für Kostenschranke x, falls $z_i > x$
2. Objekt i ist nicht zu teuer, bringt aber keine Verbesserung
3. Objekt i ist nicht zu teuer und bringt eine Verbesserung



Branch and Bound

- Beschneide Enumerationsbaum
- Verfolge Mindest- und Maximalwerte



Ailsa H. Land



Alison Doig

1960

ECONOMETRICA

VOLUME 28 July, 1960 NUMBER 3

AN AUTOMATIC METHOD OF SOLVING DISCRETE PROGRAMMING PROBLEMS

BY A. H. LAND AND A. G. DOIG

In the classical linear programming problem the behaviour of continuous, nonnegative variables subject to a system of linear inequalities is investigated. One possible generalization of this problem is to relax the continuity condition on the variables. This paper presents a simple numerical algorithm for the solution of programming problems in which some or all of the variables can take only discrete values. The algorithm requires no special techniques beyond those used in ordinary linear programming, and lends itself to automatic computing. Its use is illustrated on two numerical examples.

1. INTRODUCTION

THERE IS A growing literature [**1**, **3**, **5**, **6**] about optimization problems which could be formulated as linear programming problems with additional constraints that some or all of the variables may take only integral values. This form of linear programming arises whenever there are indivisibilities. It is not meaningful, for instance, to schedule 3--7/10 flights between two cities, or to undertake only 1/4 of the necessary setting up operation for running a job through a machine shop. Yet it is basic to linear programming that the variables are free to take on any positive value,[1] and this sort of answer is very likely to turn up.

In some cases, notably those which can be expressed as transport problems, the linear programming solution will itself yield discrete values of the variables. In other cases the percentage change in the maximand[2] from common sense rounding of the variables is sufficiently small to be neglected. But there remain many problems where the discrete variable constraints are significant and costly.

Until recently there was no general automatic routine for solving such problems, as opposed to procedures for proving the optimality of conjectured solutions, and the work reported here is intended to fill the gap. About the time of its completion an alternative method was proposed by Gomory [**5**] and subsequently extended by Beale [**1**]. Gomory's method

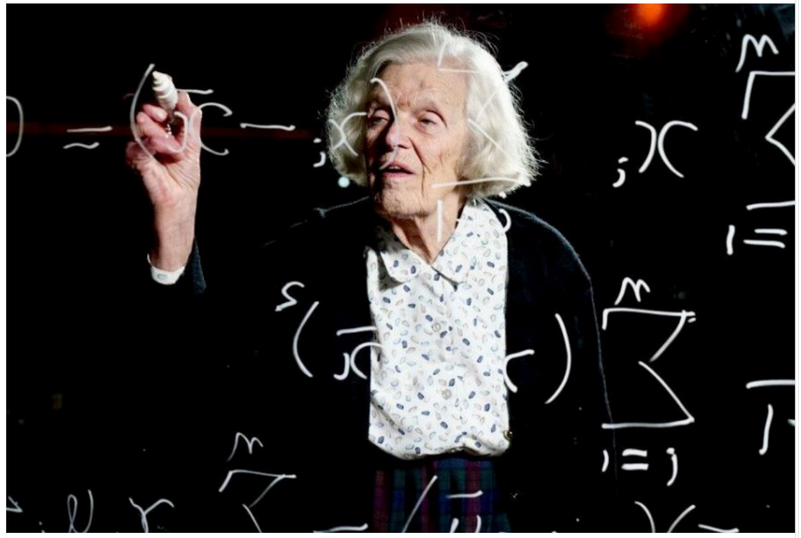
[1] Or more generally, any value within a bounded interval.

[2] We shall speak throughout of maximisation, but of course an exactly analogous argument applies to minimisation.

'I've always loved numbers': Meet the ground-breaking grandmother of Australian mathematics

7.30 / By Lauren Day

Posted 8 Oct 2018, updated 10 Oct 2018



Alison Grant Harcourt AO

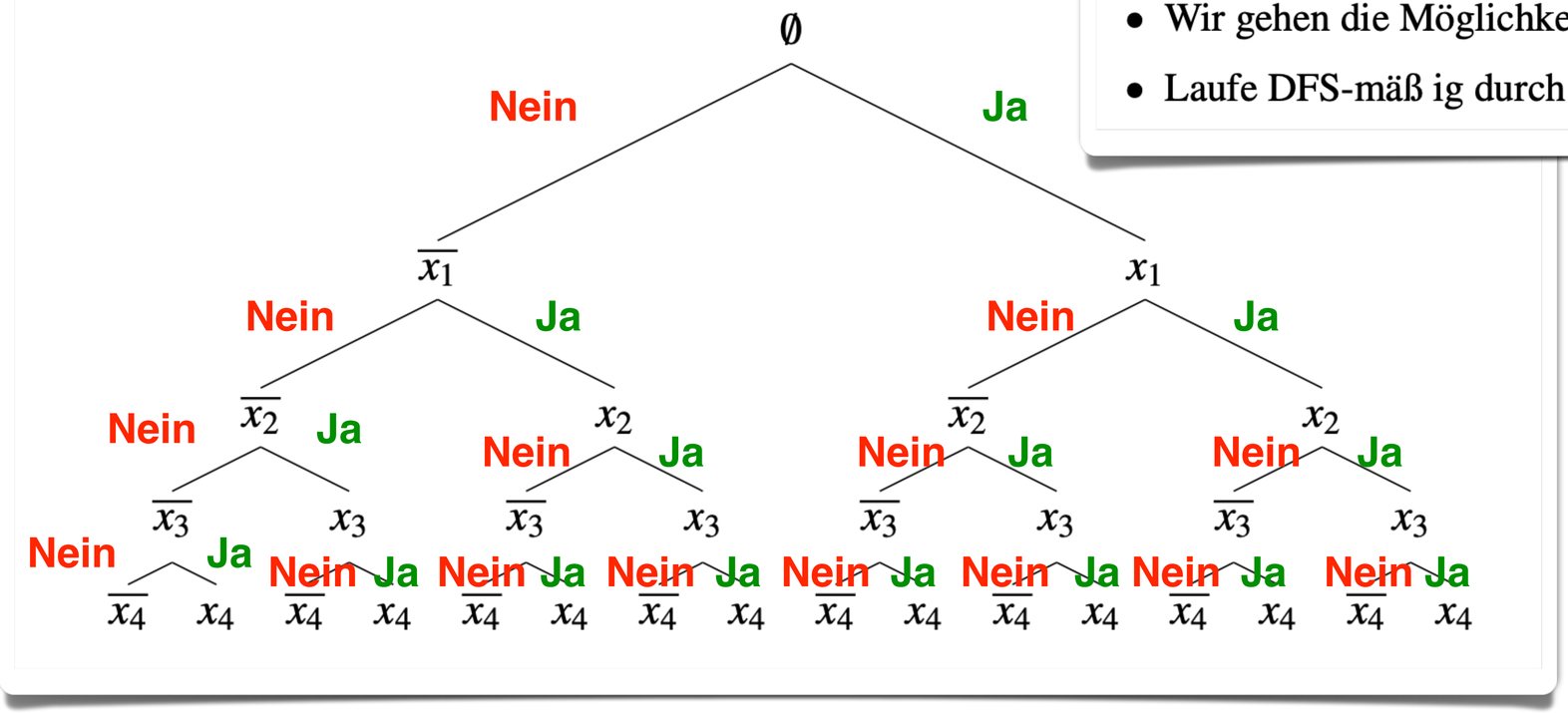
In June 2019, Harcourt was made an Officer of the Order of Australia

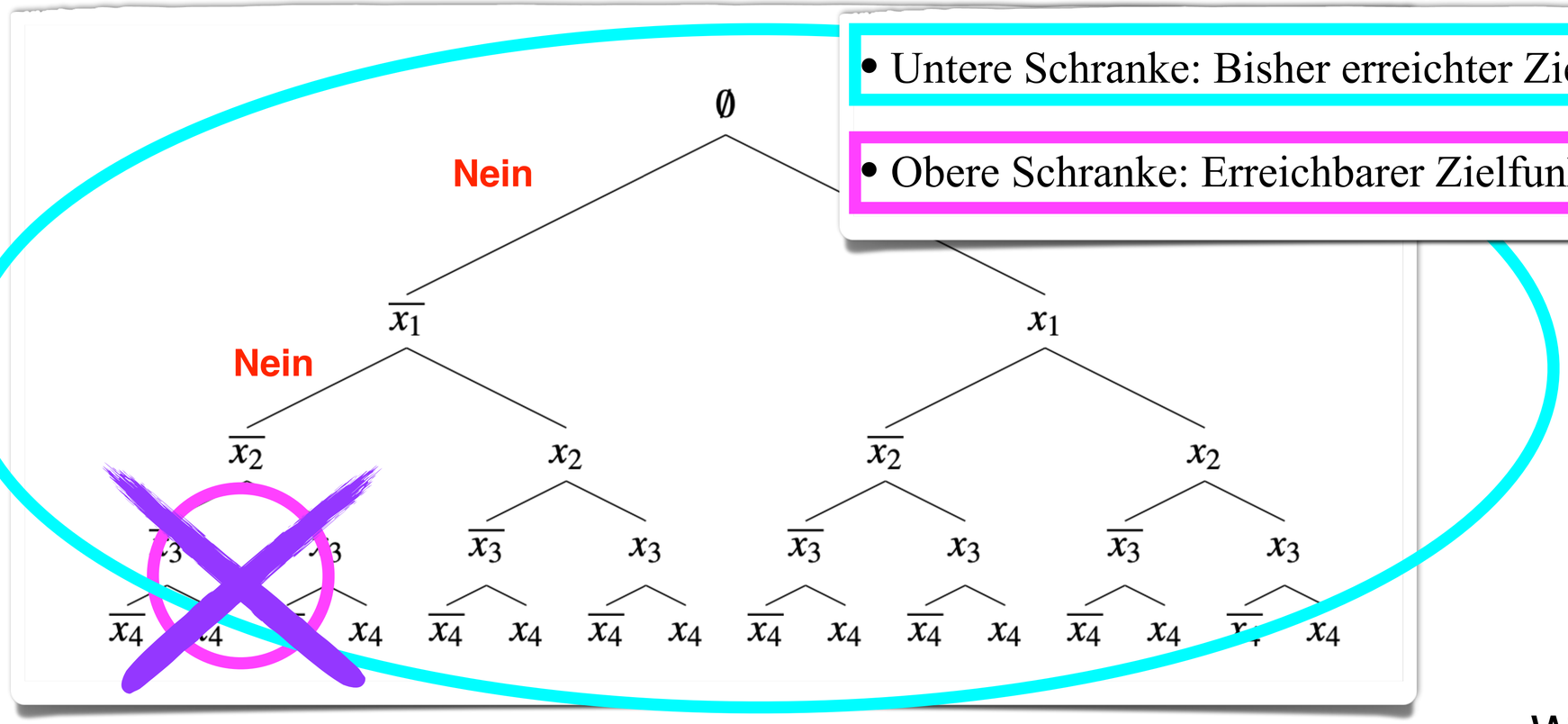
3.2 Methodik

Grundideen:

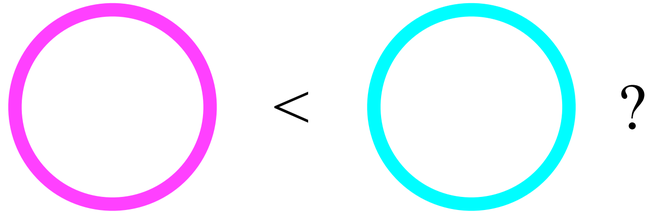
1. Enumeriere die möglichen Teilmengen in einem Enumerationsbaum
2. Behalte den Zielfunktionswert im Auge:
   - Untere Schranke: Erreichter Zielfunktionswert im ganzen Baum
   - Obere Schranke: Erreichbarer Zielfunktionswert im jeweiligen Baum
3. Beide Schranken sollten möglichst einfach und schnell zu berechnen sein
4. Wenn der erreichbare Wert in einem Teilbaum kleiner bleiben muss als der im ganzen Baum bereits erreichte, können wir den aktuellen Teilbaum abschneiden

- Probiere nacheinander für $i = 1, ..., 7$, ob $x_i = 0$ oder $x_i = 1$
- Wir gehen die Möglichkeiten baumartig durch:
- Laufe DFS-mäß ig durch den Baum, d.h. arbeite die Entscheidungen im Stack ab.

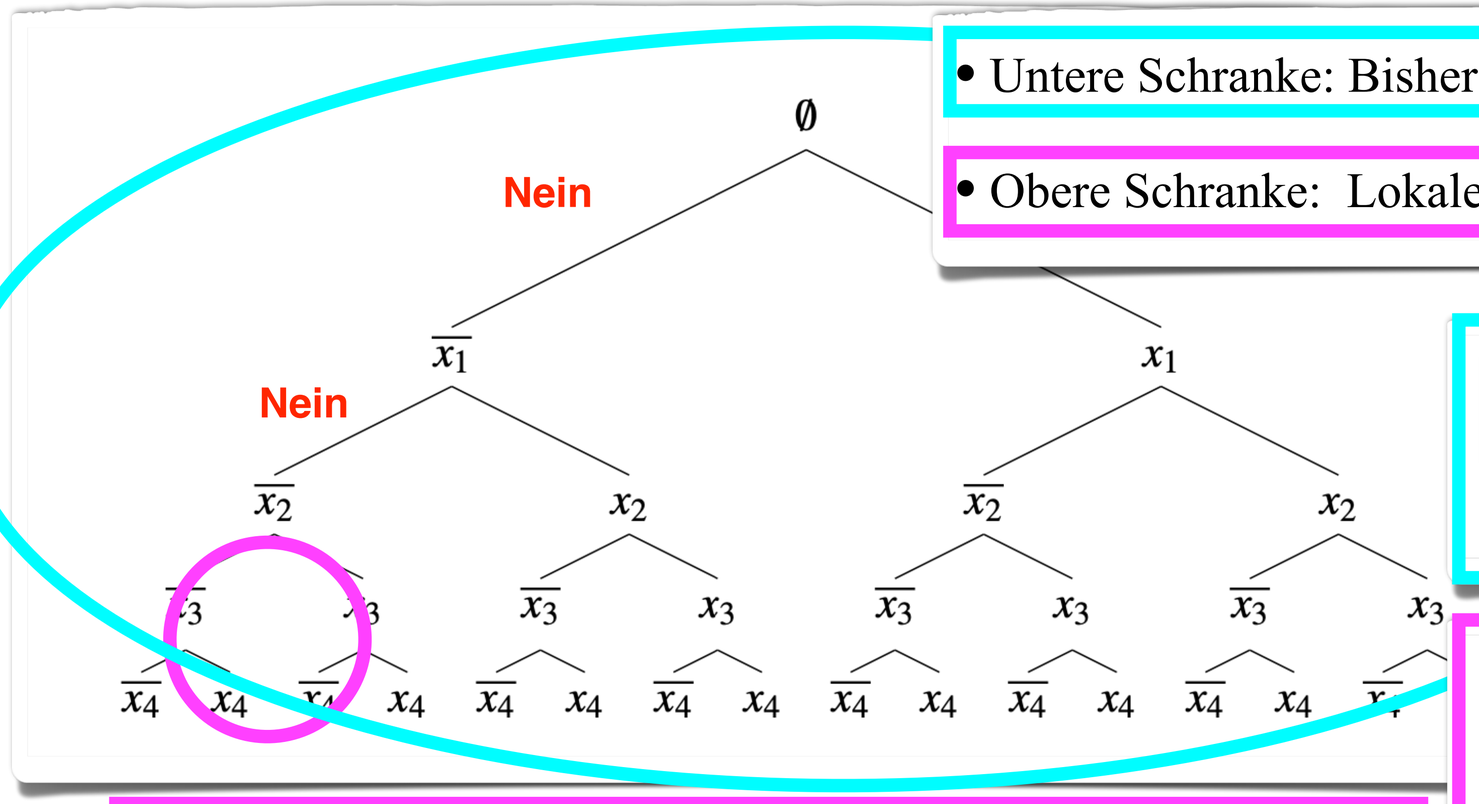




- Untere Schranke: Bisher erreichter Zielfunktionswert im ganzen Baum
- Obere Schranke: Erreichbarer Zielfunktionswert im jeweiligen Teilbaum



Was sind geeignete Schranken?



- Untere Schranke: Bisheriger Bestwert für Greedy (ganzzahlig)
- Obere Schranke: Lokaler erreichbarer Bestwert für Greedy (fraktional)

i	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
z_i	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
p_i	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

$\sum_{i=1}^{n} p_i x_i$ = 44

i	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
z_i	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
p_i	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

$\sum_{i=1}^{n} p_i x_i$ = 46 x_{15} = 0,6

Relaxierung: Einfacher durch größere Lösungsmenge

- Untere Schranke: Bisheriger Bestwert für Greedy (ganzzahlig)
- Obere Schranke: Lokaler erreichbarer Bestwert für Greedy (fraktional)

i	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
z_i	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
p_i	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

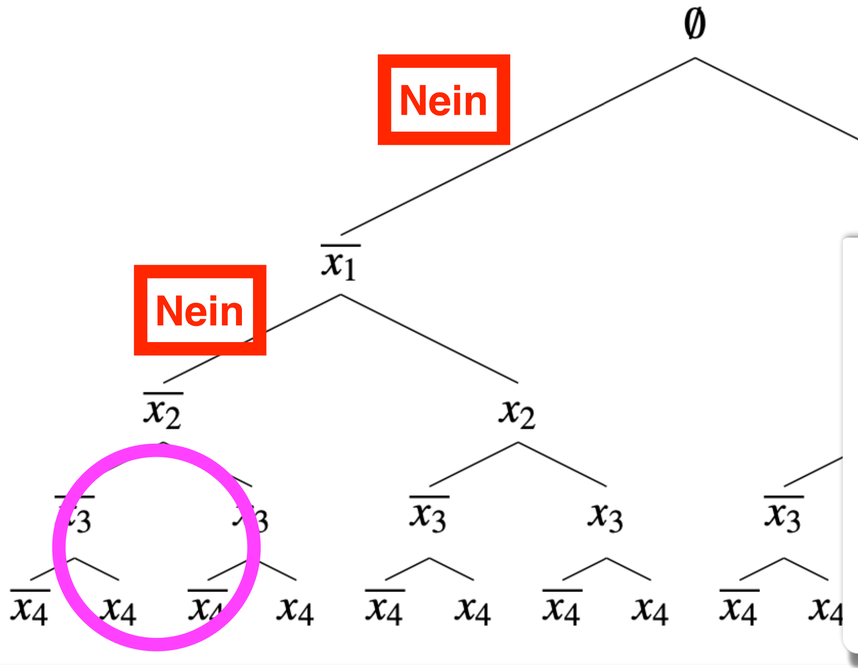
$\sum_{i=1}^{n} p_i x_i$ = 44

i	6	4	13	12	9	3	15	8	16	10	1	14	5	11	2	7
z_i	4	8	16	20	8	40	40	40	24	32	20	20	16	28	32	32
p_i	4	5	10	9	2	10	10	9	4	5	3	3	2	3	3	2

$\sum_{i=1}^{n} p_i x_i$ = 46 x_{15} = 0,6

Heuristik: Schnell zu lösen, aber nicht notwendig optimal

Relaxierung: Schnell und optimal zu lösen durch größere Lösungsmenge



S: positiv fixiert ($x_i = 1$)

S: negativ fixiert ($x_i = 0$)

Bisherige Entscheidungen fixieren Teilmenge!

i	1	2	3	3	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

3.3 Durchführung

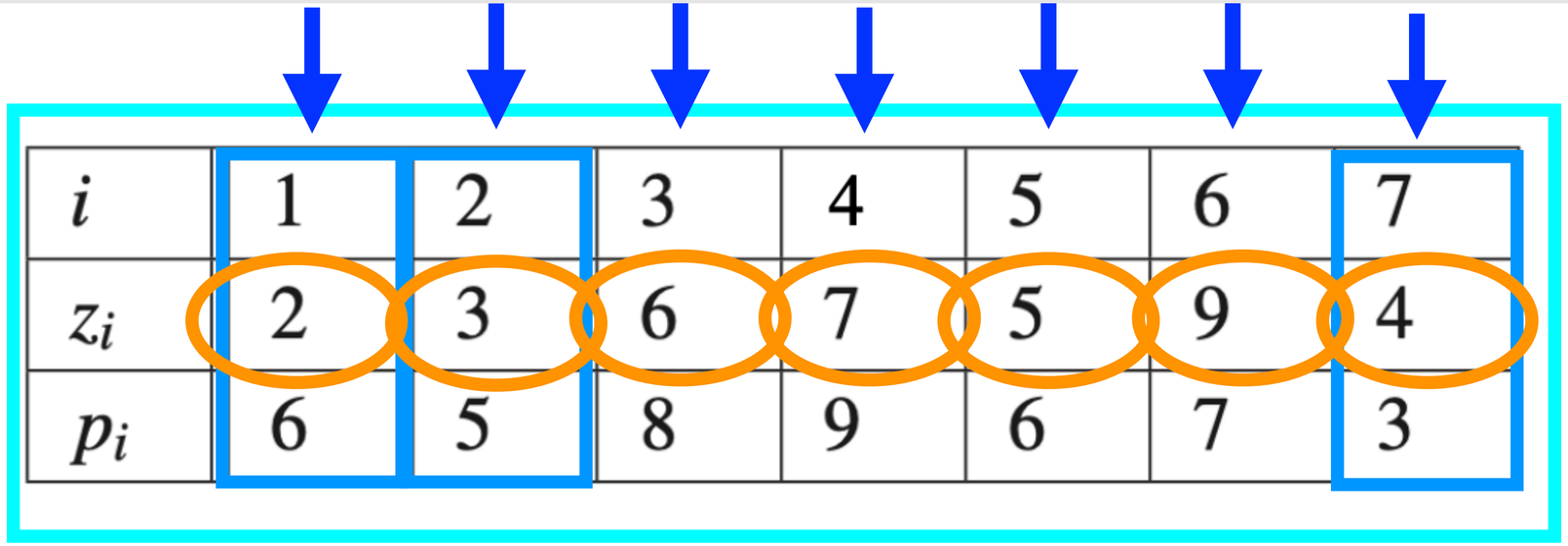
Beispiel 2.4 (Knapsackproblem).

Sei $Z = 9$ und seien folgende sieben Objekte gegeben:

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

S=Ø

$\bar{S}$=Ø



i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

Z=9

i=7 : $\sum_{i \in S} z_i = 9, \sum_{i \in S} p_i = 14$

LB=14

S=Ø

$\bar{S}$=Ø

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

Z=9

i=2 : $\sum_{i \in S} z_i = 5, \sum_{i \in S} p_i = 11$

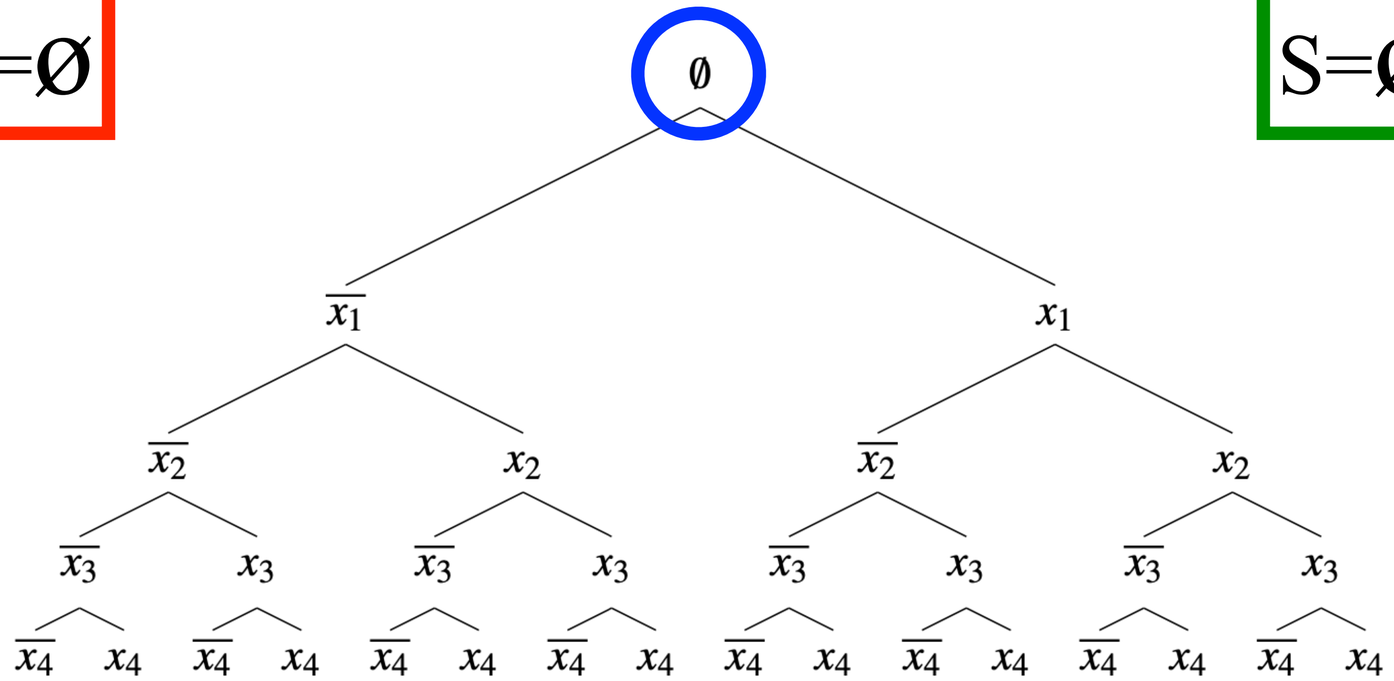
Bleibt: $Z - \sum_{i \in S} z_i = 4$; also $x_3 = \frac{2}{3} (= \frac{4}{6} = \frac{Z - \sum_{i \in S} z_i}{z_3})$

Damit: $\sum_{i \in S} x_i z_i = 9, \sum_{i \in S} x_i p_i = 11 + \frac{2}{3} * 8 = 16, \bar{3}$

UB=16

$\bar{S}=\emptyset$

$S=\emptyset$



i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

UB=16

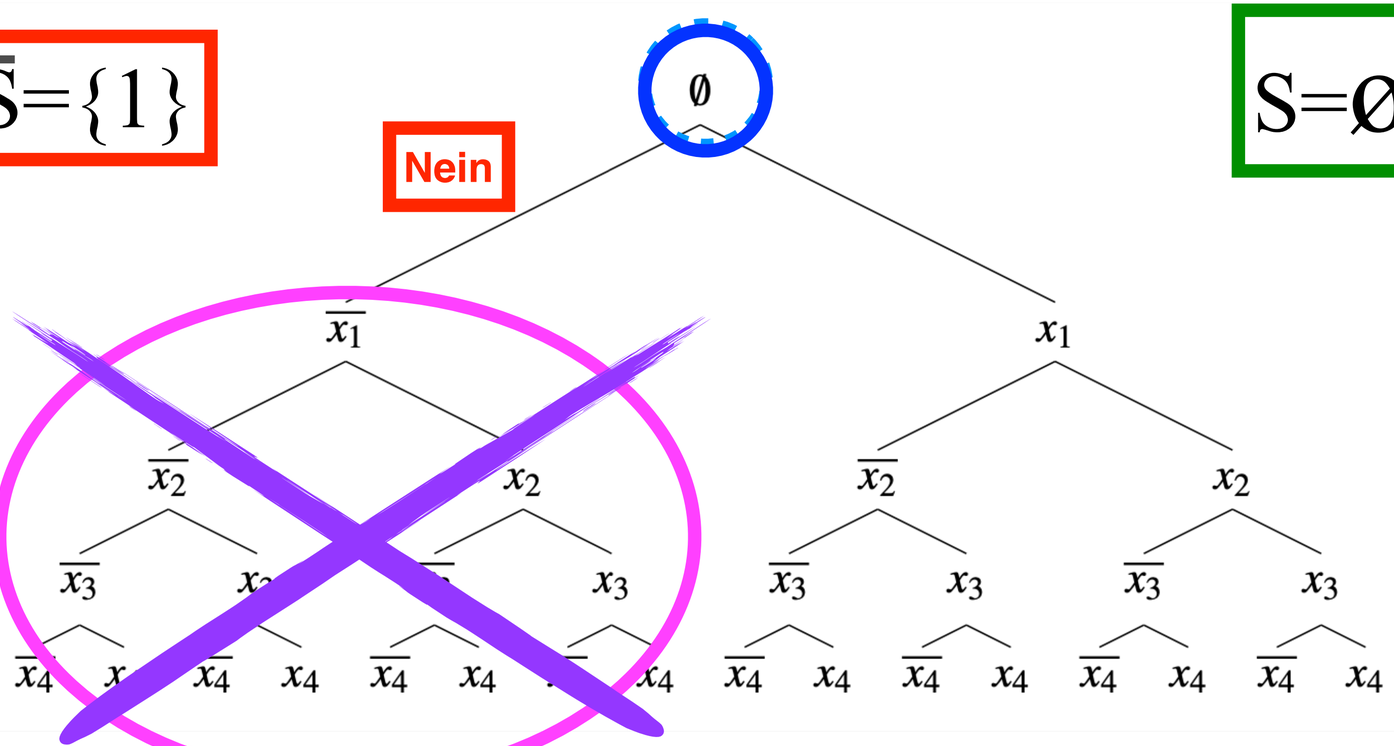
i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

LB=14

$\bar{S}=\{1\}$

Nein

$S=\emptyset$



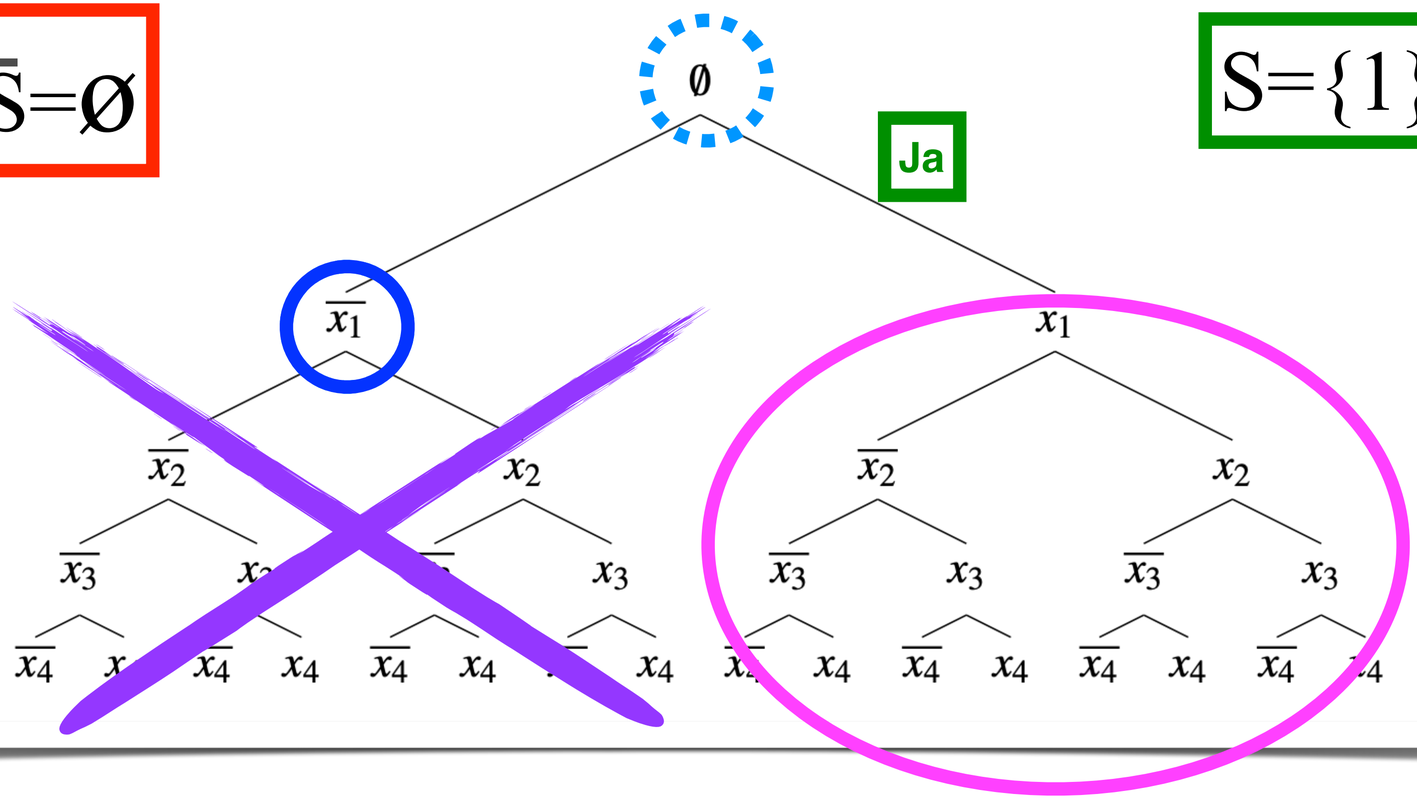
i	1	2	3	4	5	6	7
z_i		3	6	7	5	9	4
p_i		5	8	9	6	7	3

UB=13

i	1	2	3		5	6	7
z_i	2	3	6		5	9	4
p_i	6	5	8	9	6	7	3

LB=14

$\bar{S}=\emptyset$



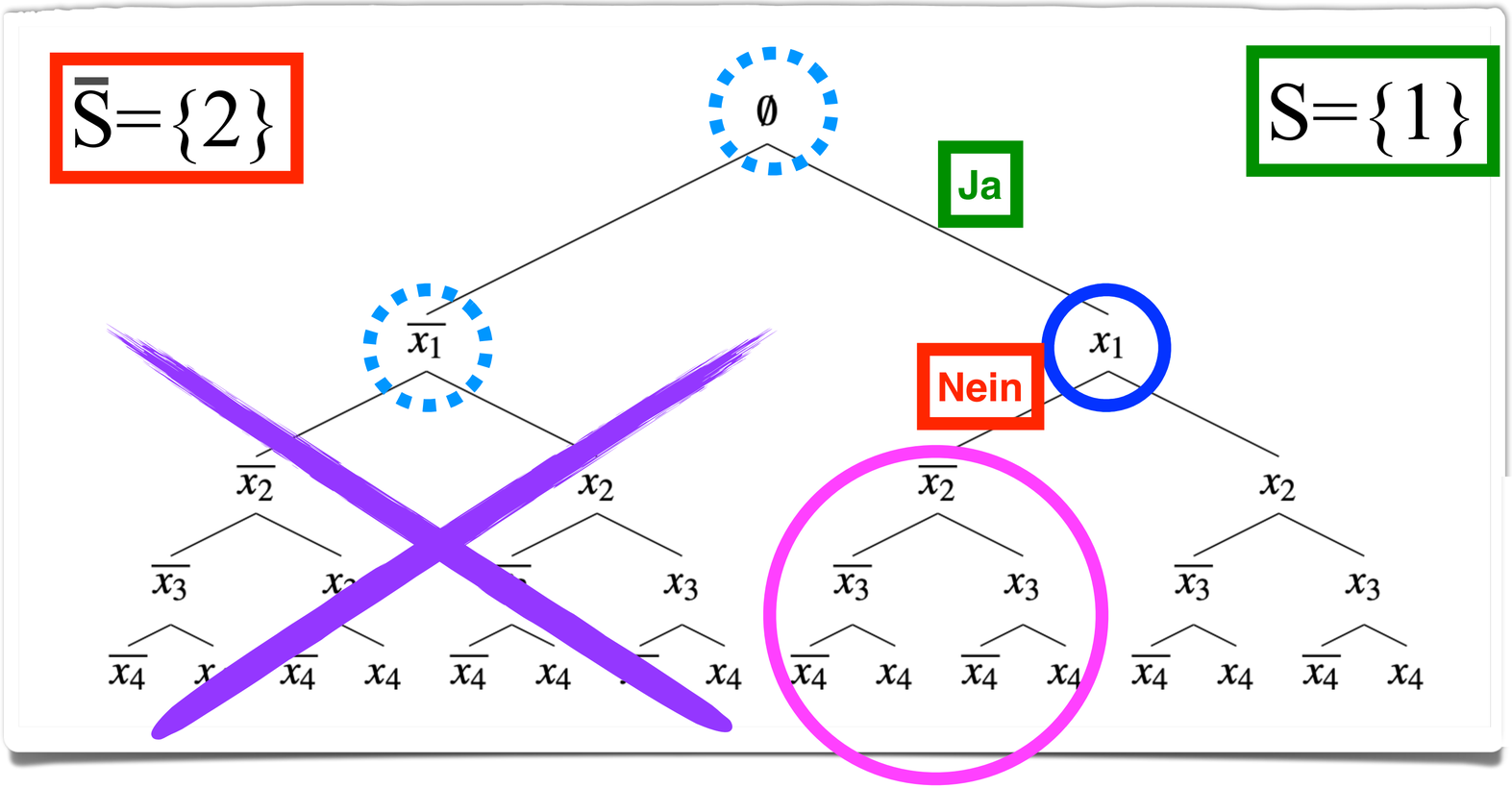
$S=\{1\}$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

UB=16

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

LB=14

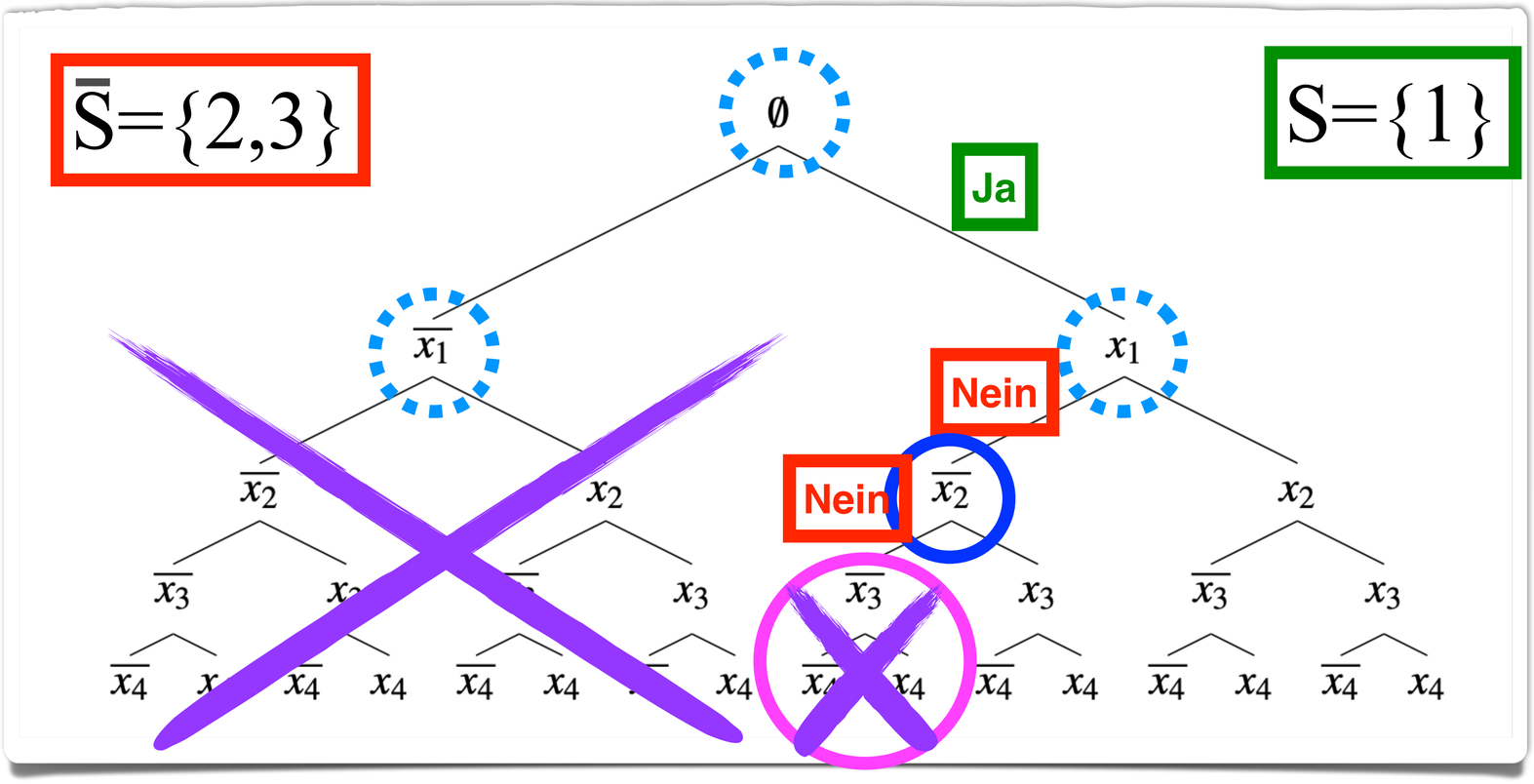


i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

UB=15

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

LB=14

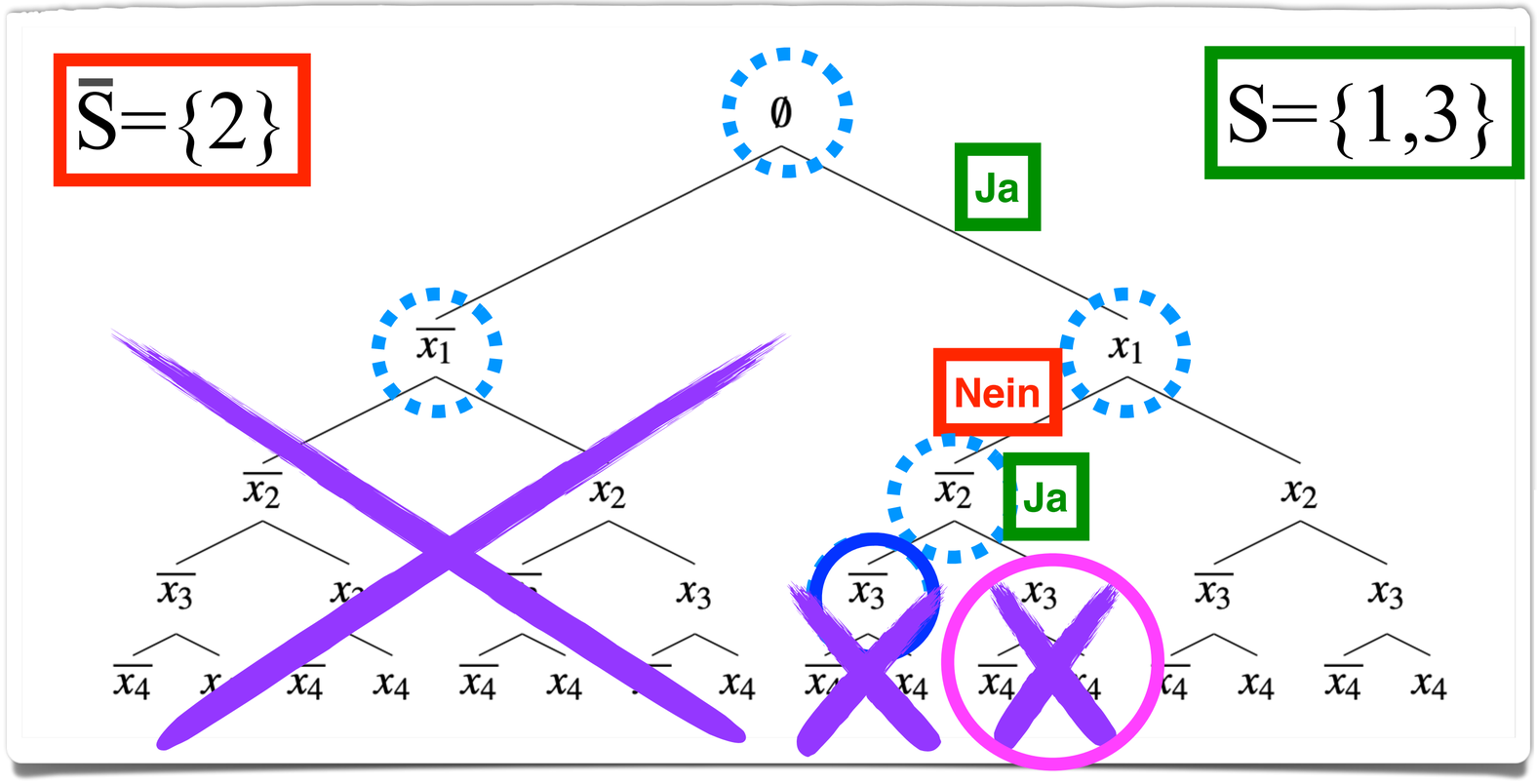


i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

UB=15

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

LB=14

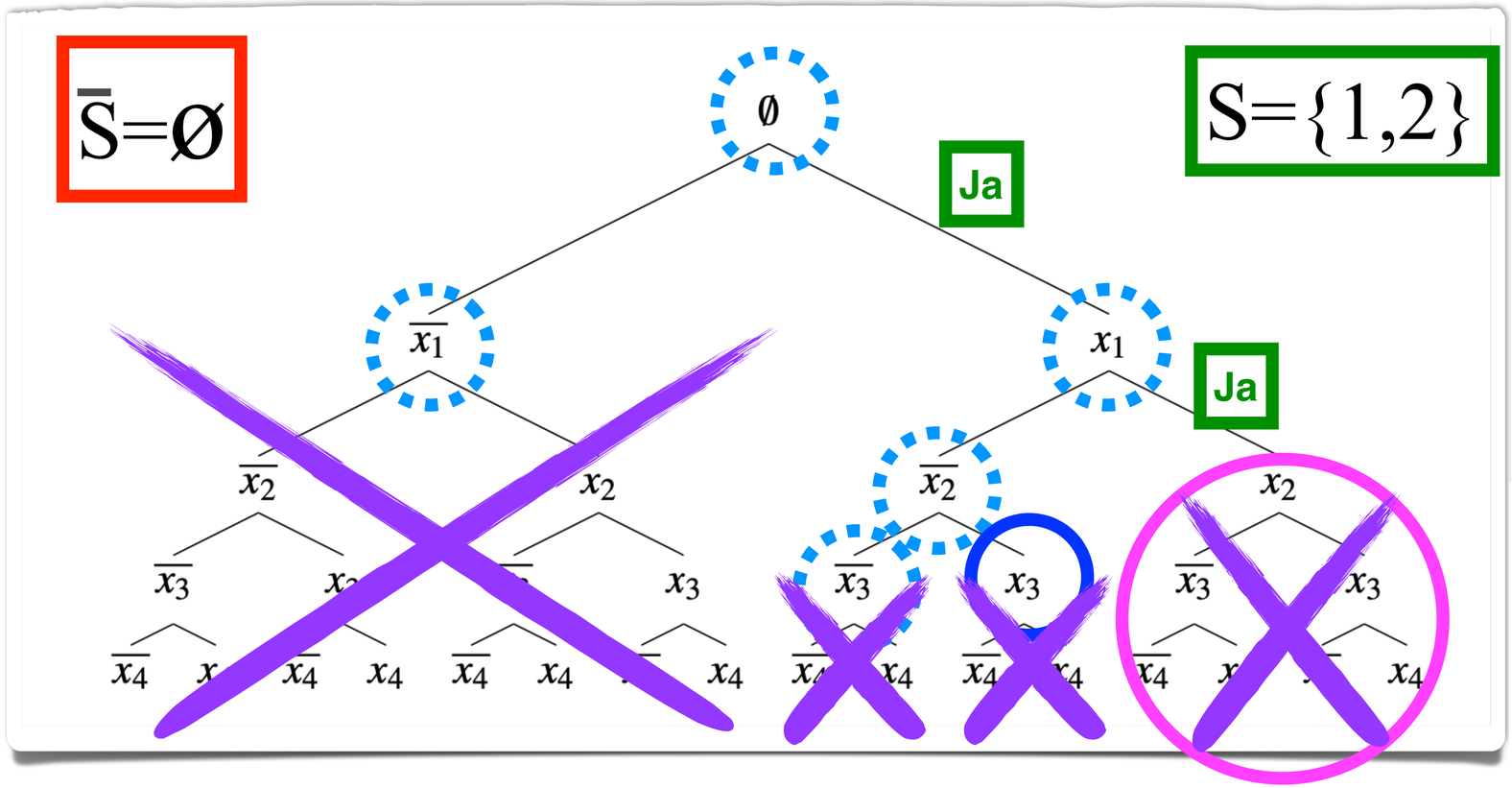


i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	6	8	9	6	7	3

UB=15

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	6	8	9	6	7	3

LB=15

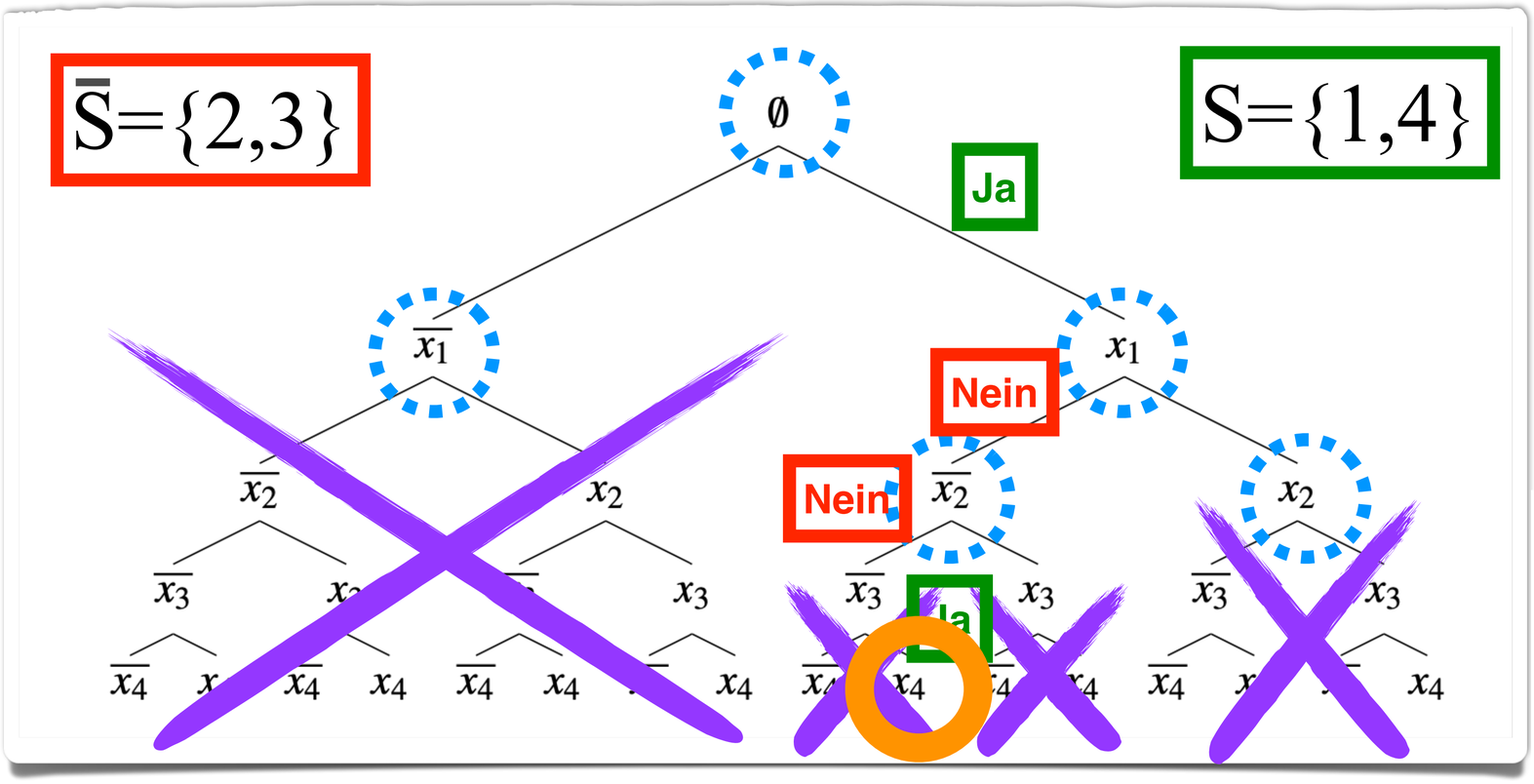


i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	5	8	9	6	7	3

UB=14

i	1	2	3		5	6	7
z_i	2	3	6		5	9	4
p_i	6	5	8	9	6	7	3

LB=15



Also: Das Optimum ist 15 mit $S = \{1, 4\}$

i	1	2	3	4	5	6	7
z_i	2	3	6	7	5	9	4
p_i	6	6	8	9	6	7	3

LB=15

Algorithmus 3.1 (Branch-And-Bound als Unterroutine).

Eingabe: $z_1, ..., z_n, Z, p_1, ...p_n$ *(global: Kosten/Nutzenwerte, Kostenschranke)*

P *(bester bekannter Lösungswert)*

ℓ *(nächster Index, über den verzweigt werden soll)*

$x_j = b_j$ *für* $j = 1, ..., \ell - 1$ *mit* $b_j \in \{0, 1\}$ *(bislang fixierte Binärvariable)*

Ausgabe: $\max \left\{ \sum_{j=1}^{\ell-1} b_j p_j + \sum_{j=\ell}^{n} x_j p_j \mid \sum_{j=1}^{\ell-1} b_j z_j + \sum_{j=\ell}^{n} x_j z_j \leq Z, x_j \in \{0, 1\} \right\}$

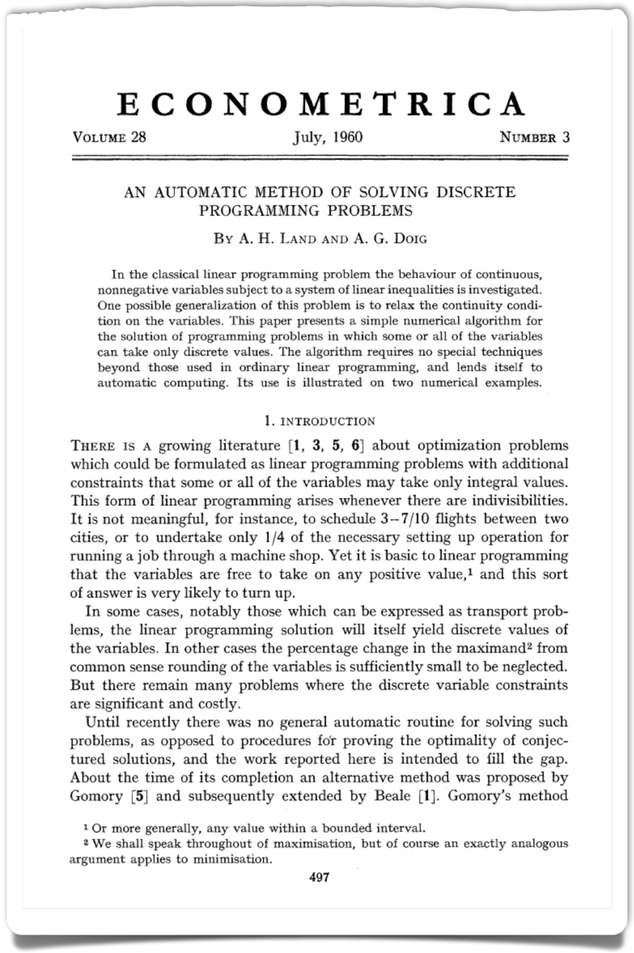
Also: Lösung des Knapsackproblems mit den ersten $\ell - 1$ *Variablen fixiert*

```
1:  procedure BRANCH-AND-BOUND(ℓ)
2:      if (Σ_{j=1}^{ℓ-1} b_j z_j > Z) then return                ▷ unzulässig
3:      Compute L := LB(b_1, ..., b_{ℓ-1})
4:      if L > P then P := L                                      ▷ Lösungswert verbessert
5:      if (ℓ > n) then return                                    ▷ Blatt im Baum erreicht
6:      U := UB(b_1, ..., b_{ℓ-1})                                ▷ (Obere Schranke berechnen)
7:      if (U > P) then
8:          b_ℓ := 0; BRANCH-AND-BOUND(ℓ + 1);
9:          b_ℓ := 1; BRANCH-AND-BOUND(ℓ + 1);
10:     return
```

Satz 3.2 *Algorithmus 2.10 (als rekursiv arbeitende Unterroutine) berechnet eine optimale Lösung für das Knapsackproblem in einer Worst-Case-Laufzeit $O(2^n f(n))$, wobei $f(n)$ die Zeit für die Berechnung der oberen Schranke ist.*

Beweis. Es werden systematisch alle Teillösungen durchprobiert. (Dabei werden Teilmengen nur dann ausgelassen, wenn sie erwiesenermaßen unzulässig sind (Zeile 2) oder keine Verbesserung bringen (Zeile 4)
Die Zahl der Rekursionsaufrufe (Schritt 9 und 11) ist insgesamt 2^n, die sonstigen Berechnungen benötigen jeweils $f(n)$. □

3.4 Ausblicke



ECONOMETRICA
VOLUME 28 July, 1960 NUMBER 3

AN AUTOMATIC METHOD OF SOLVING DISCRETE PROGRAMMING PROBLEMS

BY A. H. LAND AND A. G. DOIG

In the classical linear programming problem the behaviour of continuous, nonnegative variables subject to a system of linear inequalities is investigated. One possible generalization of this problem is to relax the continuity condition on the variables. This paper presents a simple numerical algorithm for the solution of programming problems in which some or all of the variables can take only discrete values. The algorithm requires no special techniques beyond those used in ordinary linear programming, and lends itself to automatic computing. Its use is illustrated on two numerical examples.

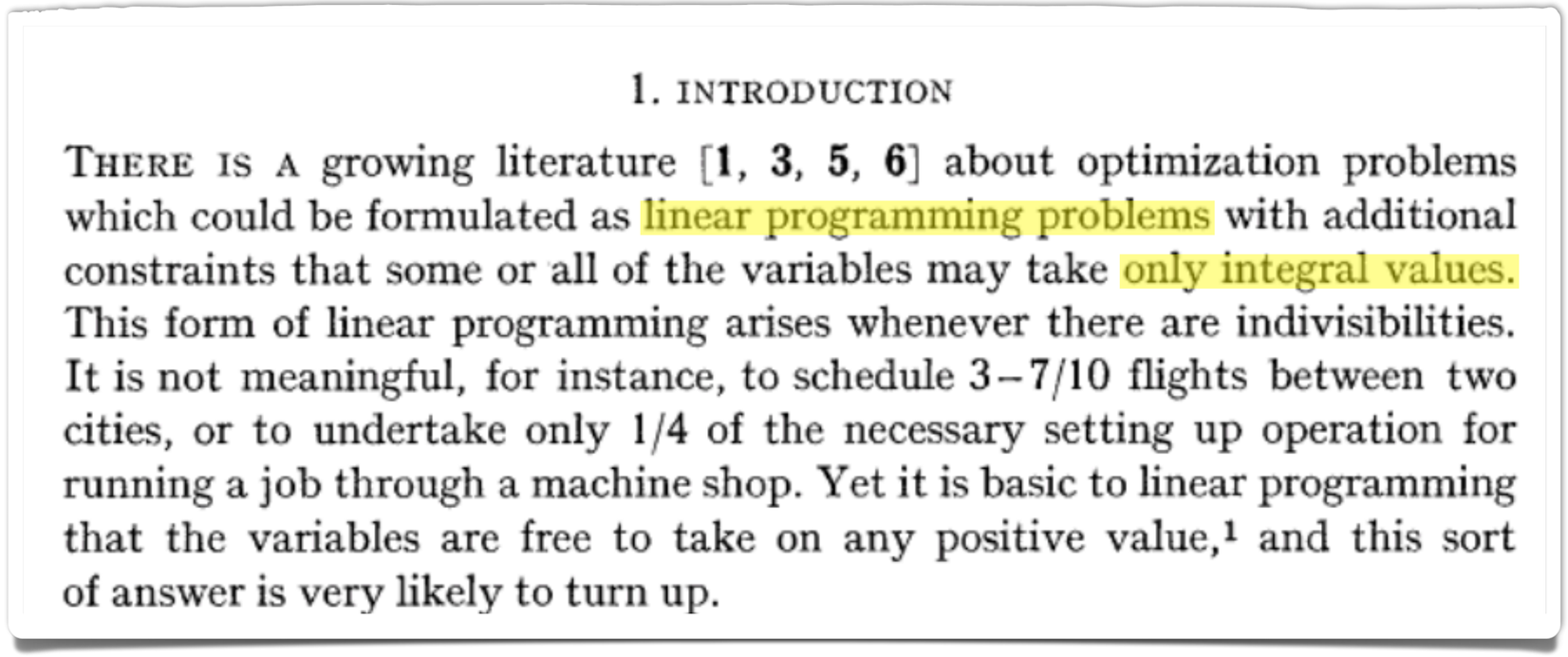
1. INTRODUCTION

THERE IS A growing literature [1, 3, 5, 6] about optimization problems which could be formulated as linear programming problems with additional constraints that some or all of the variables may take only integral values. This form of linear programming arises whenever there are indivisibilities. It is not meaningful, for instance, to schedule 3–7/10 flights between two cities, or to undertake only 1/4 of the necessary setting up operation for running a job through a machine shop. Yet it is basic to linear programming that the variables are free to take on any positive value,[1] and this sort of answer is very likely to turn up.

In some cases, notably those which can be expressed as transport problems, the linear programming solution will itself yield discrete values of the variables. In other cases the percentage change in the maximand[2] from common sense rounding of the variables is sufficiently small to be neglected. But there remain many problems where the discrete variable constraints are significant and costly.

Until recently there was no general automatic routine for solving such problems, as opposed to procedures for proving the optimality of conjectured solutions, and the work reported here is intended to fill the gap. About the time of its completion an alternative method was proposed by Gomory [5] and subsequently extended by Beale [1]. Gomory's method

[1] Or more generally, any value within a bounded interval.
[2] We shall speak throughout of maximisation, but of course an exactly analogous argument applies to minimisation.



1. INTRODUCTION

THERE IS A growing literature [1, 3, 5, 6] about optimization problems which could be formulated as linear programming problems with additional constraints that some or all of the variables may take only integral values. This form of linear programming arises whenever there are indivisibilities. It is not meaningful, for instance, to schedule 3–7/10 flights between two cities, or to undertake only 1/4 of the necessary setting up operation for running a job through a machine shop. Yet it is basic to linear programming that the variables are free to take on any positive value,[1] and this sort of answer is very likely to turn up.

0-1-Knapsack

$$\max \sum_{i=1}^{n} x_i p_i$$

$$\sum_{i=1}^{n} x_i z_i \leq Z$$

$$x_i \in \{0, 1\}$$

Schwieriger

Fraktionales Knapsack

$$\max \sum_{i=1}^{n} x_i p_i$$

$$\sum_{i=1}^{n} x_i z_i \leq Z$$

$$x_i \in [0, 1]$$

Einfacher

Ganzzahliges Optimierungsproblem

$$\max \sum_{i=1}^{n} x_i p_i$$

$$\begin{array}{llll} a_{11}x_1 & +\ldots & +a_{1n}x_n & \leq b_1 \\ a_{21}x_1 & +\ldots & +a_{2n}x_n & \leq b_2 \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1}x_1 & +\ldots & +a_{mn}x_n & \leq b_m \end{array}$$

$$x_i \in \{0, 1\}$$

Beweisbar schwer zu lösen

Lineares Optimierungsproblem

$$\max \sum_{i=1}^{n} x_i p_i$$

$$\begin{array}{llll} a_{11}x_1 & +\ldots & +a_{1n}x_n & \leq b_1 \\ a_{21}x_1 & +\ldots & +a_{2n}x_n & \leq b_2 \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1}x_1 & +\ldots & +a_{mn}x_n & \leq b_m \end{array}$$

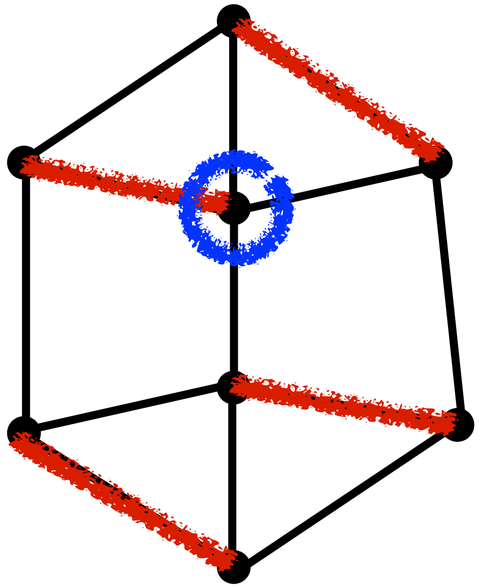
$$x_i \in [0, 1]$$

Beweisbar effizient lösbar

Gegeben: Ein Graph G=(V,E)

$$x_e = 1$$

$$\sum_{e \in E} x_e = 4$$



$$\max \sum_{e \in E} x_e$$

$$\forall v \in V : \sum_{e \in \delta(v)} x_e \leq 1$$

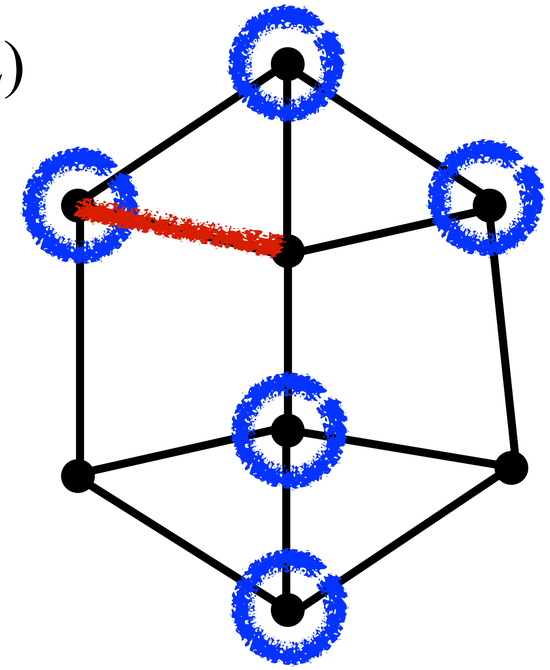
$$x_e \in \{0,1\}$$

Gesucht: Ein größtmögliches **Matching** M in G:
eine möglichst große Menge disjunkter Kanten

Gegeben: Ein Graph G=(V,E)

$$y_v = 1$$

$$\sum_{v \in V} y_v = 5$$



$$\min \sum_{v \in V} y_e$$

$$\forall e \in E : \sum_{e \in \delta(v)} y_v \geq 1$$

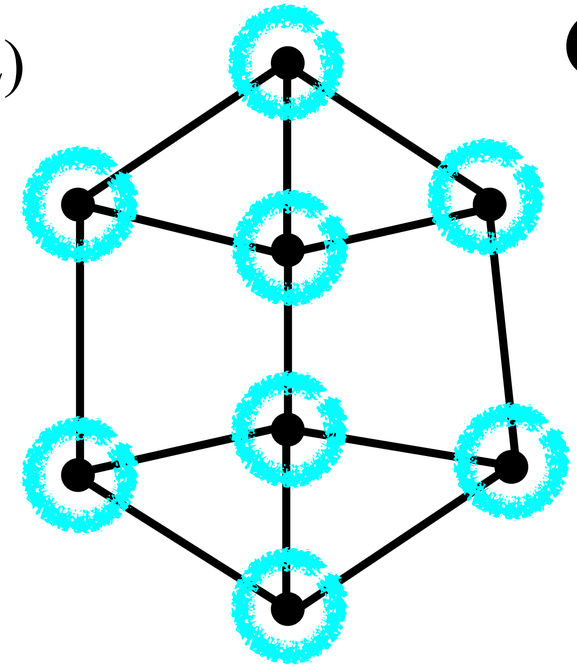
$$y_v \in \{0, 1\}$$

Gesucht: Ein kleinstmögliches **Vertex Cover** S in G:
eine möglichst kleine Menge von Knoten,
die alle Kanten überdecken

Gegeben: Ein Graph G=(V,E)

$$y_v = \frac{1}{2}$$

$$\sum_{v \in V} y_v = 4$$



Ganzzahlig?!

$$\min \sum_{v \in V} y_e$$

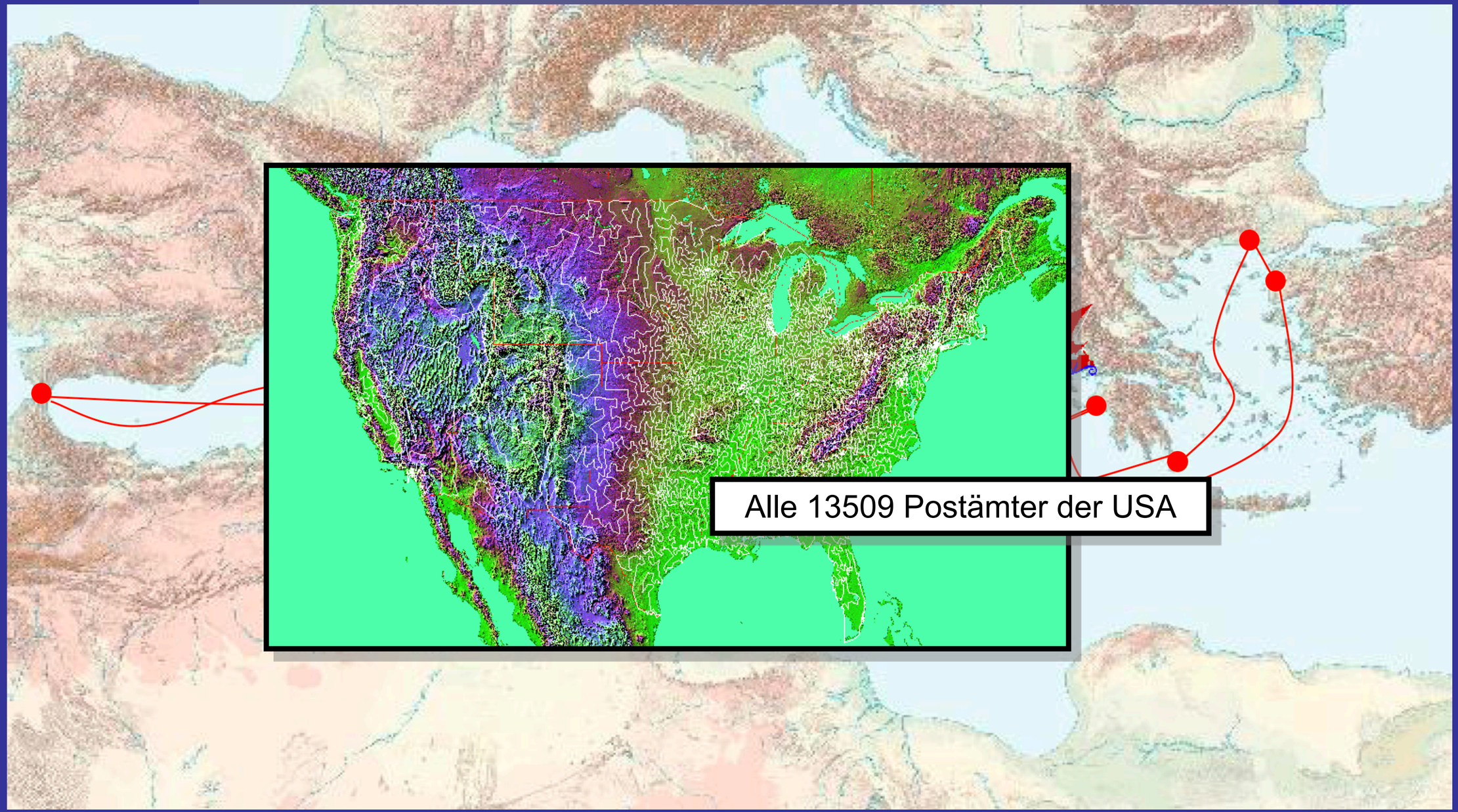
$$\forall e \in E : \sum_{e \in \delta(v)} y_v \geq 1$$

$$y_v \in [0,1]$$

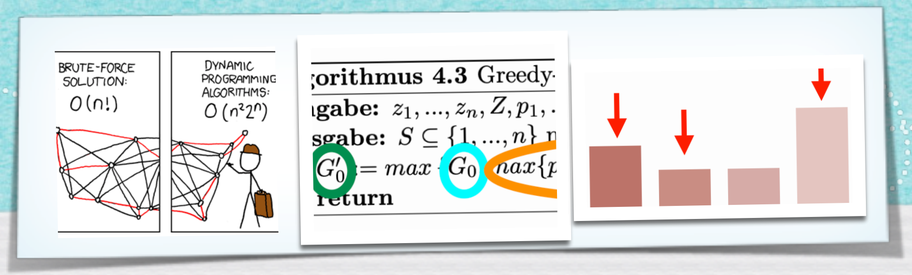
Relaxierung

Gesucht: Ein kleinstmögliches **Vertex Cover** S in G:
eine möglichst kleine Menge von Knoten,
die alle Kanten überdecken

Das Rundreiseproblem



Alle 13509 Postämter der USA



4 Approximation

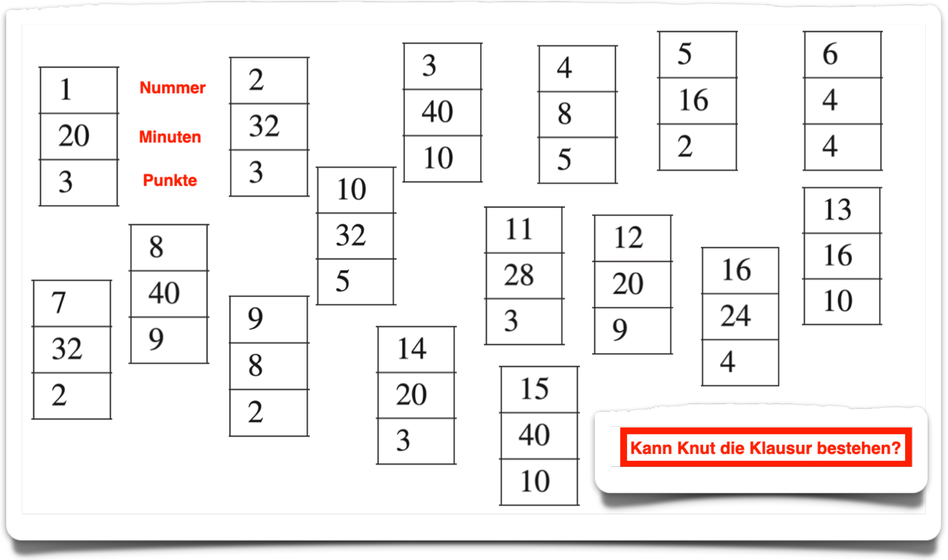
Algorithmen und Datenstrukturen 2
Sommer 2026

Prof. Dr. Sándor Fekete

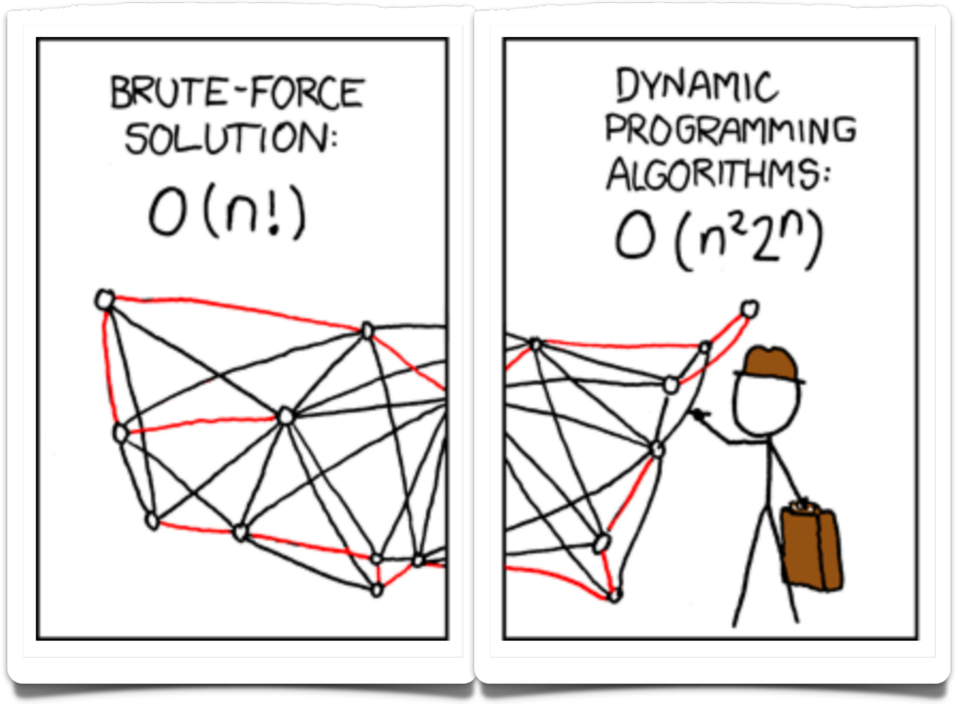
4.1 Motivation

0-1 Knapsack

$O(nZ)$ $O(2^n f(n))$

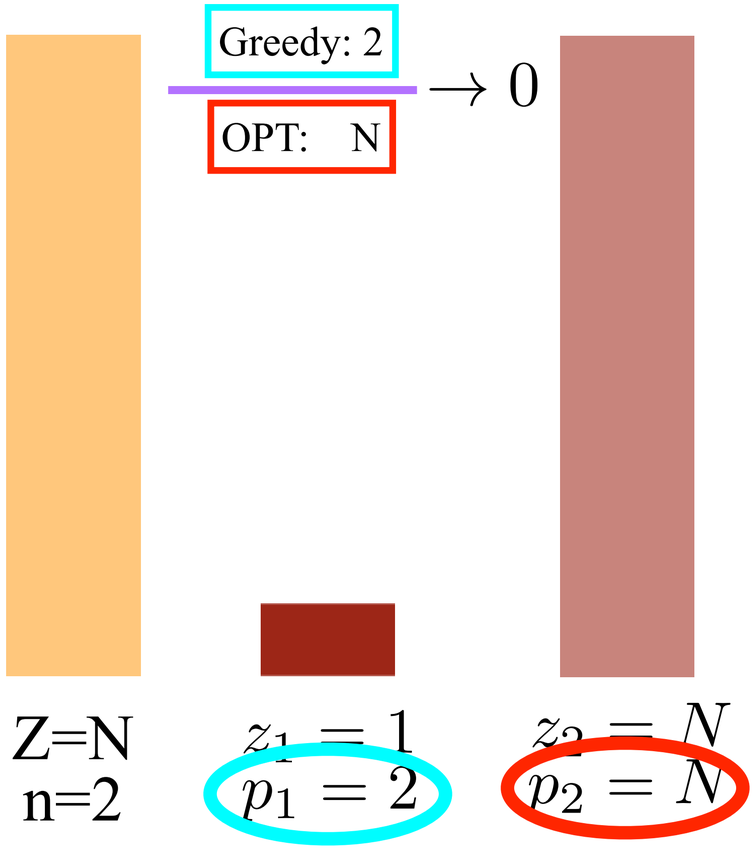


Traveling Salesman



Ziel: Vielleicht nicht die *bestmögliche* Lösung, sondern eine *gute,* aber dafür schneller…?!

4.2 Greedy und Approximation



Algorithmus 4.2 GREEDY$_0$

Eingabe: $z_1, ..., z_n, Z, p_1, ..., p_n$

Ausgabe: $S \subseteq \{1, ..., n\}$ und G_0

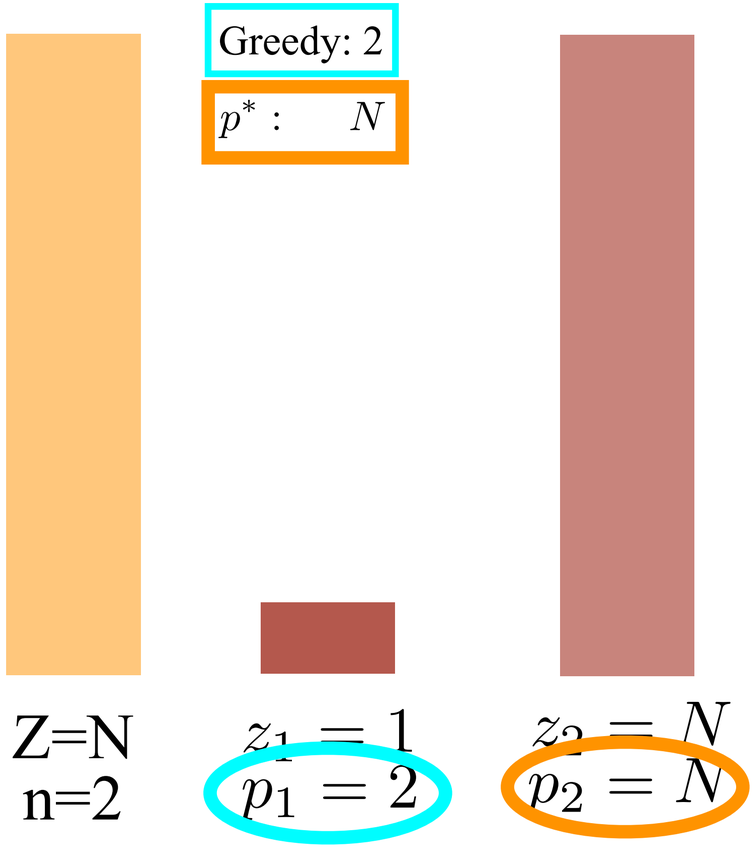
mit

$$\sum_{i \in S} z_i \leq Z$$

und

$$G_0 := \sum_{i \in S} p_i = Maximal$$

1: Sortiere {1,...,n} nach $\frac{z_i}{p_i}$ aufsteigend;
Dies ergibt die Permutation $\pi(1), ..., \pi(n)$.
Setze $j = 1$.
2: **while** $(j \leq n)$ **do**
3: **if** $\left(\sum_{i=1}^{j-1} z_{\pi(i)} x_{\pi(i)} + z_{\pi(j)} \leq Z\right)$ **then**
4: $x_{\pi(j)} := 1$
5: $j := j + 1$
6: **return**



Algorithmus 4.3 GREEDY$_0$'

Eingabe: $z_1, ..., z_n, Z, p_1, ..., p_n$

Ausgabe: $S \subseteq \{1, ..., n\}$ und G_0

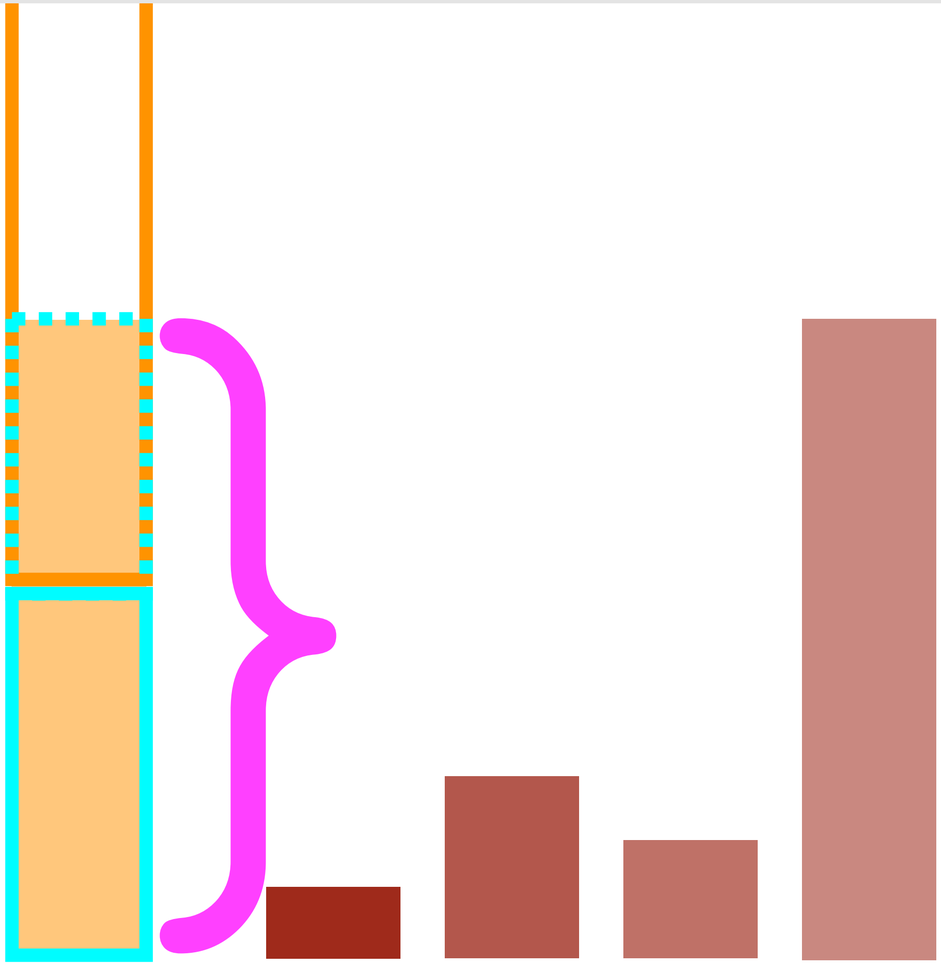
mit

$$\sum_{i \in S} z_i \leq Z$$

und

$$G_0 := \sum_{i \in S} p_i = Maximal$$

1: Sortiere {1,...,n} nach $\frac{z_i}{p_i}$ aufsteigend;
Dies ergibt die Permutation $\pi(1), ..., \pi(n)$.
Setze $j = 1$.
2: **while** $(j \leq n)$ **do**
3: **if** $\left(\sum_{i=1}^{j-1} z_{\pi(i)} x_{\pi(i)} + z_{\pi(j)} \leq Z\right)$ **then**
4: $x_{\pi(j)} := 1$
5: $j := j + 1$
6: $G_0' := max\{G_0, max\{p_i \mid z_i < Z, i \in \{1, ..., n\}\}\}$
7: **return**



Algorithmus 4.3 Greedy-Approximation

Eingabe: $z_1, ..., z_n, Z, p_1, ..., p_n$

Ausgabe: $S \subseteq \{1, ..., n\}$ mit $\sum_{i \in S} z_i \leq Z$ und Wert $G_0' := \sum_{i \in S} p_i$

1: $G_0' := max\{G_0, max\{p_i \mid z_i \leq Z, i \in \{1, ..., n\}\}\}$

2: **return**

Satz 4.4. *Algorithmus 4.3 berechnet eine Lösung mit G_0'; im Vergleich mit dem Optimalwert OPT gilt: $G_0' \geq \frac{1}{2} OPT$.*

$$G_0' \geq G_0 \qquad G_0' \geq p^*$$

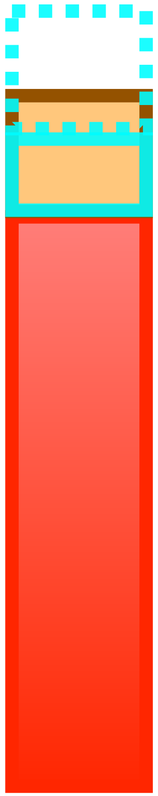
$$OPT \leq FractionalKP = G_0 + x_i p_i \leq G_0 + p^* \leq 2G_0'$$

4.3 Approximationsalgorithmen

Definition 4.5 (Approximationsalgorithmus)**.**

1. Für ein Maximierungsproblem MAX ist ein Algorithmus ALG ein *c-Approximationsalgorithmus* für MAX, wenn für jede Instanz I von MAX
   a) ALG in *polynomieller Zeit* in der Größe von I eine zulässige Lösung mit Wert ALG liefert
   b) für den Vergleich mit dem zugehörigen Optimalwert $\text{OPT}(I)$ gilt $\text{ALG}(I) \geq c \cdot \text{OPT}(I)$ (dabei ist $c \leq 1$)
2. Entsprechend für Minimierungsproblem wiederum:
   a) ALG liefert in *polynomieller Zeit* in der Größe von I eine zulässige Lösung mit Wert $\text{ALG}(I)$
   b) es gilt für den Vergleich mit dem zugehörigen Optimalwert $\text{OPT}(I)$: $\text{ALG}(I) \leq c \cdot \text{OPT}(I)$ (dabei ist $c \geq 1$)

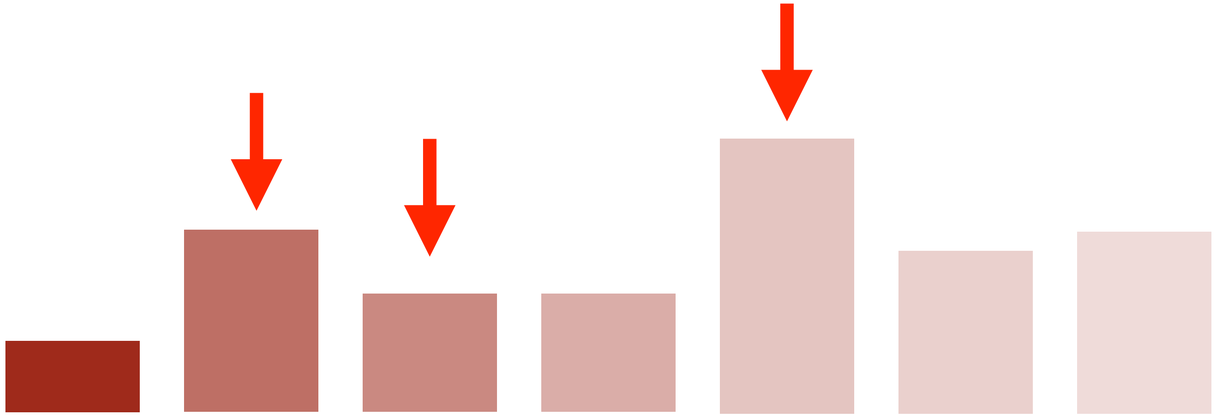
4.4 Bessere Approximation



- Verwende die *k* wertvollsten Objekte aus einer Optimallösung.
- Wende Greedy auf den Rest an.
- Der Fehler ist durch *ein* Objekt beschränkt.

Wie findet man die *k* wertvollsten Objekte?

- Enumeriere alle Teilmengen der Größe k! $\rightarrow O(n^k)$ Teilmengen



Algorithmus 4.6 GREEDY$_k$

Eingabe: $z_1, ..., z_n, Z, p_1, ..., p_n$ [Parameter k fixiert]

Ausgabe: $S \subseteq \{1, ..., n\}$ mit $\sum_{i \in S} z_i \leq Z$ und Wert $G_k := \sum_{i \in S} p_i$

1: $G_k := 0$, $S := \emptyset$

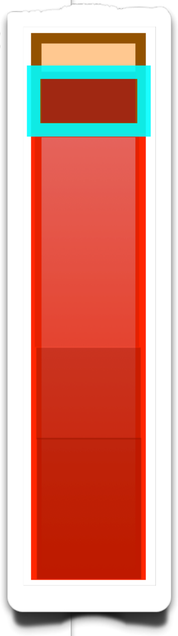
2: **for all** $\bar{S} \subseteq \{1, ..., n\}$ mit $|\bar{S}| \leq k$ **do**

3: **if** $(\sum_{i \in \bar{S}} z_i \leq Z)$ **then**

4: $G_k := max\{G_k, \sum_{i \in \bar{S}} p_i + \text{GREEDY}_0(\{z_i | i \notin \bar{S}\}, Z - \sum_{i \in \bar{S}} z_i \{p_i | i \notin \bar{S}\})\}$;

5: Update S;

6: **return** G_k, S



Satz 4.7. GREEDY$_k$ *ist ein* $(1 - \frac{1}{k+1})$*-Approximationsalgorithmus.*

Definition 4.5 (Approximationsalgorithmus)**.**

1. Für ein Maximierungsproblem MAX ist ein Algorithmus ALG ein *c-Approximationsalgorithmus* für MAX, wenn für jede Instanz I von MAX

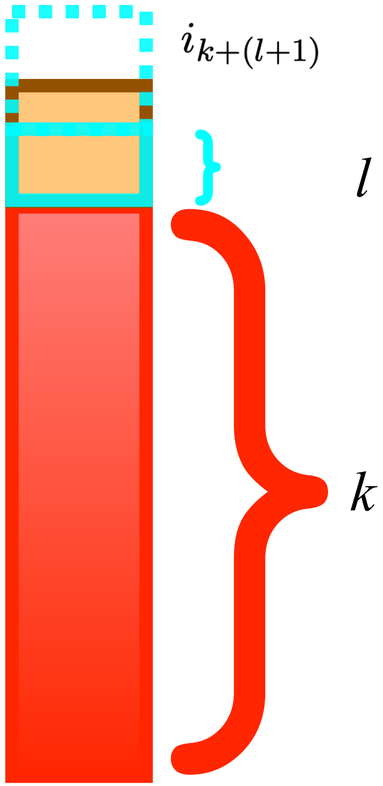
   a) ALG in *polynomieller Zeit* in der Größe von I eine zulässige Lösung mit Wert ALG liefert

   b) für den Vergleich mit dem zugehörigen Optimalwert $\text{OPT}(I)$ gilt: $\text{ALG}(I) \geq c \cdot \text{OPT}(I)$ (dabei ist $c \leq 1$)

Nicht 2^n, sondern n^2

(a) Die Laufzeit ist nicht mehr als $O(n^{k+2})$ - also polynomiell.

Satz 4.7. GREEDY$_k$ *ist ein* $(1 - \frac{1}{k+1})$*-Approximationsalgorithmus.*



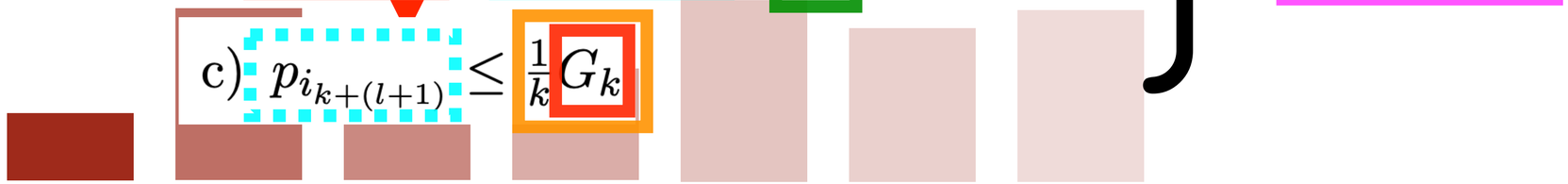
Betrachte OPT mit $S^* \subseteq \{1, \ldots, n\}$.

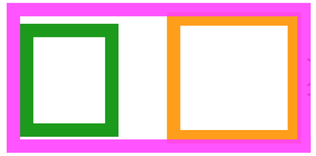
$|S^*| \lneqq k \implies$ OPT wird gefunden.

a) $\sum_{j=1}^{k} p_{i_j} + \sum_{j=1}^{l} p_{i_{k+j}} + p_{i_{k+(l+1)}} \geq OPT$

b) $\sum_{j=1}^{k} p_{i_j} + \sum_{j=1}^{l} p_{i_{k+j}} \leq G_k$

c) $p_{i_{k+(l+1)}} \leq \frac{1}{k} G_k$





Man kann zeigen:

- Für $\textsc{Greedy}_k$ ist der Faktor $(1 - \frac{1}{k+1})$ bestmöglich.
- Wenn man G_0 in Zeile 4 von Algorithmus 4.6 durch G'_0 ersetzt, bekommt man einen Approximationsalgorithmus mit Gütegarantie $(1 - \frac{1}{k+2})$.

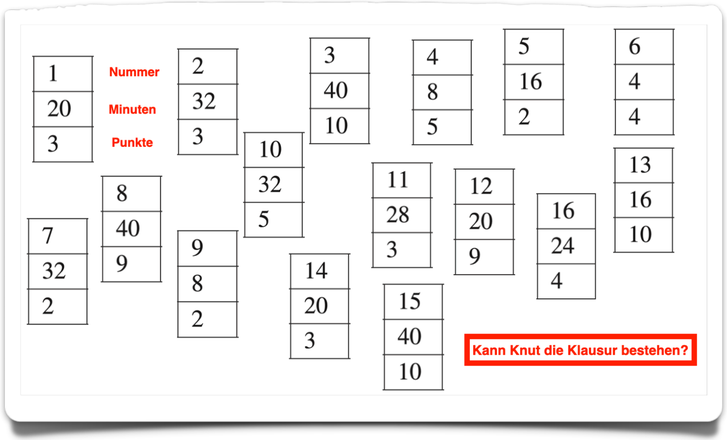
Man sieht: Für jedes feste $\epsilon > 0$ gibt es einen polynomiellen Algorithmus für Knapsack, der eine $(1-\epsilon)$-Approximation liefert. Das motiviert:

Definition 4.8 (PTAS). Ein polynomielles Approximationsschema (Engl.: Polynomialtime approximation scheme. Kurz: PTAS) für ein Optimierungsproblem ist eine Familie von Algorithmen, die für jedes beliebige, aber feste $\epsilon > 0$ einen $(1-\epsilon)$-Approximationsalgorithmus (bzw. $(1+\epsilon)$) liefert.

Korollar 4.9. $\{\text{GREEDY}_k \mid k \in \mathbb{N}\}$ *ist ein PTAS für Knapsack.*

4.4 Ausblicke

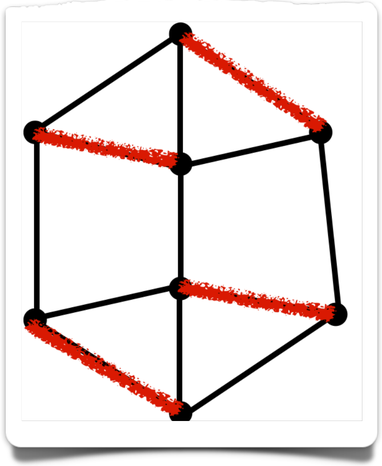
0-1 Knapsack



Max

$$1 - \frac{1}{k+2}$$

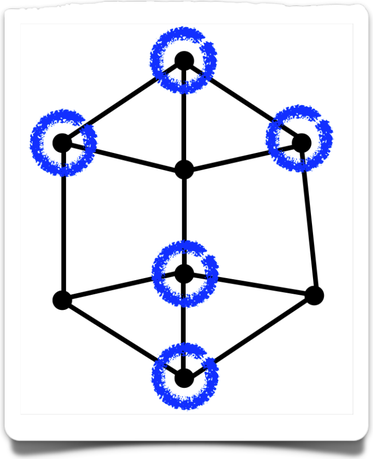
Matching



Max

$$1$$

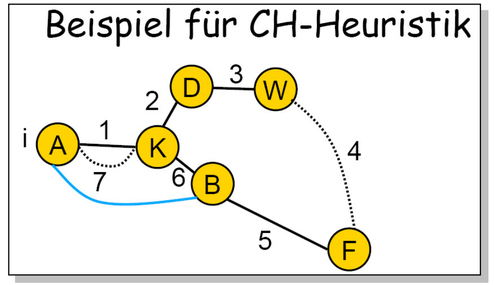
Vertex Cover

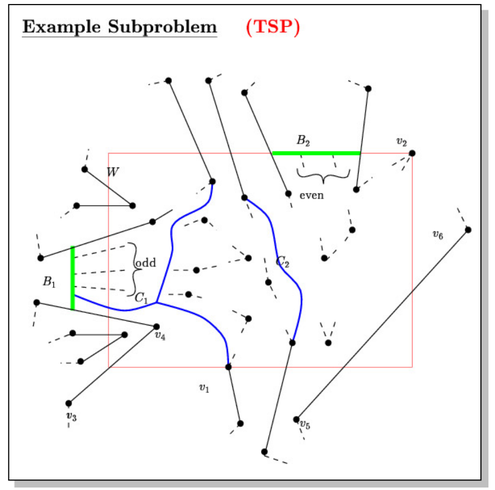


Min

$$2$$

Traveling Salesman

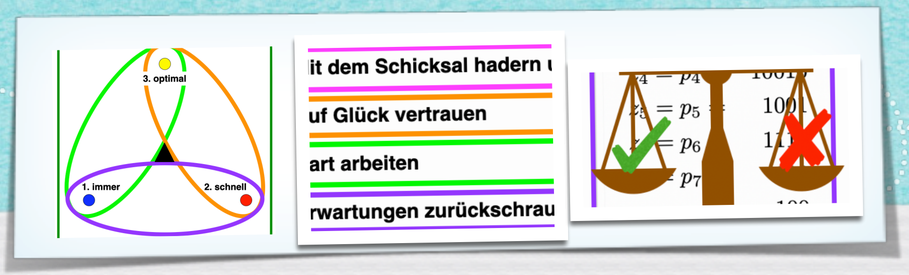




Min

$$\frac{3}{2}$$

$$1+\varepsilon$$



5 Komplexität

Algorithmen und Datenstrukturen 2
Sommer 2026

Prof. Dr. Sándor Fekete

5.1 Die Klasse P: „Perfekte“ Algorithmen

Ziele: Ein „perfekter“ Algorithmus sollte

1.immer: für jede beliebige Instanz

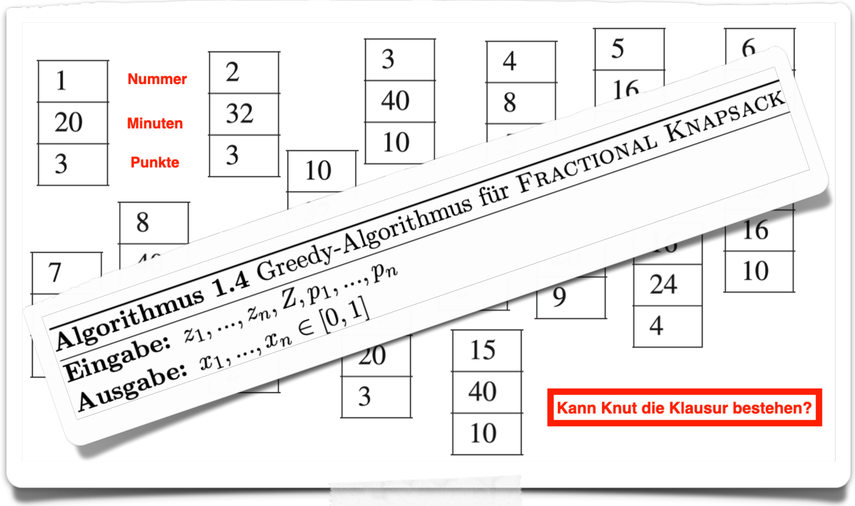
2.schnell: in polynomieller Zeit

3.optimal: eine bestmögliche Lösung

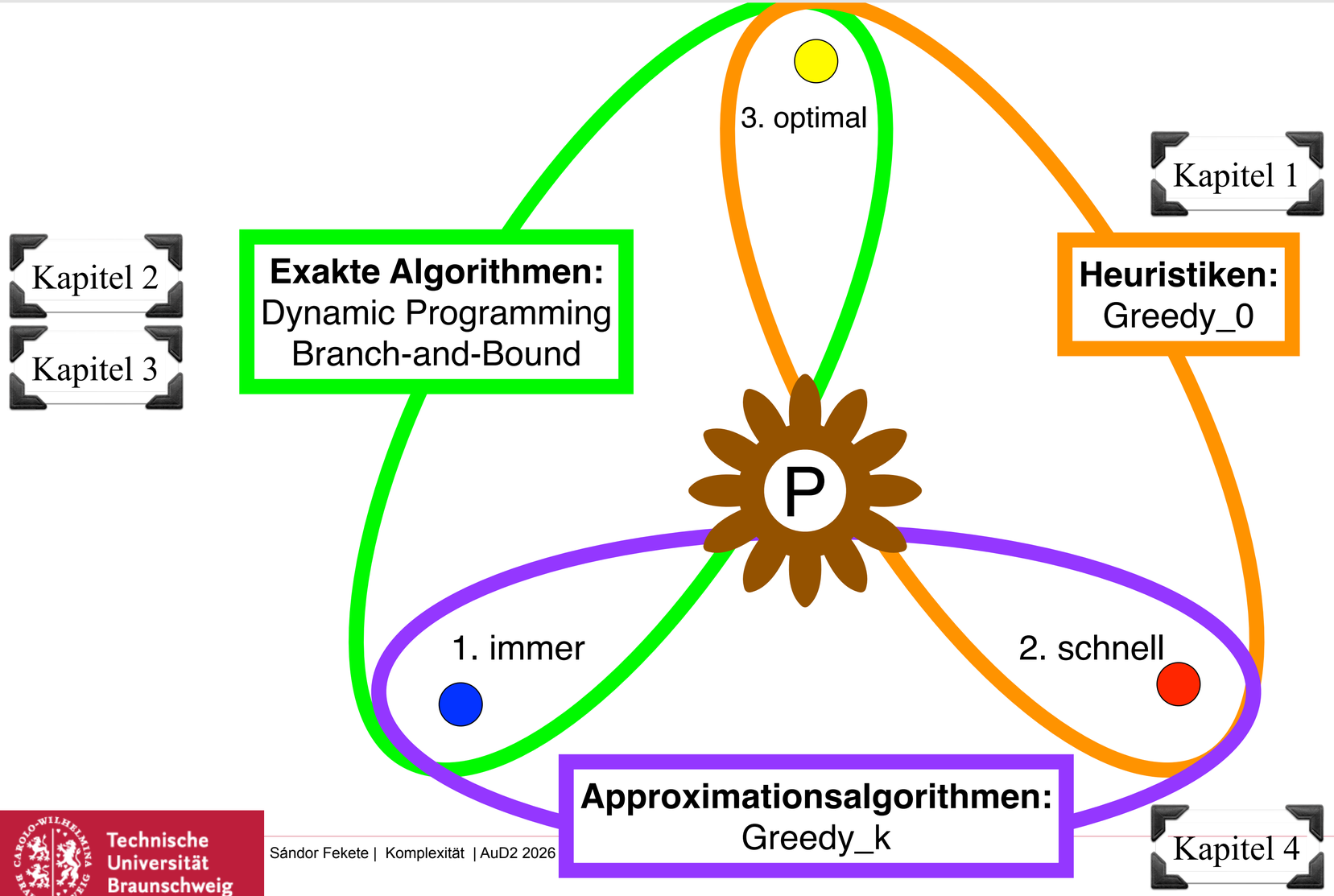
liefern.

Definition 5.1. *Ein algorithmisches Problem* Π *gehört zur Klasse* P*, wenn es für* Π *einen Algorithmus mit polynomieller Laufzeit gibt, der immer eine optimale Lösung findet.*

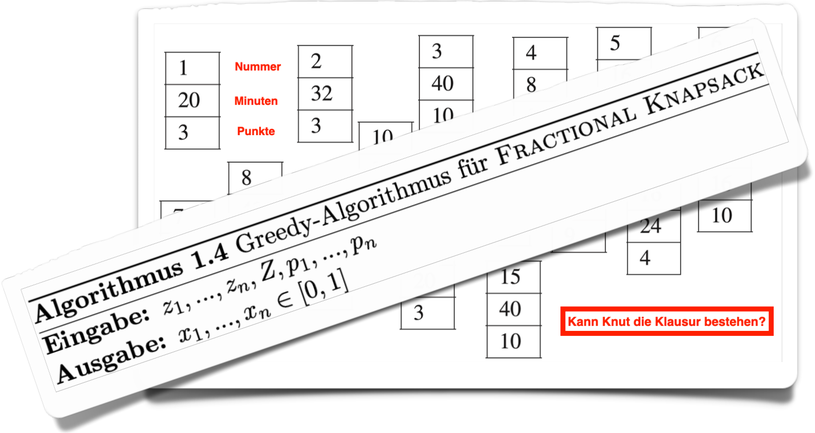
Fractional Knapsack



Achtung:
Das klassifiziert Probleme, nicht Algorithmen!



Fractional Knapsack



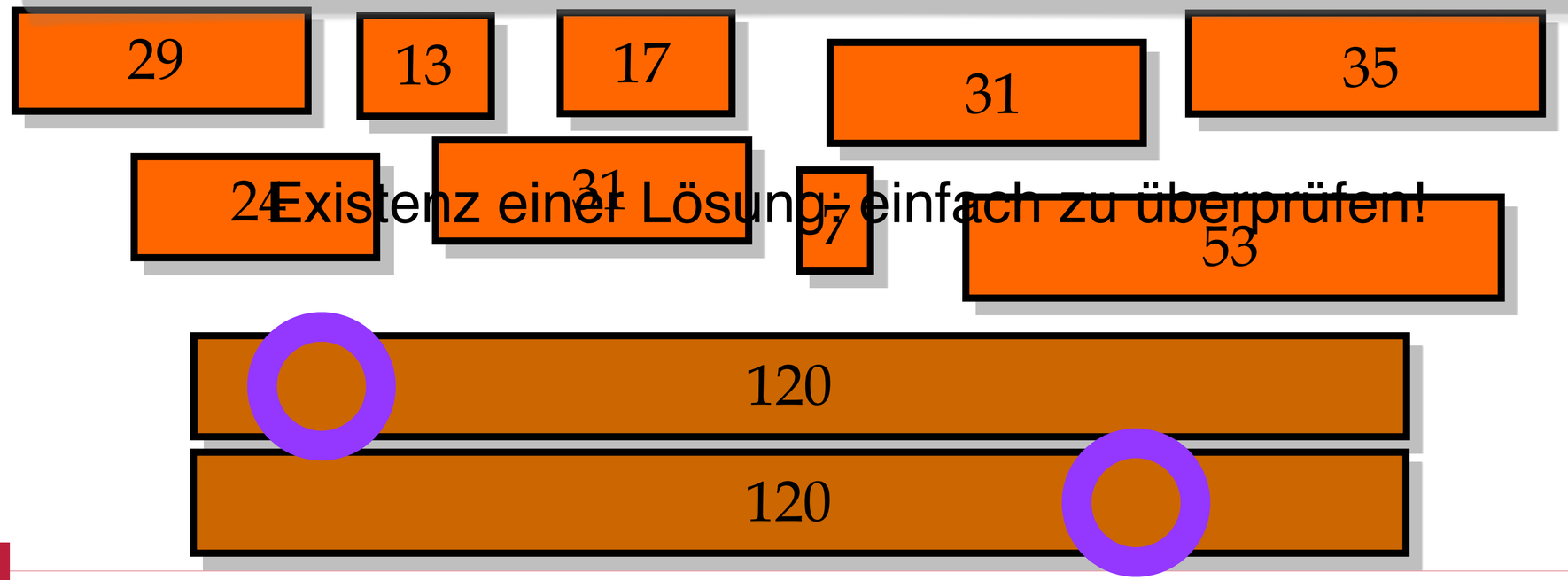
Gehört 0-1 Knapsack zu P?

5.2 Die Klasse NP: „NachPrüfen“

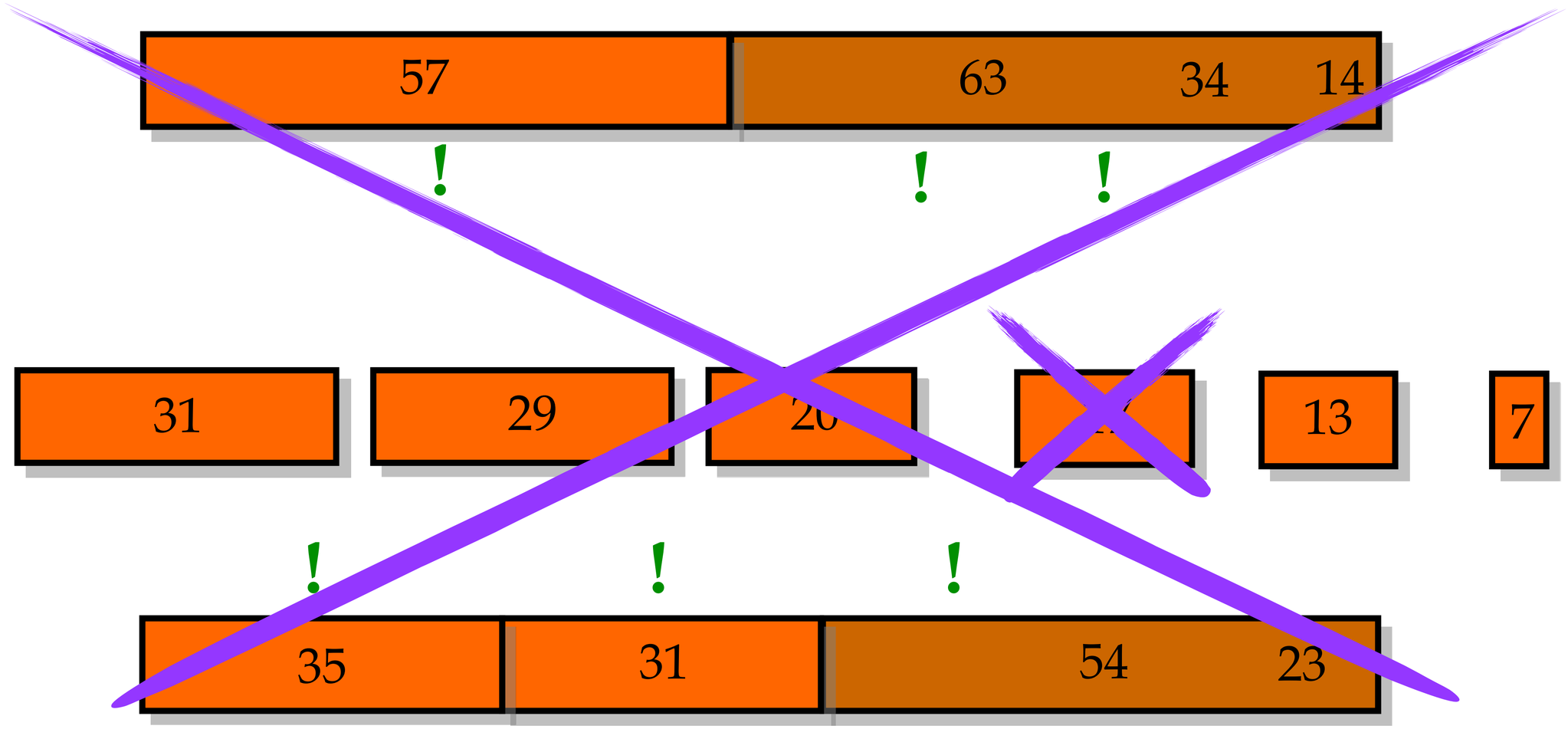
Beispiel 1.10.

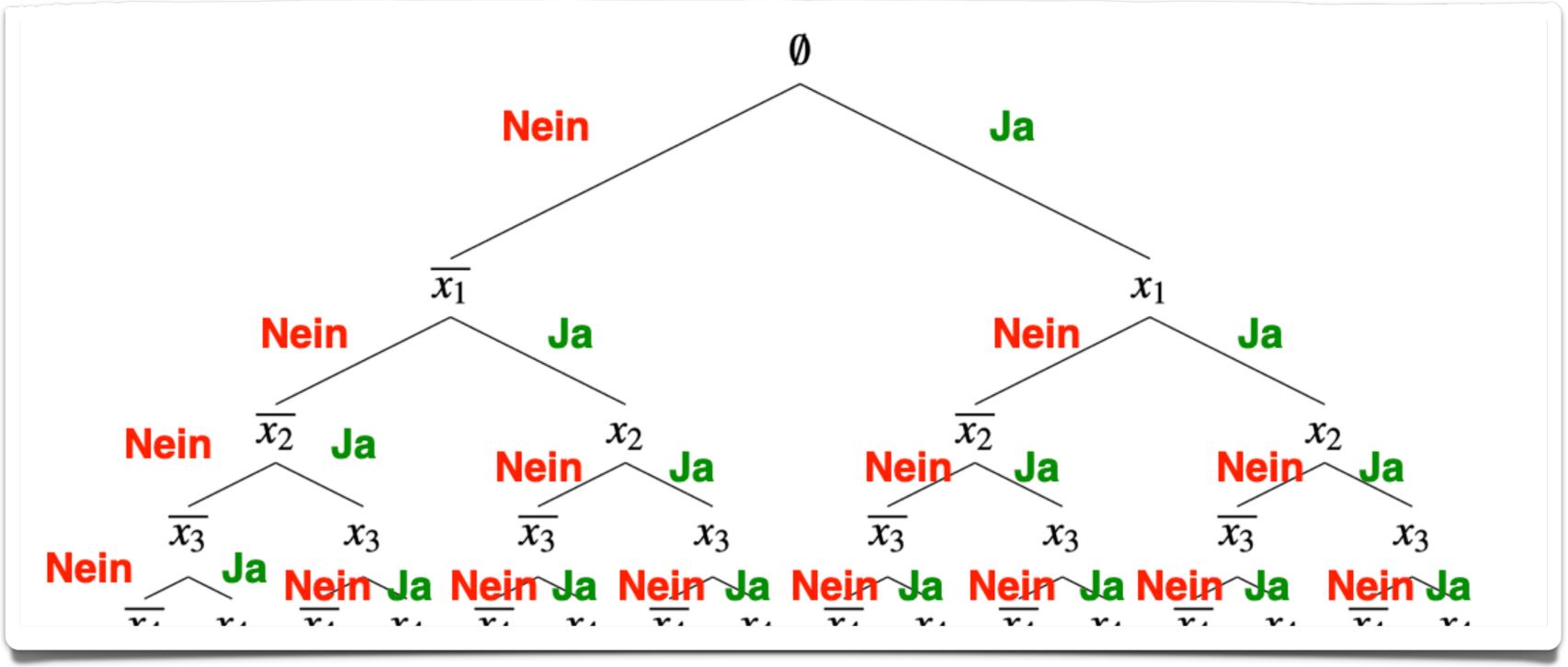
Partition für $\{z_1, ..., z_9\} = \{7, 13, 17, 24, 29, 31, 31, 35, 53\}$

Gesamtsumme: 240

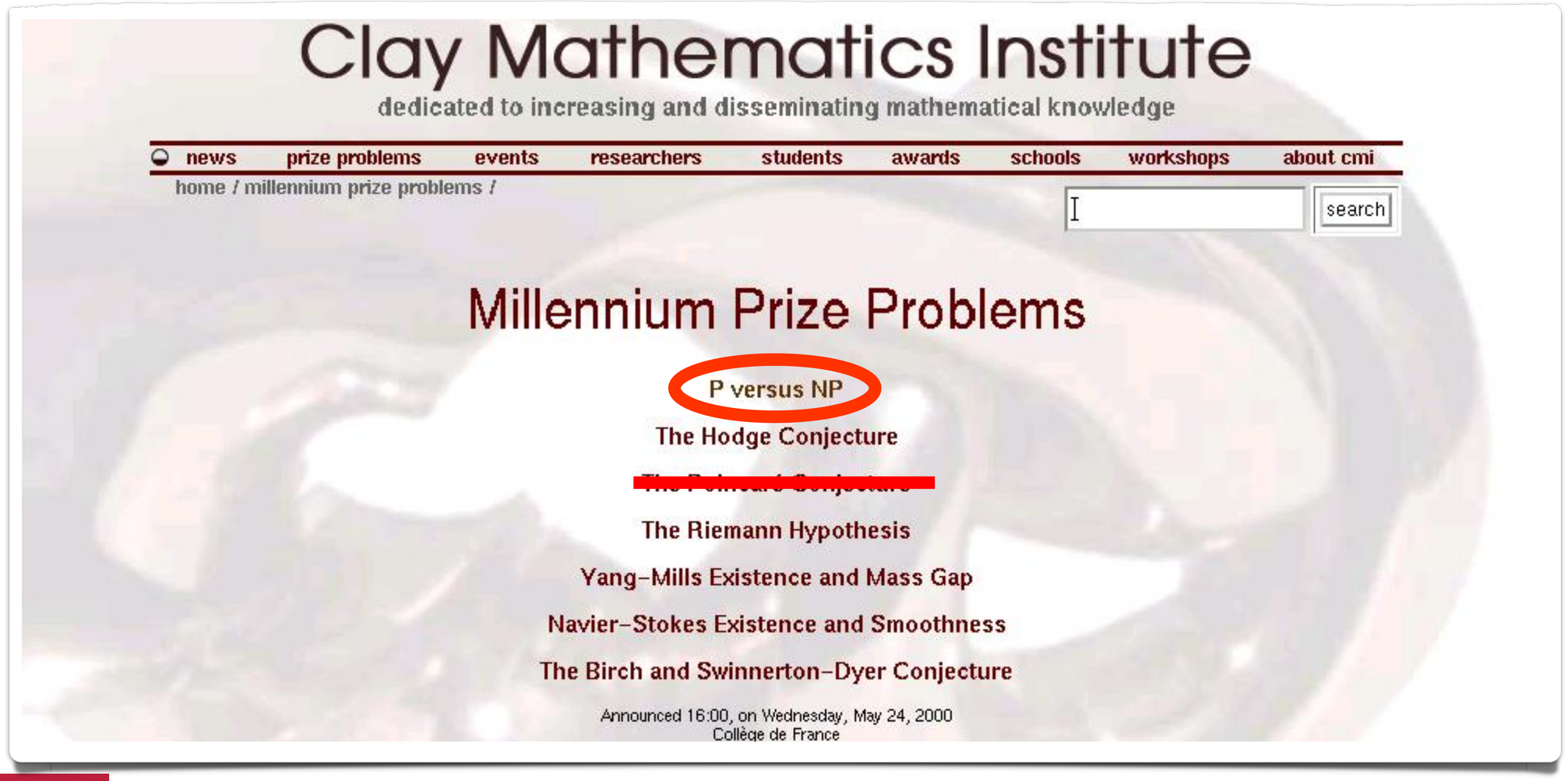


Existenz einer Lösung: einfach zu überprüfen!


57
63
34
14
!
!
!
31
29
13
7
!
!
!
35
31
54
23



- Exponentiell viele Fälle!
- Kann man Arbeit sparen?


Clay Mathematics Institute
dedicated to increasing and disseminating mathematical knowledge
news
prize problems
events
researchers
students
awards
schools
workshops
about cmi
home / millennium prize problems /
search
Millennium Prize Problems
P versus NP
The Hodge Conjecture
The Riemann Hypothesis
Yang–Mills Existence and Mass Gap
Navier–Stokes Existence and Smoothness
The Birch and Swinnerton–Dyer Conjecture
Announced 16:00, on Wednesday, May 24, 2000
Collège de France

- Definition der Klasse NP über schnelle Verifizierbarkeit ist anschaulich.

- Eine formalere Definition (über nichtdeterministische Turingmaschinen) lernt man in Theoretischer Informatik.

- Genau genommen sind Probleme in der Klasse NP Entscheidungsprobleme: Gibt es eine Lösung?

- Die zugehörigen Optimierungsprobleme kann man aber durch eine Reihe von Entscheidungsproblemen lösen:
  - ✓ Gibt es eine Lösung mindestens vom Wert OPT?

5.3 Ein Beispiel mit Logik

Beispiel 5.5.
Wir betrachten die folgende Knapsack-Instanz mit $n = 12$, $z_i = p_i$, $Z = 111444$ und den folgenden Objekten:

$$
\begin{aligned}
z_1 = p_1 &= 100110 \\
z_2 = p_2 &= 100001 \\
z_3 = p_3 &= 10101 \\
z_4 = p_4 &= 10010 \\
z_5 = p_5 &= 1001 \\
z_6 = p_6 &= 1110 \\
z_7 = p_7 &= 200 \\
z_8 = p_8 &= 100 \\
z_9 = p_9 &= 20 \\
z_{10} = p_{10} &= 10 \\
z_{11} = p_{11} &= 2 \\
z_{12} = p_{12} &= 1
\end{aligned}
$$

Gibt es eine Menge $S \subseteq \{1, ...12\}$ mit $\sum_{i \in S} z_i = \sum_{i \in S} p_i = Z$?

$x_1 \vee \overline{x_1}$

$x_2 \vee \overline{x_2}$

$x_3 \vee \overline{x_3}$

$(x_1 \vee x_2 \vee \overline{x_3})$

$(x_1 \vee \overline{x_2} \vee \overline{x_3})$

$(\overline{x_1} \vee x_2 \vee x_3)$

1. Ziffer: Man muss 1 oder 2 auswählen, aber nicht beide.

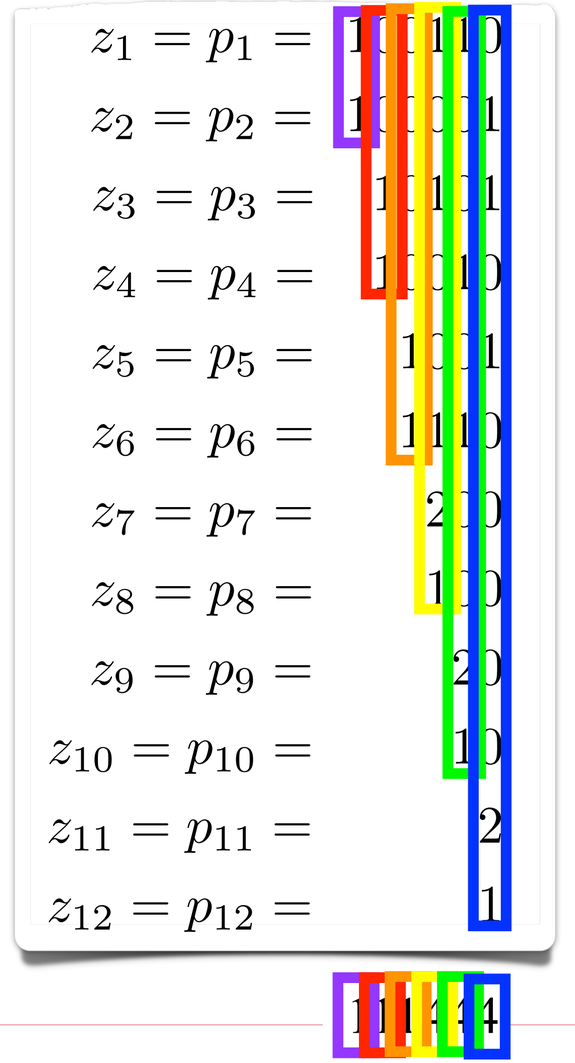
2. Ziffer: Man muss 3 oder 4 auswählen, aber nicht beide.

3. Ziffer: Man muss 5 oder 6 auswählen, aber nicht beide.

4. Ziffer: Man muss 1, 3 oder 6 auswählen, dann kann man mit 7 und 8 den Wert 4 erzeugen.

5. Ziffer: Man muss 1, 4 oder 6 auswählen, dann kann man mit 9 und 10 den Wert 4 erzeugen.

6. Ziffer: Man muss 2, 3 oder 5 auswählen, dann kann man mit 11 und 12 den Wert 4 erzeugen.



$x_1 \vee \overline{x_1}$

$x_2 \vee \overline{x_2}$

$x_3 \vee \overline{x_3}$



$\Leftrightarrow$

$$
\begin{aligned}
z_1 = p_1 &= 100110\\
z_2 = p_2 &= 100001\\
z_3 = p_3 &= 10101\\
z_4 = p_4 &= 10010\\
z_5 = p_5 &= 1001\\
z_6 = p_6 &= 1110\\
z_7 = p_7 &= 200\\
z_8 = p_8 &= 100\\
z_9 = p_9 &= 20\\
z_{10} = p_{10} &= 10\\
z_{11} = p_{11} &= 2\\
z_{12} = p_{12} &= 1
\end{aligned}
$$

111444

$(x_1 \vee x_2 \vee \overline{x_3})$

$(x_1 \vee \overline{x_2} \vee \overline{x_3})$

$(\overline{x_1} \vee x_2 \vee \overline{x_3})$

Konkret: Jede Lösung der logischen Formel entspricht einer Lösung der Instanz SUBSET SUM — und umgekehrt.

Allgemein: Für jede logische Formel dieser Art lässt sich schnell eine äquivalente Instanz von SUBSET SUM konstruieren.

Also: Wenn wir einen „perfekten“ Algorithmus für SUBSET SUM haben, dann können wir auch schnell entscheiden, ob eine logische Formel lösbar ist.

5.4 3SAT

Definition 5.6. (3-Satisfiability (3SAT))

Gegeben: Eine Boolesche Formel, zusammengesetzt aus:

- n Boolesche Variablen $x_1, ..., x_n$, aus denen wir Literale ℓ_i der Form x_k oder $\bar{x}_k$ bilden können.
- m Klauseln, jede zusammengesetzt aus genau drei Literalen $C_j = (\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3})$ mit $1 \leq j \leq m$.

Gesucht: Eine alle m Klauseln erfüllende (engl: satisfying) Wahrheitsbelegung der n Variablen.

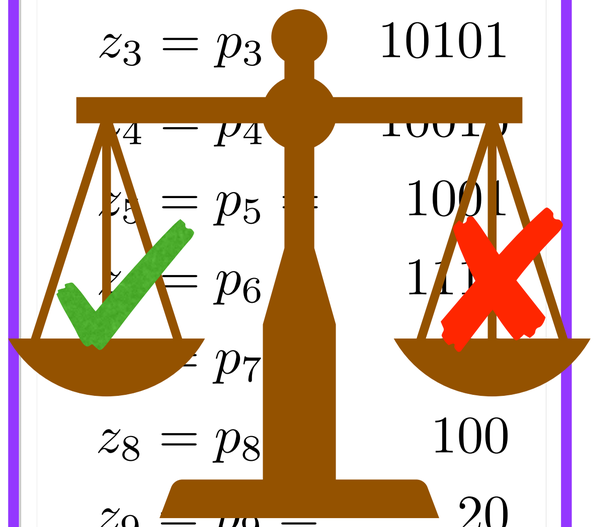
3SAT

$$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3)$$



KNAPSACK

$$\begin{aligned}
z_1 = p_1 &= 110 \\
z_2 = p_2 &= 100001 \\
z_3 = p_3 &= 10101 \\
z_4 = p_4 &= \\
z_5 = p_5 &= 1001 \\
z_6 = p_6 &= \\
z_7 = p_7 &= \\
z_8 = p_8 &= 100 \\
z_9 = p_9 &= 20 \\
z_{10} = p_{10} &= 10 \\
z_{11} = p_{11} &= 2 \\
z_{12} = p_{12} &= 1
\end{aligned}$$



Satz 5.9. *Wenn* 0-1-KNAPSACK $\in P$ *ist, dann ist auch* $3SAT \in P$.

Satz 5.12 (Satz von Cook 1971). *Wenn* $3SAT \in P$, *dann gilt* $P = NP$.

Korollar 5.11. *Wenn* $Knapsack \in P$, *dann gilt* $P = NP$.

5.5 Konsequenzen

Ziele: Ein „perfekter“ Algorithmus liefert

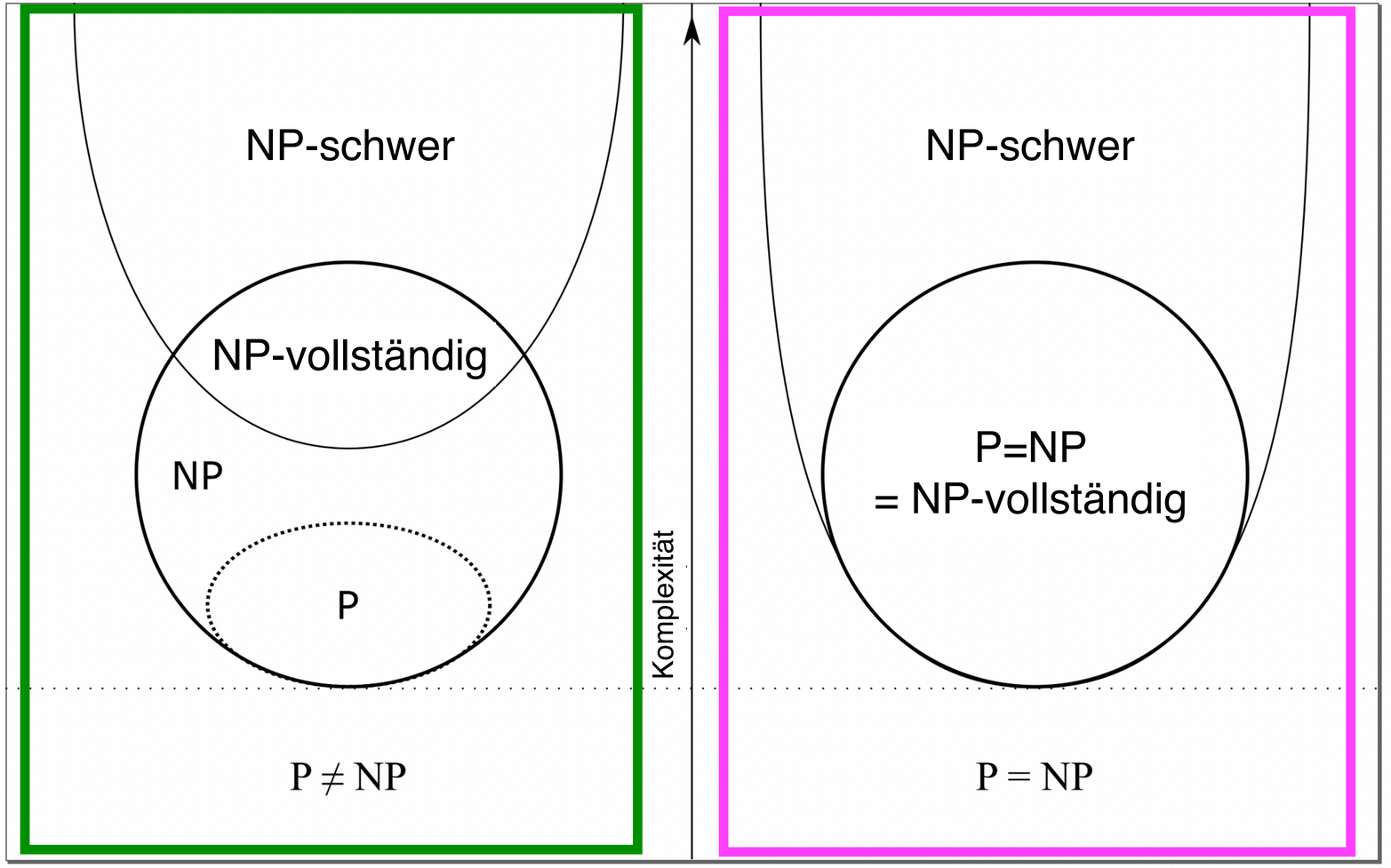
1. **immer**
2. **schnell**
3. eine **optimale** Lösung.

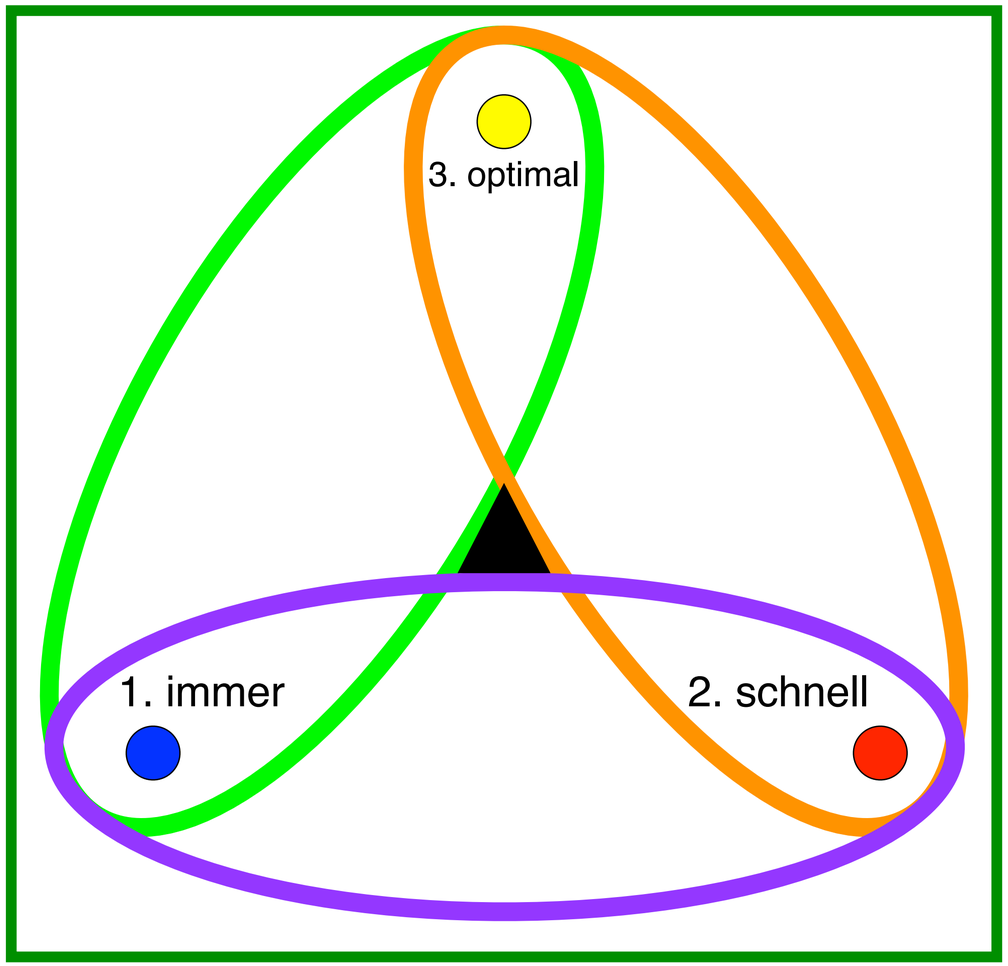
Ziele: Ein „perfekter“ Finanzberater ist ein

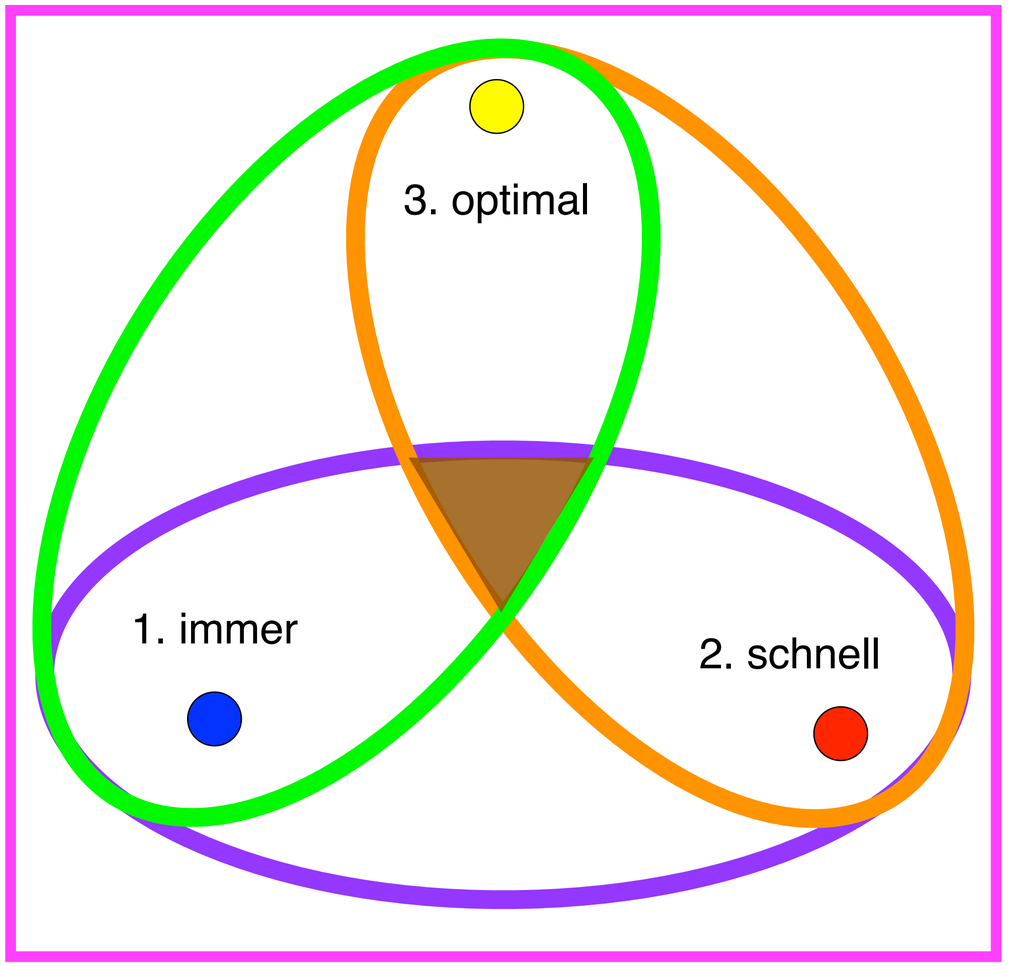
1. **ehrlicher**
2. **intelligenter**
3. **Investmentbanker.**



Technische Universität Braunschweig


NP-schwer
NP-vollständig
NP
P
Komplexität
NP-schwer
P=NP
= NP-vollständig
P ≠ NP
P = NP


3. optimal
1. immer
2. schnell


3. optimal
1. immer
2. schnell

Ziel: Algorithmus für ~~NP-schweres~~ Problem

1. immer

2. schnell

3. eine **optimale** Lösung.

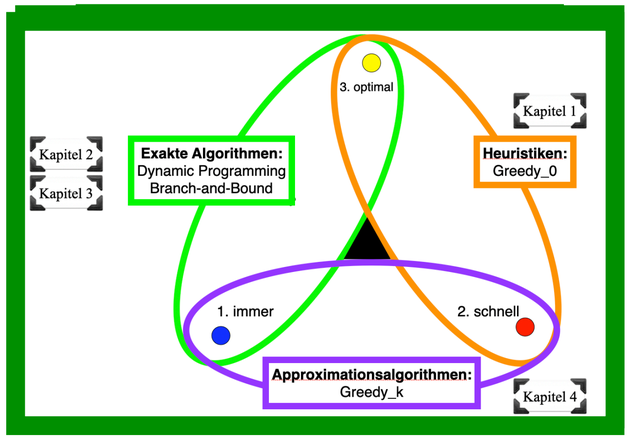
Persönlichkeitsprofile:

(A) Mit dem Schicksal hadern und diskutieren

(B) Auf Glück vertrauen

(C) Hart arbeiten

(D) Erwartungen zurückschrauben



(A) Komplexitätsanalyse: Spezialfall nicht NP-schwer?

(B) Heuristiken: raten und hoffen

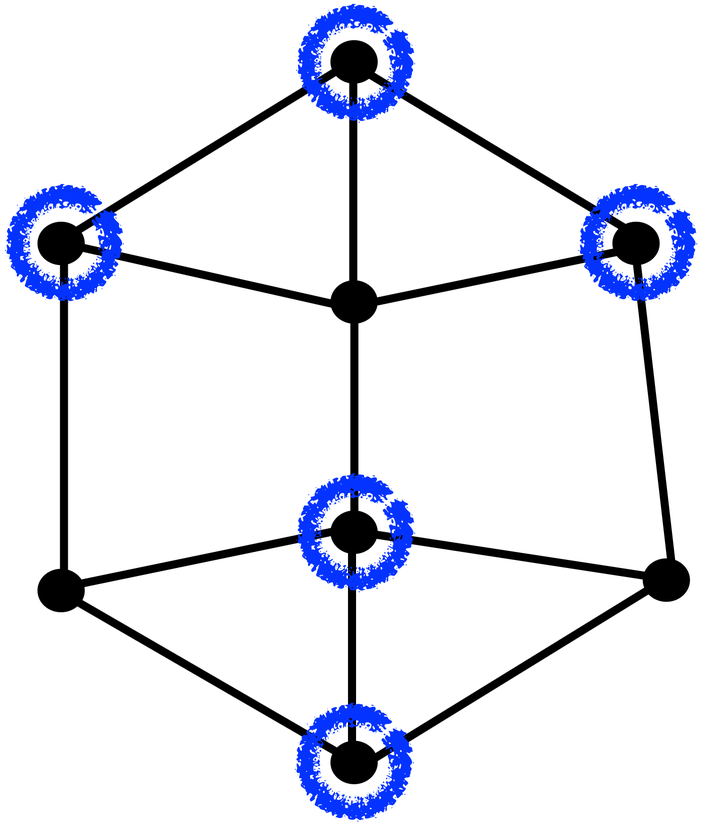
(C) Exakte Algorithmen

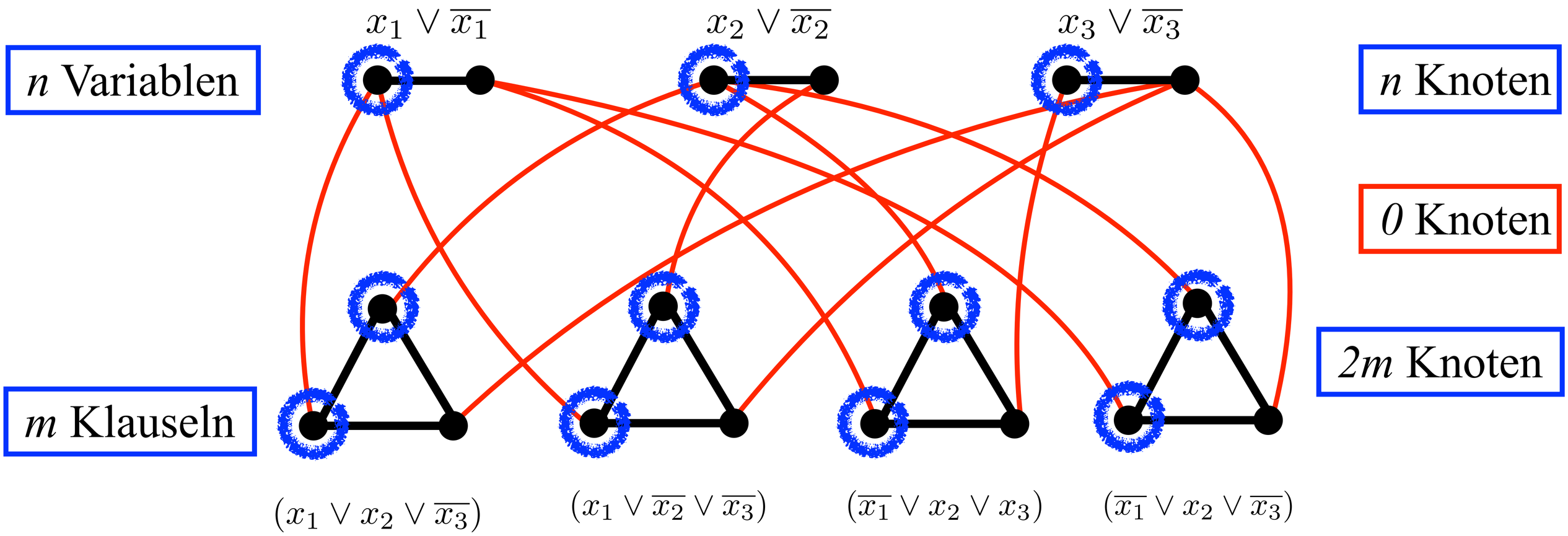
(D) Approximationsalgorithmen: „gut“ statt „optimal“

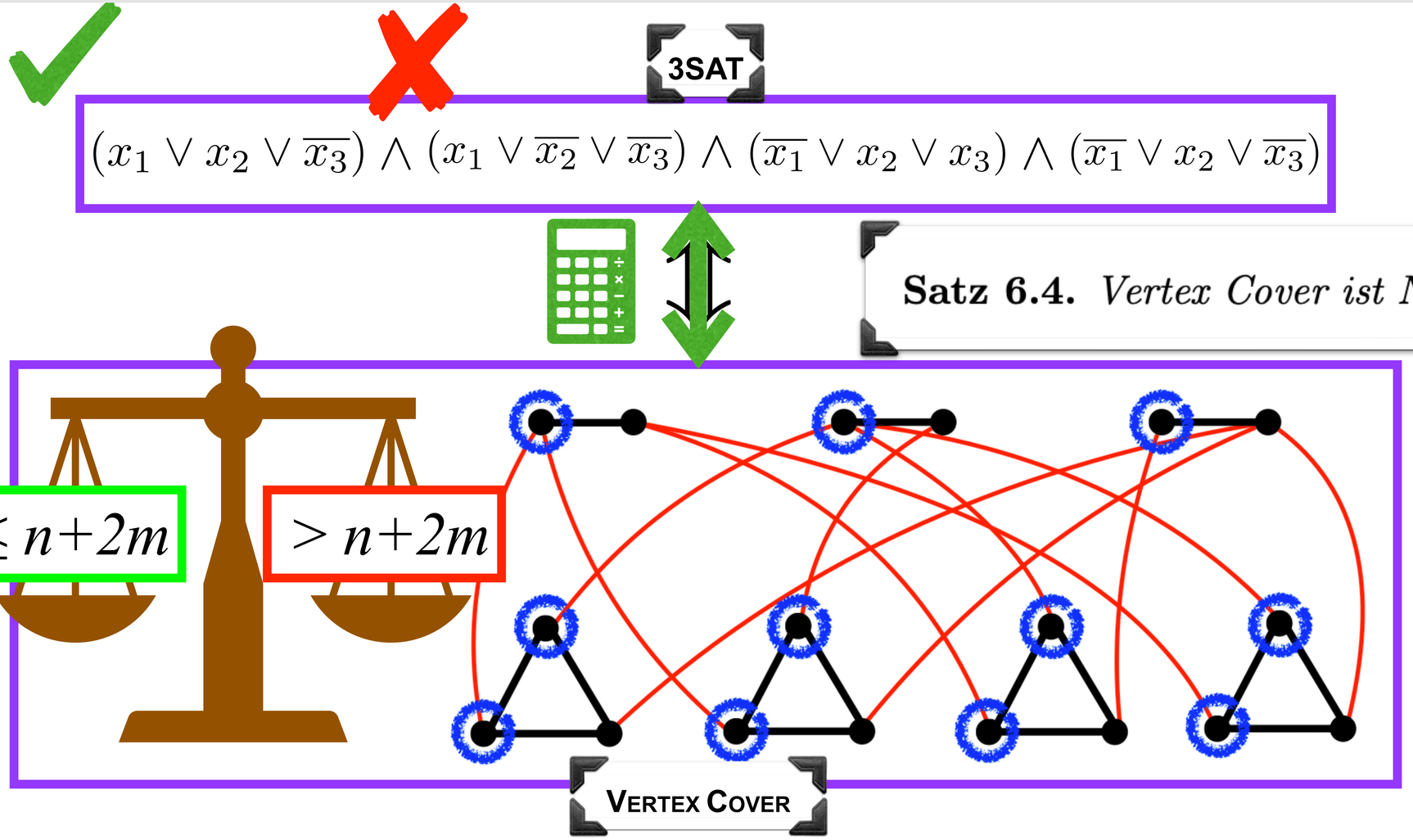
5.6 Ausblick

Gegeben: Ein Graph G=(V,E)

Gesucht: Ein kleinstmögliches **Vertex Cover** S in G:
eine möglichst kleine Menge von Knoten,
die alle Kanten überdecken



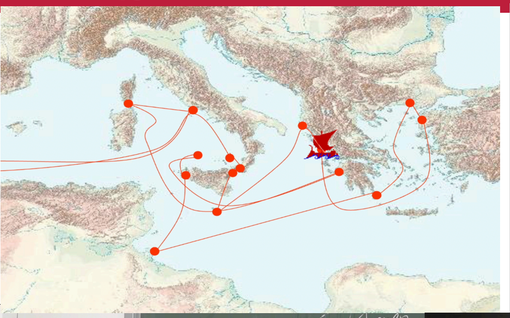

$x_1 \vee \overline{x_1}$
$x_2 \vee \overline{x_2}$
$x_3 \vee \overline{x_3}$
n Variablen
m Klauseln
n Knoten
0 Knoten
$2m$ Knoten
$(x_1 \vee x_2 \vee \overline{x_3})$
$(x_1 \vee \overline{x_2} \vee \overline{x_3})$
$(\overline{x_1} \vee x_2 \vee x_3)$
$(\overline{x_1} \vee x_2 \vee \overline{x_3})$

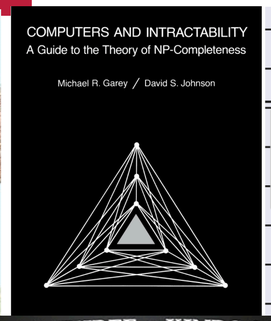

3SAT
$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3})$
Satz 6.4. Vertex Cover ist NP-schwer.
$\leq n+2m$
$> n+2m$
VERTEX COVER

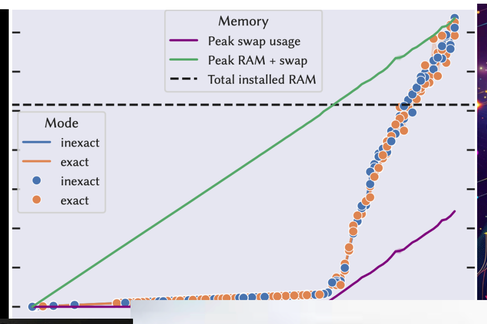
Optimierungsmethoden für höherdimensionale Packprobleme

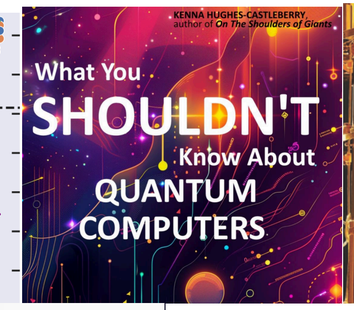
Sándor P. Fekete
TU Braunschweig
s.fekete@tu-bs.de

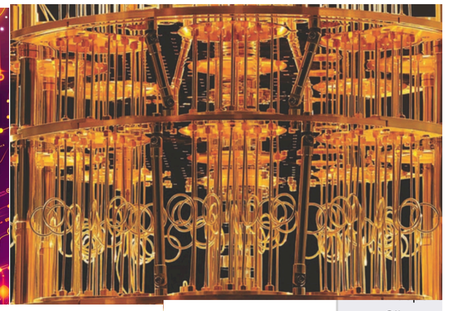
Technische Universität Braunschweig

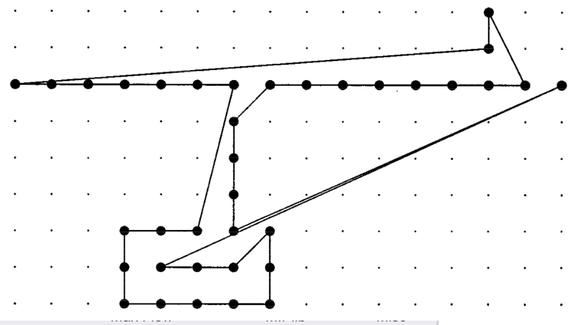


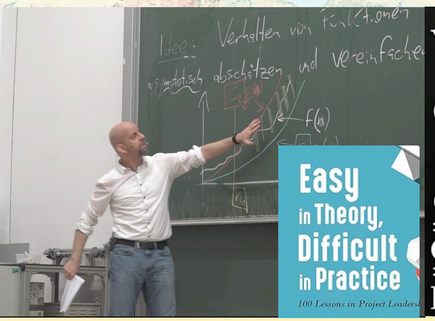




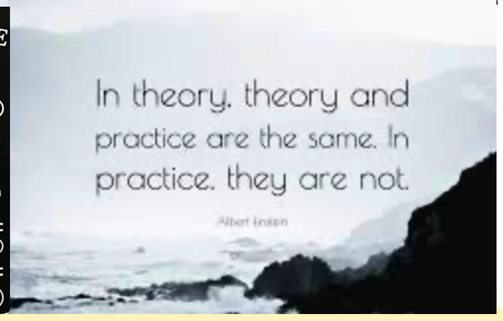


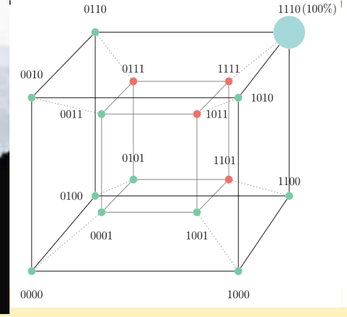


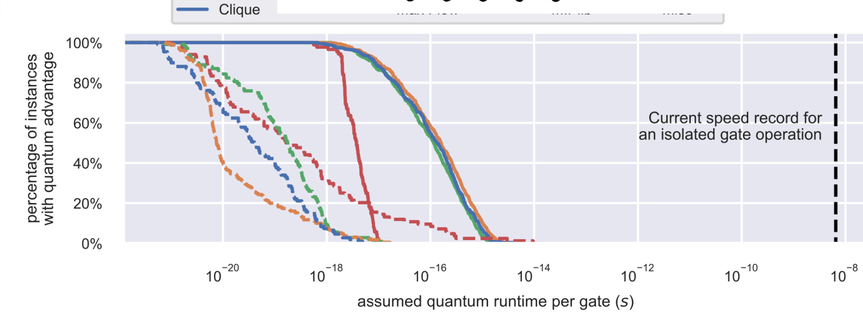












Hard in Theory, Easy in Practice?
Solving Hard Optimization Problems in Geometry and Elsewhere

Sándor P. Fekete
Department of Computer Science
TU Braunschweig

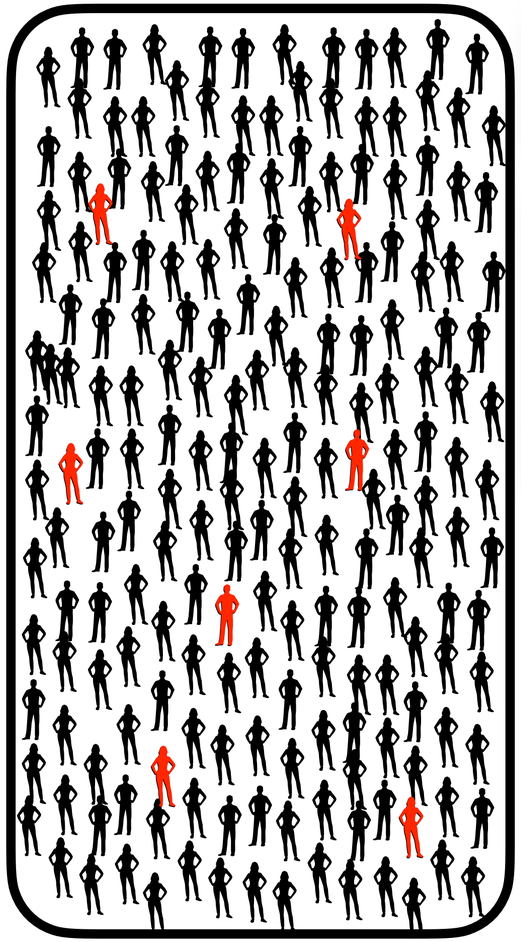


7 Hashing

Algorithmen und Datenstrukturen 2
Sommer 2026

Prof. Dr. Sándor Fekete

7.1 Motivation



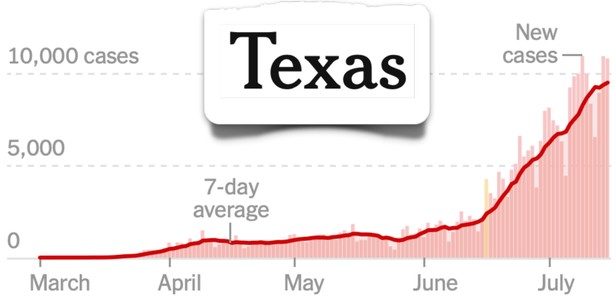
The New York Times

July 13, 2020

Bottleneck for U.S. Coronavirus Response: The Fax Machine

Before public health officials can manage the pandemic, they must deal with a broken data system that sends incomplete results in formats they can't easily use.

By The New York Times Updated July 16, 2020, 12:06 A.M. E.T.



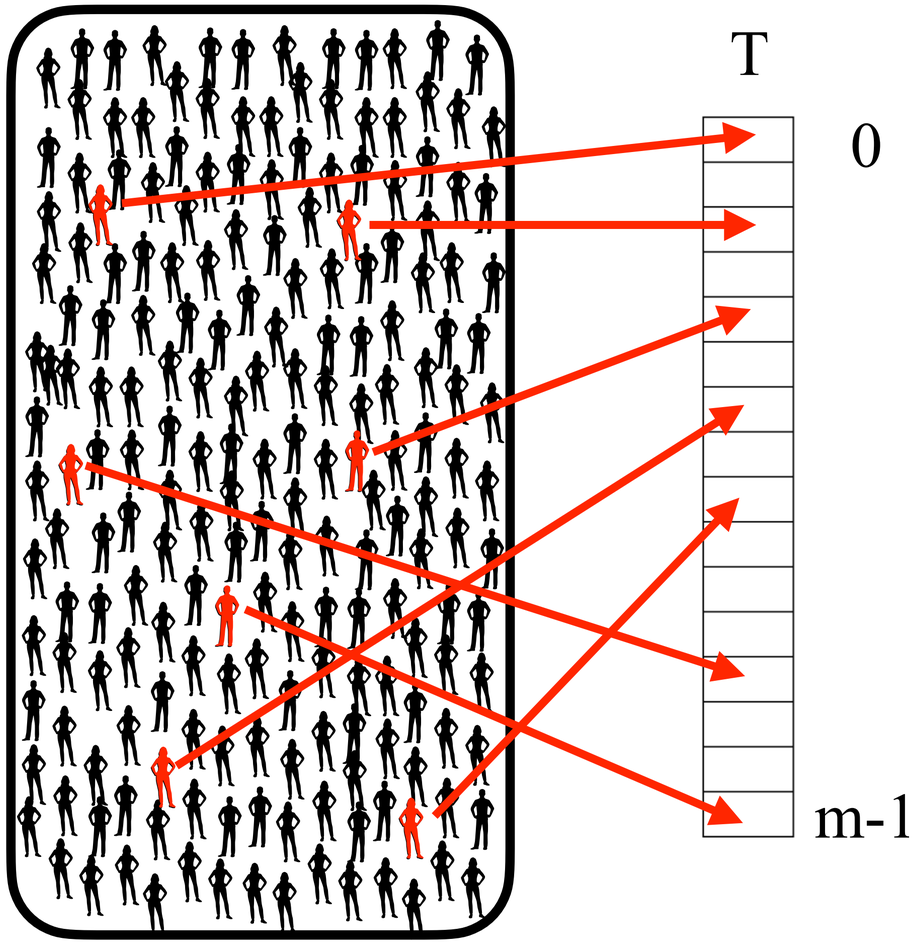
TOTAL CASES
295,423

DEATHS
3,582

Includes confirmed and probable cases where available



Stacks of coronavirus test results at the Harris County Public Health department in Houston. They arrived by mistake and had to be sent to other jurisdictions.



U S

Hashing

- Menge U potentieller Schlüssel sehr groß, aktuelle Schlüsselmenge S jeweils nur kleine Teilmenge des Universums (im allgemeinen S nicht bekannt)
- Idee: durch Berechnung feststellen, wo Datensatz mit Schlüssel x gespeichert
- Abspeicherung der Datensätze in einem Array T mit Indizes $\{0, 1, \ldots, m-1\}$: Hashtabelle
- Hashfunktion h liefert für jeden Schlüssel $x \in U$ eine Adresse in Hashtabelle, d.h. $h : U \to \{0, 1, \ldots, m-1\}$.

7.2 Aufgabenstellung

Aufgabe

Dynamische Verwaltung von Daten, wobei jeder Datensatz eindeutig durch einen Schlüssel charakterisiert ist

Viele Anwendungen benötigen nur einfache Daten-Zugriffsmechanismen (dictionary operations):

- Suche nach Datensatz bei gegebenem Schlüssel x
  search(x)
- Einfügen eines neuen Datensatzes d mit Schlüssel x
  insert(x, d) (abgekürzt insert(x))
- Entfernen eines Datensatzes bei gegebenem Schlüssel x
  delete(x)

Menge potentieller Schlüssel (Universum) kann sehr groß sein!

Sei $n := |S|$.

- Balancierte Suchbäume (AVL-Bäume, B-Bäume): dictionary operations haben Komplexität $O(\log n)$
- Hashing: für alle Operationen mittlere Komplexität $O(1)$

Belegungsfaktor

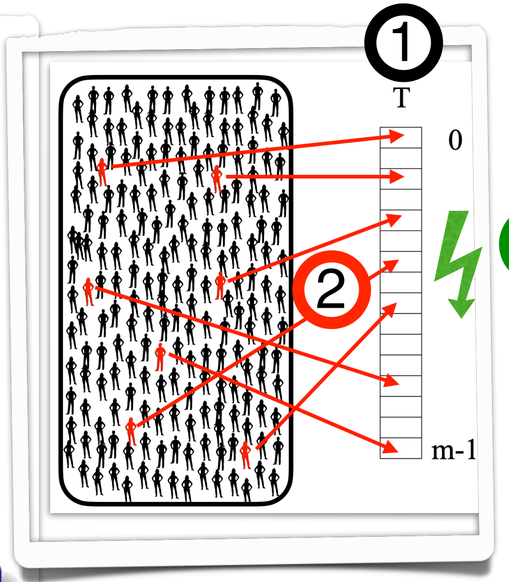
Quotient $\beta := n/m$ heißt Belegungsfaktor oder Auslastungsfaktor einer Hashtabelle der Größe m.

Mittlerer Aufwand für dictionary operations als Funktion in β abschätzbar

$\rightarrow$ Anzahl aktueller Schlüssel geht nur indirekt in Aufwand ein

Anzahl möglicher Schlüssel viel größer als Hashtabelle, also $|U| >> m$

- Hashfunktion muss verschiedene Schlüssel x_1 und x_2 auf gleiche Adresse abbilden.
- x_1 und x_2 beide in aktueller Schlüsselmenge
  $\rightarrow$ Adresskollision

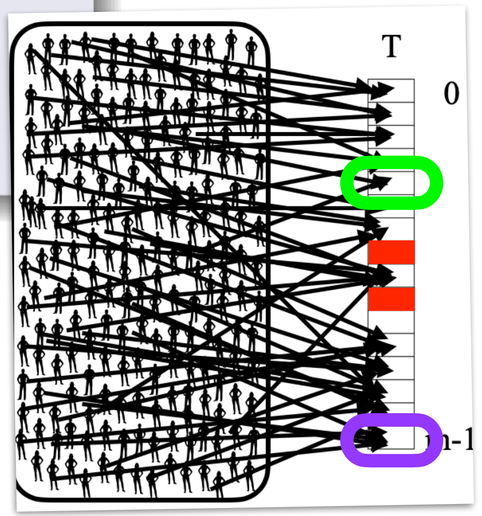


Hashverfahren gegeben durch:

1. eine Hashtabelle,
2. eine Hashfunktion, die Universum der möglichen Schlüssel auf Adressen einer Hashtabelle abbildet,
3. eine Strategie zur Auflösung möglicher Adresskollisionen.

Gute Hashfunktionen sollten:

- surjektiv sein, d.h. den ganzen Wertebereich umfassen,
- die zu speichernden Schlüssel (möglichst) gleichmäßig verteilen, d.h. für alle Speicherplätze i und j sollte gelten $|h^{-1}(i)| \approx |h^{-1}(j)|$,
- effizient berechenbar sein.



7.3 Hashfunktionen

Divisions-Rest-Methode

$$h(x) = x \bmod m := x - \left\lfloor \frac{x}{m} \right\rfloor \cdot m$$

Beispiel: m=11, S={49, 22, 6, 52, 76, 34, 13, 29}

Hashwerte:

$h(49) = 5$
$h(22) = 0$
$h(6) = 6$
$h(52) = 8$
$h(76) = 10$
$h(34) = 1$
$h(13) = 2$
$h(29) = 7$

Problem: Daten oft nicht gleichverteilt!

Beispiel: Texte in Zahlen übertragen, oft viele Leerzeichen, bestimmte Wörter häufiger etc.

Wichtig: geeignete Wahl von m

- m Zweierpotenz: $x \bmod m$ wählt nur die letzten $\log m$ Bits
- m Primzahl: $x \bmod m$ *beeinflusst* alle Bits

7.4 Kollisionen

Zur Erinnerung:

Im Allgemeinen unvermeidbar, dass Kollisionen auftreten, denn aus $N >> m$ folgt Existenz eines Speicherplatzes i mit $|h^{-1}(i)| \geq N/m$.

Frage:

Sei $n := |S|$. Wie wahrscheinlich sind Kollisionen bei $n << m$?

Geburtstagsparadoxon

Bei wie vielen (zufällig gewählten) Personen ist es *wahrscheinlich*, dass hiervon zwei am selben Datum (Tag und Monat) Geburtstag haben?

Annahme:

- Daten unabhängig
- Prob($h(x) = j$)=$1/m$

Prob(i-tes Datum kollidiert nicht mit den ersten $i-1$ Daten, wenn diese kollisionsfrei sind)= $\frac{m-(i-1)}{m}$

Intuition:

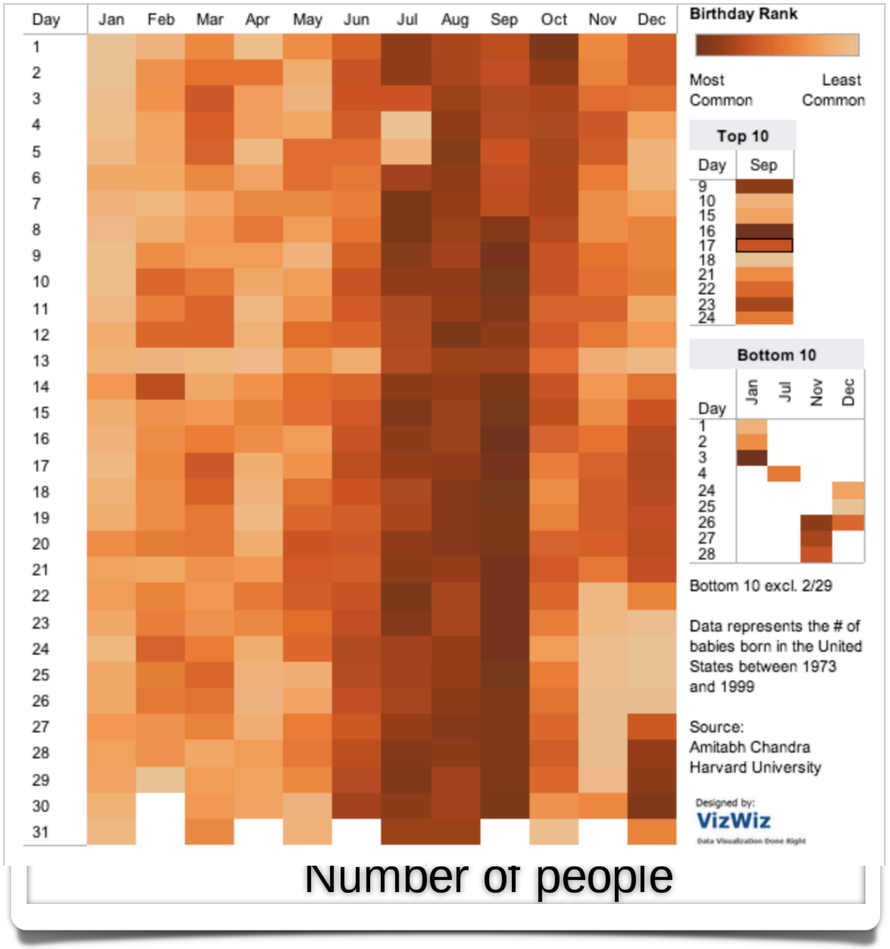
Egal welche Speicherplätze die ersten $i-1$ Daten belegen, $m-i+1$ der m Möglichkeiten sind *gut*.

Prob(n Daten kollisionsfrei) = $\frac{m-1}{m} \cdot \frac{m-2}{m} \ldots \frac{m-n+1}{m}$

Beispiel: $m = 365$

Prob(23 Daten kollisionsfrei) ≈ 0.49

Prob(50 Daten kollisionsfrei) ≈ 0.03



Prob($2m^{1/2}$ Daten kollisionsfrei) =

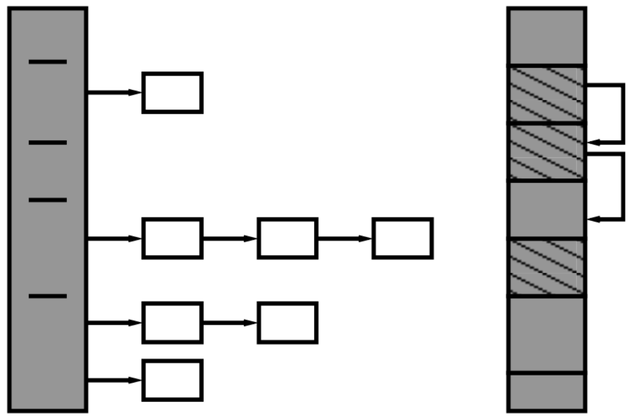
$$\underbrace{\frac{m-1}{m}}_{1} \cdots \underbrace{\frac{m-m^{1/2}}{m} \cdots \frac{m-2m^{1/2}+1}{m}}_{\left(\frac{m-m^{1/2}}{m}\right)^{m^{1/2}}}$$

$$\leq 1 \cdot \left(\frac{m-m^{1/2}}{m}\right)^{m^{1/2}} = \left(1-\frac{1}{m^{1/2}}\right)^{m^{1/2}} \approx \frac{1}{e}$$

Hashing muss mit Kollisionen leben und benötigt Strategien zur Kollisionsbehandlung!

Verschiedene Arten der Kollisionsbehandlung:

- mittels verketteter Listen (links)
- mittels offener Adressierung (rechts)



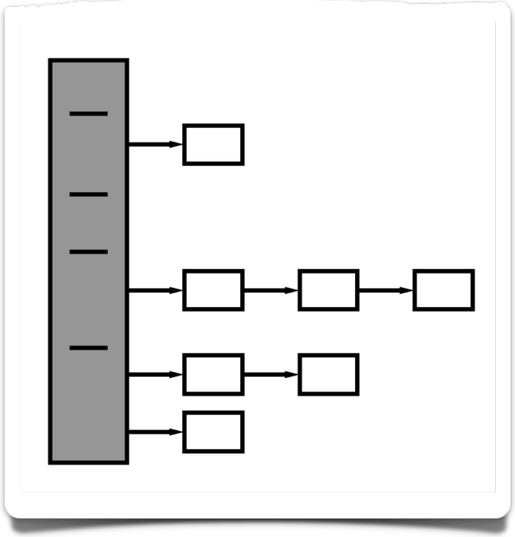
7.5 Verkettung von Überläufern

Realisierung:

Jede Komponente der Hashtabelle enthält Zeiger auf paarweise disjunkte lineare Listen. Die i-te Liste $L(i)$ enthält alle Schlüssel $x \in S$ mit $h(x) = i$.

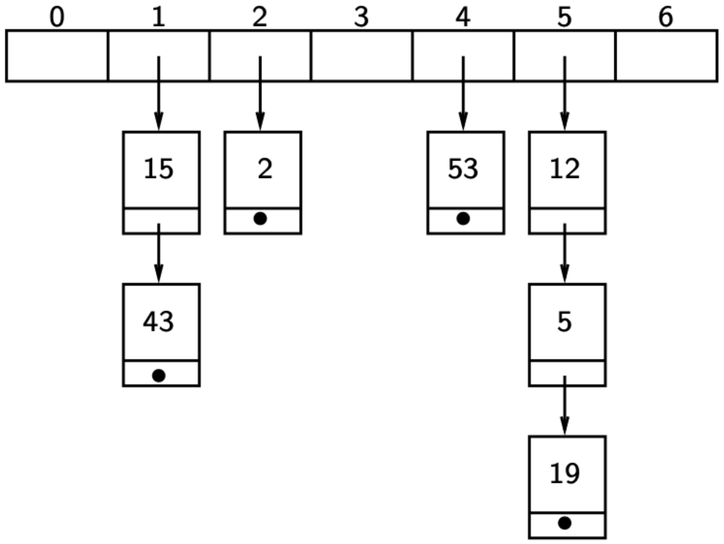
Vorteil: Alle Operationen werden unterstützt und $n > m$ ist möglich. (Für $n >> m$ jedoch Rehashing ratsam) Nachteil: Speicherplatzbedarf für Zeiger

- search(x): Berechne $h(x)$ und suche in Liste $L(h(x))$.
- insert(x) (nach erfolgloser Suche): Berechne $h(x)$ und füge x in Liste $L(h(x))$ ein.
- delete(x) (nach erfolgreicher Suche): Berechne $h(x)$, suche x in Liste $L(h(x))$ und entferne x.



Beispiel: $m = 7$ und $h(x) = x \bmod m$

$$S = \{2, 12, 5, 15, 19, 43, 53\}$$



Bei zufälligen Daten und ideal streuenden Hashfunktion gilt für

$$X_{ij} := \begin{cases} 1 & i\text{-tes Datum kommt in Liste } L(j) \\ 0 & \text{sonst} \end{cases}$$

$\text{Prob}(X_{ij} = 1) = \frac{1}{m}$

$\rightarrow \text{E}(X_{ij}) = 1 \cdot \frac{1}{m} + 0 \cdot \frac{m-1}{m} = \frac{1}{m}$

$X_j = X_{1j} + \cdots + X_{nj}$ zählt Anzahl Daten in Liste $L(j)$.

$\text{E}(X_j) = \text{E}(X_{1j} + \cdots + X_{nj}) = \text{E}(X_{1j}) + \cdots + \text{E}(X_{nj}) = \frac{n}{m}$

- Erfolglose Suche in Liste $L(j)$:

  Inklusive nil-Zeiger durchschnittlich $1 + \frac{n}{m} = 1 + \beta$ Objekte betrachten
  Beispiel: Für $n \approx 0.95 \cdot m$ ist dies ≈ 1.95.

- Erfolgreiche Suche in Liste $L(j)$ der Länge ℓ:

  Jede Position in der Liste hat Wahrscheinlichkeit $1/\ell$, also $\frac{1}{\ell}(1 + 2 + \cdots + \ell) = \frac{\ell+1}{2}$.
  Durchschnittliche Listenlänge hier: $1 + \frac{n-1}{m}$
  (Liste enthält sicher das gesuchte Datum, und die anderen $n - 1$ Daten sind zufällig verteilt.)
  Also erwartete Suchdauer $\frac{1}{2}(1 + \frac{n-1}{m} + 1) = 1 + \frac{n-1}{2m} \approx 1 + \frac{\beta}{2}$
  Beispiel: Für $n \approx 0.95 \cdot m$ ist dies ≈ 1.475.

7.6 Offene Adressierung

7.6 Offene Adressierung

Zur Erinnerung:

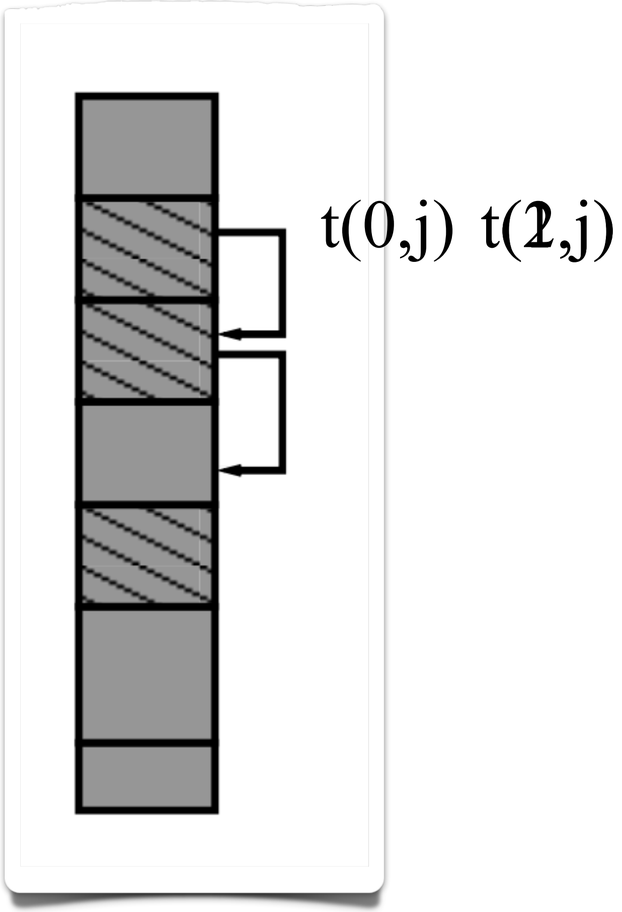
Im Kollisionsfall nach fester Regel alternativen freien Platz in Hashtabelle suchen (Sondierungsfolge).

Voraussetzung: Auswertung von h gilt als eine Operation.

$t(i,j)$:= Position des i-ten Versuchs zum Einfügen von Daten x mit $h(x) = j$

Anforderung an Funktion t:

- auch t in Zeit $O(1)$ berechenbar
- $t(0,j) = j$
- $t(\cdot,j) : \{0,\ldots,m-1\} \to \{0,\ldots,m-1\}$ bijektiv



- search(x):
  - Berechne $j := h(x)$.
  - Suche x an den Positionen $t(0, j), \ldots, t(m-1, j)$.
  - Abbruch, wenn x gefunden oder freie Stelle entdeckt (kein Datum mit Schlüssel x).
- insert(x) nach erfolgloser Suche:
  Freien Platz finden (sonst Overflow) und x dort einfügen.
- delete(x) nach erfolgreicher Suche:
  Das Datum kann nicht einfach entfernt werden, da search frühzeitig Lücken finden würde und eine Suche fälschlicherweise als erfolglos abbrechen könnte.

Problem: Datum kann bei Operation delete(x) nicht ohne weiteres gelöscht werden.

Ausweg: Speicherplatz/Position als besetzt, noch nie besetzt oder wieder frei markieren.

$\longrightarrow$ Suche wird nur an Positionen mit Markierung noch nie besetzt vorzeitig abgebrochen.

Problem: Im Laufe der Zeit keine Position mehr, die mit noch nie besetzt markiert ist.

$\longrightarrow$ Hashing wird ineffizient.

Offenes Hashing nur bei Anwendungen mit search und insert.

- Lineares Sondieren
- Quadratisches Sondieren
- Multiplikatives Sondieren
- Doppeltes Hashing

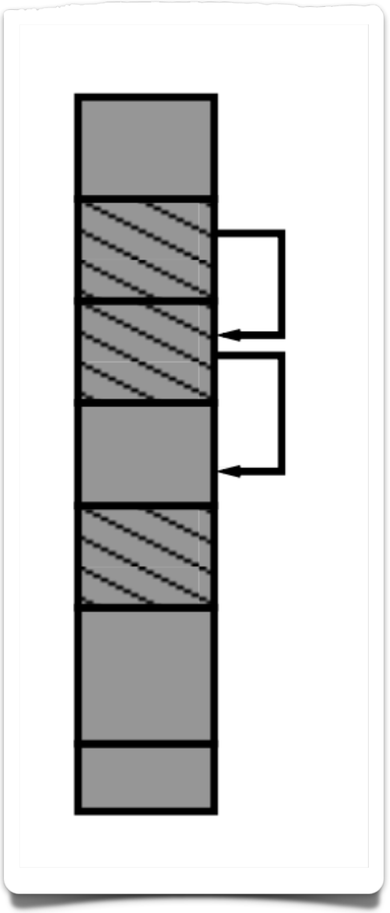
Hilfsmittel bei der Analyse: ideales Hashing

$t(i,j) := (i+j) \bmod m$

Beispiel: $m = 19$ und $j = h(x) = 7$

Sondierungsfolge:
7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 0, 1, 2, 3, 4, 5, 6

Problem: *Clusterbildung*
Tendenz, dass immer längere zusammenhängende, belegte Abschnitte in der Hashtabelle entstehen, sogenannte *Cluster*
$\rightarrow$ erhöhte Suchzeiten



Beispiel: $m = 19$, Positionen $2, 5, 6, 9, 10, 11, 12, 17$ belegt

$h(x)$:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
landet an Position:	0	1	3	3	4	7	7	7	8	13	13	13	13	13	14	15	16	18	18
W.keit:	$\frac{1}{19}$	$\frac{1}{19}$	0	$\frac{2}{19}$	$\frac{1}{19}$	0	0	$\frac{3}{19}$	$\frac{1}{19}$	0	0	0	0	$\frac{5}{19}$	$\frac{1}{19}$	$\frac{1}{19}$	$\frac{1}{19}$	0	$\frac{2}{19}$



Wunsch:

Zu jedem Zeitpunkt haben alle Positionen die gleiche Wahrscheinlichkeit, besetzt zu werden.

Beobachtung:

Das geht numerisch nicht genau, im Beispiel 11 freie Plätze, 19 mögliche Hashwerte, also alle Wahrscheinlichkeiten $k/19$, nicht $k/11$.

Modell des idealen Hashings:

Alle $\binom{m}{n}$ Möglichkeiten, die n besetzten Plätze für m Schlüssel auszuwählen, haben die gleiche Wahrscheinlichkeit.

Lineares Sondieren ist weit vom idealen Hashing entfernt.

$t(i,j) := j + (-1)^{i+1} \cdot \lfloor \frac{i+1}{2} \rfloor^2 \bmod m$

Sondierungsfolge:

$j,\ j+1^2,\ j-1^2,\ j+2^2,\ j-2^2, \ldots,\ j+(\frac{m-1}{2})^2,\ j-(\frac{m-1}{2})^2$

Beispiel: $m = 19$ und $j = h(x) = 7$

Sondierungsfolge:

$$7,\quad 8,\quad 6,\quad 11,\quad 3,\quad 16,\quad \underset{=17,}{-2}\quad \underset{=4,}{23}\quad \underset{=10,}{-9}\quad \underset{=13,}{32}\quad \underset{=1,}{-18}$$

$$\underset{=5,}{43}\quad \underset{=9,}{-29}\quad \underset{=18,}{56}\quad \underset{=15,}{-42}\quad \underset{=14,}{71}\quad \underset{=0,}{-57}\quad \underset{=12,}{88}\quad \underset{=2}{-74}$$

Frage: Ist $t(\cdot, j)$ für alle j und m bijektiv?

Nein, aber immer wenn $m \equiv 3 \mod 4$ und m eine Primzahl ist

(Beweis: Zahlentheorie)

Besser als lineares Sondieren, aber für großes m sind die ersten Werte noch *nah* an j

Hier: $h(x) = x \mod (m-1) + 1$ und damit in $\{1, \ldots, m-1\}$

$t(i,j) := i \cdot j \mod m$, $1 \leq i \leq m-1$

Hashwerte $1, \ldots, m-1$

Beispiel: $m = 19$ und $j = h(x) = 7$

Sondierungsfolge:

$i \cdot j$		7	14	21	28	35	42	49	56	63
$i \cdot j$	mod 19	7	14	2	9	16	4	11	18	6

$i \cdot j$		70	77	84	91	98	105	112	119	126
$i \cdot j$	mod 19	13	1	8	15	3	10	17	5	12

Frage: Ist $t(\cdot, j)$ für alle j und m bijektiv?

Nein, aber immer wenn m Primzahl und $j \neq 0$

Beweis: Durch Widerspruch!

Falls nicht, gibt es $1 \leq i_1 < i_2 \leq m-1$ mit

$$
\begin{aligned}
& i_1 \cdot j \equiv i_2 \cdot j \mod m \\
\Rightarrow \quad & j \cdot (i_2 - i_1) \equiv 0 \mod m \\
\Rightarrow \quad & j \cdot (i_2 - i_1) \text{ ist Vielfaches von } m \\
\Rightarrow \quad & \text{Primfaktorzerlegung von } j \cdot (i_2 - i_1) \text{ muss } m \text{ enthalten}
\end{aligned}
$$

Widerspruch zu $1 \leq j \leq m-1$, $1 \leq i_2 - i_1 \leq m-1$

Sei $h_1(x) \equiv x \mod m$ und $h_2(x) \equiv x \mod (m-2)+1$.

i-te Position für x: $h_1(x) + i \cdot h_2(x) \mod m$, $1 \leq i \leq m-1$

Beispiel: $m = 19$ und $x = 47$

$h_1(47) \equiv 47 \mod 19 = 9$ und $h_2(47) \equiv 47 \mod 17+1 = 14$

Sondierungsfolge:

9, 4, 18, 13, 8, 3, 17, 12, 7, 2, 16, 11, 6, 1, 15, 10, 5, 0, 14

Beobachtung:

$i \cdot h_2(x)$ durchläuft für $1 \leq i \leq m-1$ die Werte $1, \ldots, m-1$ in irgendeiner Reihenfolge, ergänzt wird $0 = 0 \cdot h_2(x)$

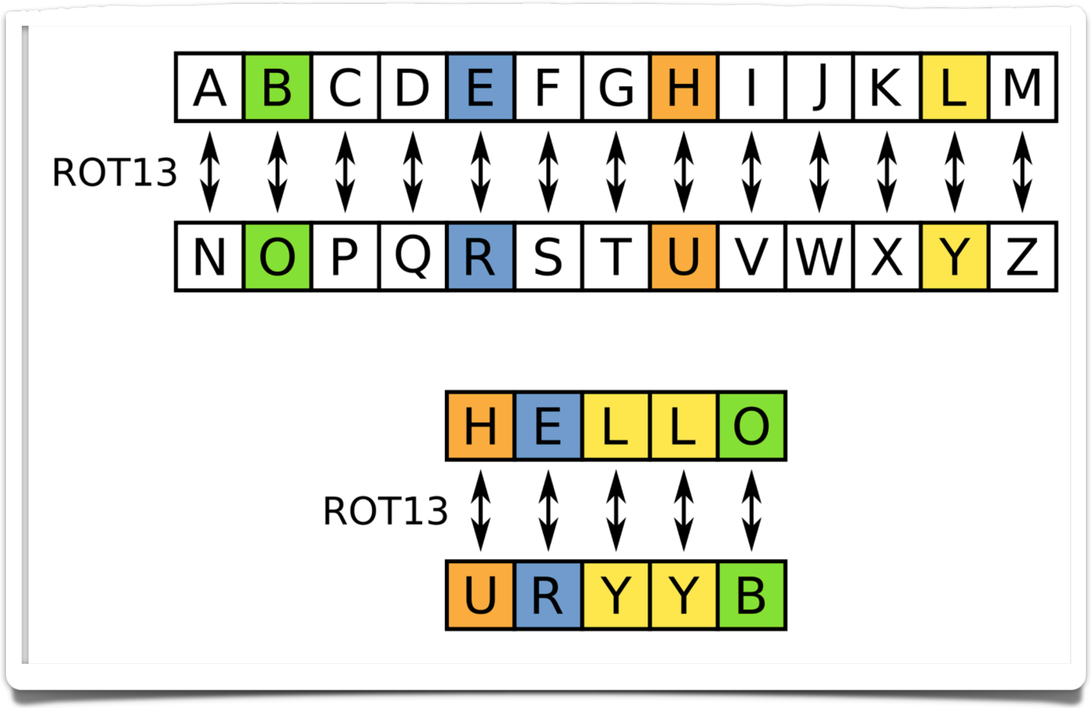
Durch den Summanden $h_1(x)$ wird der Anfang zufällig verschoben.

Doppeltes Hashing kommt dem idealen Hashing am nächsten.

- Auch die beste Hashfunktion kann Kollisionen nicht ganz vermeiden, deshalb sind Hashverfahren im *Worst Case* ineffiziente Realisierungen der Operationen search, insert, delete.
- Im Durchschnitt sind sie jedoch weitaus effizienter als Verfahren, die auf Schlüsselvergleichen basieren.
- Die Anzahl benötigter Schritte zum Suchen, Einfügen und Entfernen hängt (im Durchschnitt) im wesentlichen vom Belegungsfaktor, d.h. dem Verhältnis von Anzahl aktueller Schlüssel zur Größe der Hashtabelle, ab.

7.7 Kryptographie

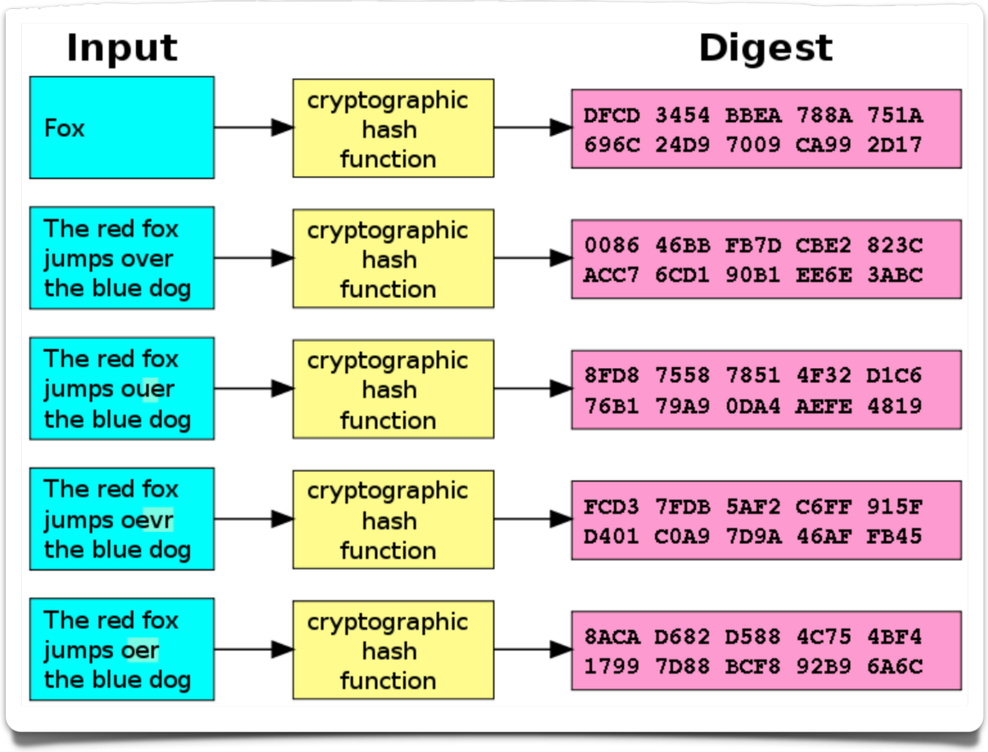
- Text codieren:
  1. Berechtigter Empfänger soll Nachricht entschlüsseln können.
  2. Unberechtigter Empfänger soll Nachricht nicht entschlüsseln können.







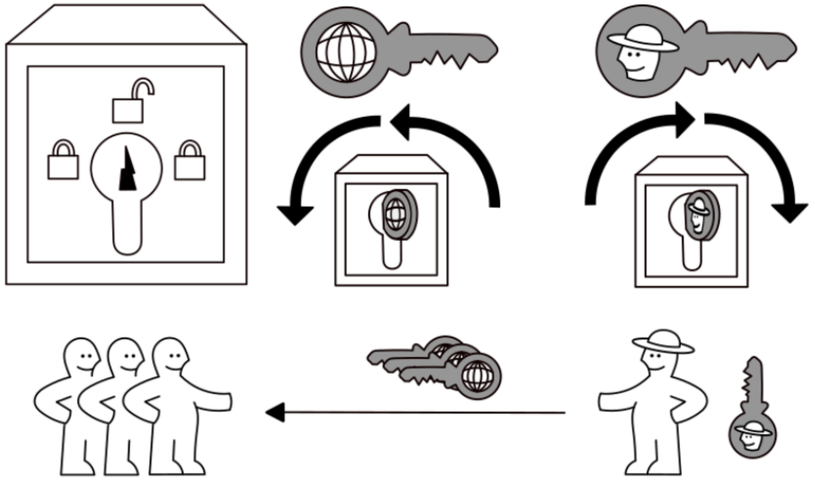
- Deterministisch, d.h. dieselbe Nachricht ergibt immer den selben Wert.
- Hashwert lässt sich schnell berechnen.
- SEHR schwer eine Nachricht mit demselben Hashwert zu generieren.
- SEHR schwer zwei Nachrichten mit demselben Hashwert zu finden.
- Ein kleine Änderung in der Nachricht ändert das Ergebnis so stark, dass man aus den Hashwerten nicht auf ähnliche Nachrichten schließen kann.

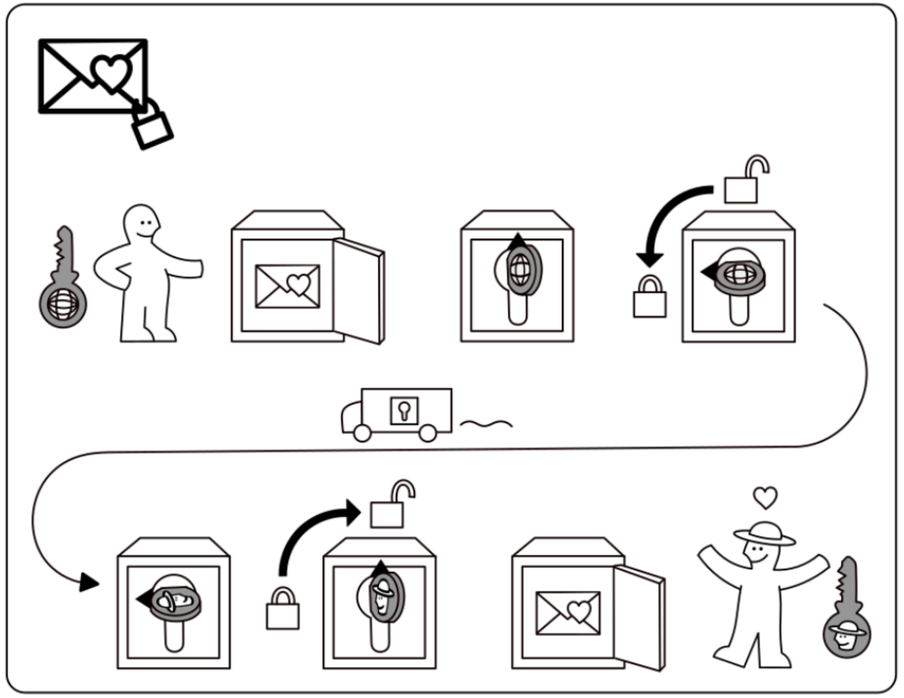


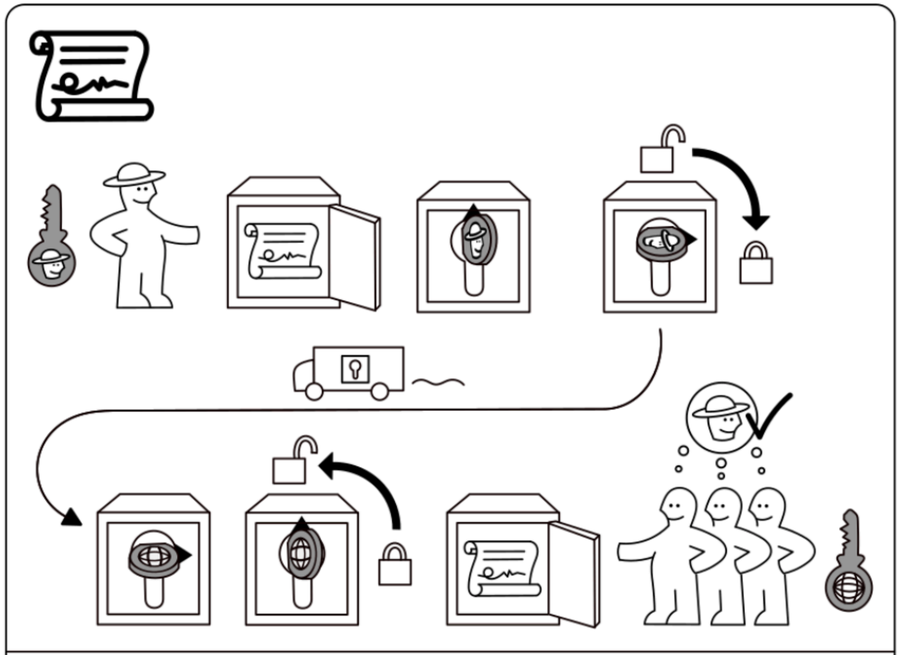
PUBLIK KEY KRYPTO

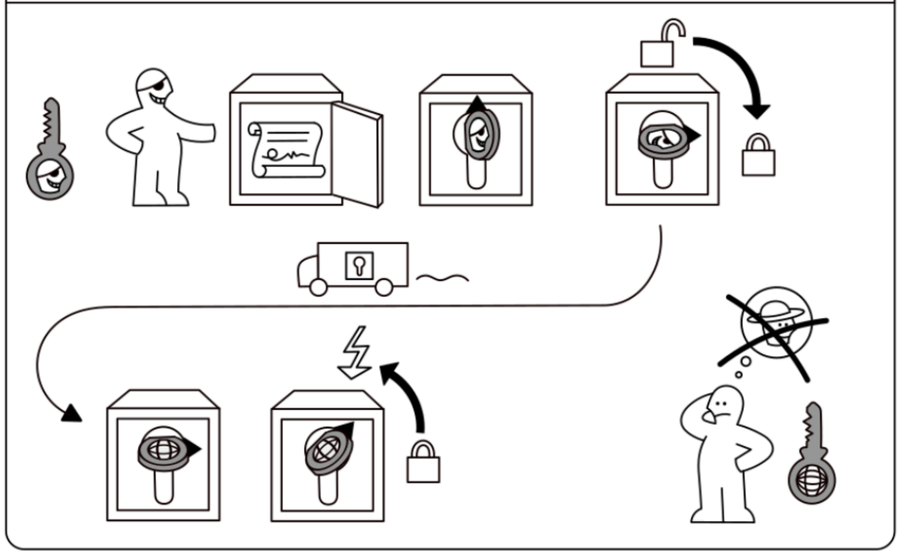
idea-instructions.com/public-key/
v1.2, CC by-nc-sa 4.0

IDEA









Vielen Dank!

s.fekete@tu-bs.de