

# Effiziente Algorithmen

Master of Science

Prof. Dr. Rethmann

Fachbereich Elektrotechnik und Informatik  
Hochschule Niederrhein

WiSe 2022/23

## *Einleitung*

- *Bewertung von Algorithmen*
- *Berechenbarkeitstheorie*
- *Komplexitätstheorie*
- *Exakte Algorithmen für schwere Probleme*

## *Entwurfsmethoden*

- *divide and conquer*
- *dynamic programming*
- *greedy*
- *local search*

## *Sortieren*

- *Quick-Sort – unterschiedliche Varianten*
- *Heap-Sort*
- *Untere Schranke*
- *Counting-/Radix-Sort*
- *Bucket-Sort*

## *Auswahlproblem (Median-Berechnung)*

- *randomisiert*
- *worst-case*

## *Graphalgorithmen*

- *Tiefen- und Breitensuche und deren Anwendungen*
- *Zusammenhangsprobleme*
- *Kürzeste Wege*
- *Minimaler Spannbaum*
- *Netzwerkfluss*
- *Matching*

## *Spezielle Graphklassen*

- *Bäume und Co-Graphen*
- *Chordale Graphen*
- *Vergleichbarkeitsgraphen (comparability graph)*
- *Planare Graphen*

## *Vorrangwarteschlangen*

- *Linksbäume*
- *Binomial-Queues*
- *Fibonacci-Heaps*

## *Suchbäume*

- *AVL-Bäume*
- *Rot-Schwarz-Bäume*
- *B-Bäume*
- *Splay-Bäume*
- *Tries*

## *Amortisierte Laufzeitanalysen*

- *dynamic tables*
- *Selbstanordnende lineare Listen*
- *Splay-Bäume*
- *Fibonacci-Heaps*
- *Union-Find-Datenstruktur*

## *Algorithmen für moderne Hardware*

- *Lokalität der Daten*
- *Externe Datenstrukturen*

## *Algorithmen für geometrische Probleme*

- *Scan-Line-Prinzip*
- *Konvexe Hülle*
- *Polygon-Triangulierung*
- *Voronoi-Diagramme*
- *Datenstrukturen*

## *Randomisierte Algorithmen*

- *Identitätstest*
- *Primzahltest*

- T. Ottmann, P. Widmayer: *Algorithmen und Datenstrukturen*. Springer Spektrum
- T.H. Cormen, C.E. Leiserson, R.L. Rivest: *Introduction to Algorithms*. MIT Press
- Robert Sedgewick: *Algorithms*. Addison-Wesley
- Uwe Schöning: *Algorithmen - kurz gefasst*. Spektrum Akademischer Verlag
- Volker Heun: *Grundlegende Algorithmen*. Vieweg Verlag
- A.V. Aho, J.E. Hopcroft, J.D. Ullman: *Datastructures and Algorithms*. Addison-Wesley



- F. Gurski, I. Rothe, J. Rothe, E. Wanke: *Exakte Algorithmen für schwere Graphenprobleme*. Springer Verlag
- Shimon Even: *Graph Algorithms*. Computer Science Press
- Uwe Schöning: *Theoretische Informatik - kurz gefasst*. Spektrum Akademischer Verlag
- C.H. Papadimitriou: *Computational Complexity*. Addison-Wesley
- Juraj Hromkovič: *Randomisierte Algorithmen*. Vieweg + Teubner Verlag
- Rolf Wanka: *Approximationsalgorithmen*. Teubner B.G.
- I. Gerdes, F. Klawonn, R. Kruse: *Evolutionäre Algorithmen*. Vieweg Verlag

- *Einleitung*
- Entwurfsmethoden
- Sortieren
- Auswahlproblem
- Graphalgorithmen
- Spezielle Graphklassen
- Vorrangwarteschlangen
- Suchbäume
- Amortisierte Laufzeitanalyse
- Algorithmen für moderne Hardware
- Algorithmen für geometrische Probleme
- Randomisierte Algorithmen

## *Einleitung*

- *Bewertung von Algorithmen*
- Berechenbarkeitstheorie
- Komplexitätstheorie
- Exakte Algorithmen für schwere Probleme

## *Welche Eigenschaften interessieren uns?*

- **Korrektheit:** Testen kann die Anwesenheit, aber nicht die Abwesenheit von Fehlern zeigen. (E. Dijkstra)
- **Laufzeit:** Wieviele Operationen werden zur Ausführung des Algorithmus auf einer idealisierten Maschine benötigt?
- **Speicherplatz:** Wieviel Speicherplatz wird benötigt?
- **Kommunikationszeit:** Bei parallelen/verteilten Algorithmen sollen die Prozesse/Threads das eigentliche Problem lösen und möglichst wenig Zeit mit Kommunikation vergeuden.
- **Güte:** Manche Probleme sind so schwer, dass sie sich nicht exakt lösen lassen bzw. eine exakte Lösung zu viel Zeit benötigen würde. Für solche Probleme ist man an möglichst guten Lösungen interessiert.

*Dichtestes Zahlenpaar:* Finde aus  $n$  reellen Zahlen  $x_1, \dots, x_n \in \mathbb{R}$  das Zahlenpaar  $x_i, x_j$ ,  $i \neq j$ , das unter allen Zahlenpaaren den kleinsten Abstand  $d(x_i, x_j) = |x_i - x_j|$  hat.

*Naiver Algorithmus:* Betrachte alle  $\binom{n}{2}$  Paare und bestimme das dichteste Paar.

$$\binom{n}{2} = \frac{n \cdot (n-1)}{2} = \frac{n^2 - n}{2}$$

Laufzeit:  $\Theta(n^2)$

*Aufgabe:* Entwickeln Sie einen effizienten Algorithmus für das Problem. Implementieren Sie Ihren und obigen Algorithmus und vergleichen Sie deren Laufzeiten für verschiedene Eingabegrößen.

*Idee:* Sortiere zunächst die Zahlen. Anschließend stehen die Zahlen, die das dichteste Paar bilden, unmittelbar nebeneinander.

*Algorithmus:*

sort the numbers in non-decreasing order

$min := \infty$

$idx := -1$

**for**  $i := 1, \dots, n-1$  **do**

$diff := x_{i+1} - x_i$

**if**  $diff < min$  **then**

$min := diff$

$idx := i$

output  $x_{idx}$  and  $x_{idx+1}$

*Laufzeit:* Sortieren der Zahlen erfolgt in Zeit  $\mathcal{O}(n \cdot \log(n))$ , das Finden des dichtesten Paares erfolgt in linearer Zeit.

*Definition:* Für eine gegebene Funktion  $g : \mathbb{N}_0 \rightarrow \mathbb{N}_0$  bezeichnet  $\mathcal{O}(g)$  die Menge der Funktionen  $f : \mathbb{N}_0 \rightarrow \mathbb{N}_0$ , die asymptotisch höchstens so stark wachsen wie  $g$ .

→  $g$  ist obere Schranke!

$$\mathcal{O}(g) = \{f \mid \exists c \in \mathbb{N} \exists n_0 \in \mathbb{N} \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}$$

Wir schreiben:

$\mathcal{O}(n)$	für	$\mathcal{O}(g)$	falls $g(n) = n$
$\mathcal{O}(n^k)$	für	$\mathcal{O}(g)$	falls $g(n) = n^k$
$\mathcal{O}(\log(n))$	für	$\mathcal{O}(g)$	falls $g(n) = \log(n)$
$\mathcal{O}(\sqrt{n})$	für	$\mathcal{O}(g)$	falls $g(n) = \sqrt{n}$
$\mathcal{O}(2^n)$	für	$\mathcal{O}(g)$	falls $g(n) = 2^n$

Wir betrachten nur Funktionen  $f : \mathbb{N}_0 \rightarrow \mathbb{N}_0$ , da sowohl die Größe der Eingabe als auch die Anzahl der ausgeführten Operationen eines Algorithmus immer ganzzahlig und nicht-negativ sind.

## Beispiele:

- $37 \cdot n^3 + 2 \cdot n^2 \in \mathcal{O}(n^3)$
- $37 \cdot n^3 + 2 \cdot n^2 \in \mathcal{O}(n^4)$
- $37 \cdot n^3 + 2 \cdot n^2 \in \mathcal{O}(2^n)$
- $37 \cdot n^3 + 2 \cdot n^2 \notin \mathcal{O}(n^2)$
- $3 \cdot n^2 + 42 \cdot n \cdot \log(n) \in \mathcal{O}(n^2)$
- $139 \cdot n + 17 \cdot \sqrt{n} \in \mathcal{O}(n)$
- $139 \cdot n + 17 \cdot \sqrt{n} \notin \mathcal{O}(\log(n))$
- $0,7 \cdot 2^n + 928 \cdot n^4 \in \mathcal{O}(2^n)$
- $c_k \cdot n^k + c_{k-1} \cdot n^{k-1} + \dots + c_0 \in \mathcal{O}(n^k)$  für  $c_0, \dots, c_k$  konstant

Wir suchen natürlich immer die kleinste, obere Schranke!



*Definition:* Für eine gegebene Funktion  $g : \mathbb{N}_0 \rightarrow \mathbb{N}_0$  bezeichnet  $\Omega(g)$  die Menge der Funktionen  $f : \mathbb{N}_0 \rightarrow \mathbb{N}_0$ , die asymptotisch mindestens so stark wachsen wie  $g$ .

→  $g$  ist untere Schranke!

$$\Omega(g) = \{f \mid \exists c \in \mathbb{N} \exists n_0 \in \mathbb{N} \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)\}$$

*Beispiele:*

- $37 \cdot n^3 + 2 \cdot n^2 \in \Omega(n^3)$
- $37 \cdot n^3 + 2 \cdot n^2 \in \Omega(n^2)$
- $37 \cdot n^3 + 2 \cdot n^2 \notin \Omega(n^4)$
- $3 \cdot n^2 + 42 \cdot n \cdot \log(n) \in \Omega(n^2)$
- $139 \cdot n + 17 \cdot \sqrt{n} \in \Omega(n)$
- $c_k \cdot n^k + c_{k-1} \cdot n^{k-1} + \dots + c_0 \in \Omega(n^k)$  für  $c_0, \dots, c_k$  konstant

Wir suchen natürlich immer die größte, untere Schranke!

*Definition:* Für eine gegebene Funktion  $g : \mathbb{N}_0 \rightarrow \mathbb{N}_0$  bezeichnet  $\Theta(g)$  die Menge der Funktionen  $f : \mathbb{N}_0 \rightarrow \mathbb{N}_0$ , die asymptotisch genauso stark wie  $g$  wachsen.

$$\Theta(g) = \{f \mid f \in \mathcal{O}(g) \wedge f \in \Omega(g)\}$$

*Beispiele:*

- $37 \cdot n^3 + 2 \cdot n^2 \in \Theta(n^3)$
- $3 \cdot n^2 + 42 \cdot n \cdot \log(n) \in \Theta(n^2)$
- $139 \cdot n + 17 \cdot \sqrt{n} \in \Theta(n)$
- $0,7 \cdot 2^n + 928 \cdot n^4 \in \Theta(2^n)$
- $c_k \cdot n^k + c_{k-1} \cdot n^{k-1} + \dots + c_0 \in \Theta(n^k)$  für  $c_1, \dots, c_k$  konstant

*Beispiele:*

$6n^3 + 2n^2 + 7n - 10$	$\in \mathcal{O}(n^3)$	$\in \Omega(n^3)$	$\in \Theta(n^3)$
$n^2 \log(n)$	$\notin \mathcal{O}(n^2)$	$\in \Omega(n^2)$	$\notin \Theta(n^2)$
$n^k$ für $k > 0$ und $c > 1$	$\in \mathcal{O}(c^n)$	$\notin \Omega(c^n)$	$\notin \Theta(c^n)$
$\log^k(n)$ für $k > 0$	$\in \mathcal{O}(n)$	$\notin \Omega(n)$	$\notin \Theta(n)$

In der Mathematik findet man Aussagen der Art

$$\frac{1}{n^2} \in \mathcal{O}\left(\frac{1}{n}\right).$$

Bei der Laufzeitanalyse von Algorithmen kommen monoton fallende Funktionen nicht vor, da Algorithmen bei größerer Eingabe nicht kürzer laufen. Obwohl es Einbrüche bei der Laufzeit geben kann.

In der Regel bestehen Algorithmen aus verschiedenen Teilen, die unterschiedliche Laufzeiten haben. Um die Laufzeit des gesamten Algorithmus angeben zu können, benötigen wir Rechenregeln.

*Übung:* Beweisen Sie die folgende Aussage.

$\forall g : \mathbb{N}_0 \rightarrow \mathbb{N}_0 \ \forall f \in \mathcal{O}(g) \ \forall c \in \mathbb{N}$  gilt:

- $f + c \in \mathcal{O}(g)$ : Führe eine Prozedur und eine konstante Anzahl von Operationen aus.
- $f \cdot c \in \mathcal{O}(g)$ : Führe eine Prozedur  $c$ -mal aus.

Dabei soll  $f + c$  die Funktion  $f' : \mathbb{N}_0 \rightarrow \mathbb{N}_0$  mit  $f'(n) := f(n) + c$  bezeichnen. Analog sei  $f \cdot c$  als  $f''(n) := f(n) \cdot c$  definiert.

Wir gehen davon aus, dass die Funktionen nicht monoton fallend sind, sondern Laufzeiten von Algorithmen beschreiben.

Einfluss additiver, konstanter Terme:

$$f \in \mathcal{O}(g)$$

$$\Rightarrow \exists n_0 \exists c_1 \forall n \geq n_0 : 0 \leq f(n) \leq c_1 \cdot g(n)$$

$$\Rightarrow \exists n_0 \exists c_1 \forall n \geq n_0 : 0 \leq f(n) + c \leq c_1 \cdot g(n) + c$$

$$\Rightarrow \exists n_0 \exists c_1 \forall n \geq n_0 : 0 \leq f(n) + c \leq c_1 \cdot g(n) + c \cdot g(n)$$

$$\Rightarrow \exists n_0 \exists c_1 \forall n \geq n_0 : 0 \leq f(n) + c \leq (c_1 + c) \cdot g(n)$$

$$\Rightarrow f + c \in \mathcal{O}(g)$$

Einfluss konstanter Faktoren:

$$f \in \mathcal{O}(g) \Rightarrow \exists n_0 \exists c_1 \forall n \geq n_0 : 0 \leq f(n) \leq c_1 \cdot g(n)$$

$$\Rightarrow \exists n_0 \exists c_1 \forall n \geq n_0 : 0 \leq f(n) \cdot c \leq c_1 \cdot g(n) \cdot c$$

$$\Rightarrow \exists n_0 \exists c_1 \forall n \geq n_0 : 0 \leq f(n) \cdot c \leq (c_1 \cdot c) \cdot g(n)$$

$$\Rightarrow f \cdot c \in \mathcal{O}(g)$$

*Übung:* Beweisen Sie die folgende Aussage.

$\forall g_1, g_2 : \mathbb{N}_0 \rightarrow \mathbb{N}_0 \quad \forall f_1 \in \mathcal{O}(g_1) \quad \forall f_2 \in \mathcal{O}(g_2)$  gilt:

- $f_1 + f_2 \in \mathcal{O}(g_1 + g_2)$ : Führe zwei verschiedene Prozeduren hintereinander aus.
- $f_1 \cdot f_2 \in \mathcal{O}(g_1 \cdot g_2)$ : Eine Prozedur mit Laufzeit  $f_2$  wird jeweils innerhalb einer Schleife aufgerufen, wobei der Schleifenrumpf die Laufzeit  $f_1$  hat.

Dabei bezeichnet

- $f_1 + f_2$  die Funktion  $f' : \mathbb{N}_0 \rightarrow \mathbb{N}_0$  mit  $f'(n) := f_1(n) + f_2(n)$
- und  $f_1 \cdot f_2$  die Funktion  $f''(n) := f_1(n) \cdot f_2(n)$ .

Addition zweier Laufzeiten:

$$f_1 \in \mathcal{O}(g_1) \Rightarrow \exists n_1 \exists c_1 \forall n \geq n_1 : 0 \leq f_1(n) \leq c_1 \cdot g_1(n)$$

$$f_2 \in \mathcal{O}(g_2) \Rightarrow \exists n_2 \exists c_2 \forall n \geq n_2 : 0 \leq f_2(n) \leq c_2 \cdot g_2(n)$$

Sei  $n_0 := \max\{n_1, n_2\}$  und  $c := c_1 + c_2$ . Dann gilt für  $n \geq n_0$ :

$$\begin{aligned} f_1(n) + f_2(n) &\leq c_1 \cdot g_1(n) + c_2 \cdot g_2(n) \\ &\leq c_1 \cdot \max\{g_1(n), g_2(n)\} + c_2 \cdot \max\{g_1(n), g_2(n)\} \\ &= c \cdot \max\{g_1(n), g_2(n)\} \\ &\Rightarrow f_1 + f_2 \in \mathcal{O}(\max\{g_1, g_2\}) \end{aligned}$$

Damit gilt natürlich auch  $f_1 + f_2 \in \mathcal{O}(g_1 + g_2)$ . Außerdem folgt damit unmittelbar  $f_1 \in \mathcal{O}(g) \wedge f_2 \in \mathcal{O}(g) \Rightarrow f_1 + f_2 \in \mathcal{O}(g)$ .

Multiplikation zweier Laufzeiten:

$$f_1 \in \mathcal{O}(g_1) \Rightarrow \exists n_1 \exists c_1 \forall n \geq n_1 : 0 \leq f_1(n) \leq c_1 \cdot g_1(n)$$

$$f_2 \in \mathcal{O}(g_2) \Rightarrow \exists n_2 \exists c_2 \forall n \geq n_2 : 0 \leq f_2(n) \leq c_2 \cdot g_2(n)$$

Sei  $n_0 := \max\{n_1, n_2\}$ . Dann gilt für  $n \geq n_0$ :

$$\begin{aligned} f_1(n) \cdot f_2(n) &\leq (c_1 \cdot g_1(n)) \cdot (c_2 \cdot g_2(n)) \\ &= \underbrace{(c_1 \cdot c_2)}_{=:c} \cdot g_1(n) \cdot g_2(n) \\ &\Rightarrow f_1 \cdot f_2 \in \mathcal{O}(g_1 \cdot g_2) \end{aligned}$$



## Wichtige Aufwandsklassen:

$\mathcal{O}(1)$	konstant	$\mathcal{O}(n^2)$	quadratisch
$\mathcal{O}(\log(n))$	logarithmisch	$\mathcal{O}(n^3)$	kubisch
$\mathcal{O}(\log^k(n))$	poly-logarithmisch	$\mathcal{O}(n^k)$	polynomiell
$\mathcal{O}(n)$	linear	$\mathcal{O}(2^n)$	exponentiell
$\mathcal{O}(n \cdot \log(n))$			

*Frage:* Warum geben wir bei der logarithmischen Laufzeit keine Basis des Logarithmus an?

*Antwort:* Die Basis des Logarithmus spielt bei der asymptotischen Aufwandsabschätzung keine Rolle.

Aus der Mathematik wissen wir:

$$\log_b(n) = \frac{\log_a(n)}{\log_a(b)} = \underbrace{\frac{1}{\log_a(b)}}_{=:c} \cdot \log_a(n)$$

Die Logarithmen zu unterschiedlichen Basen unterscheiden sich nur durch einen konstanten Faktor, der in der asymptotischen Aufwandsabschätzung, also in der Groß-O-Notation, weg fällt.

*Übung:* Beweisen Sie folgende Inklusionen.

$$\begin{aligned}\mathcal{O}(1) &\subset \mathcal{O}(\log(n)) \subset \mathcal{O}(\log^2(n)) \subset \mathcal{O}(\sqrt{n}) \\ &\subset \mathcal{O}(n) \subset \mathcal{O}(n \cdot \log(n)) \subset \mathcal{O}(n^2) \subset \mathcal{O}(n^3) \subset \mathcal{O}(2^n)\end{aligned}$$

Zur Erinnerung:

$$\begin{aligned}\frac{d}{dx} \ln(x) &= \frac{1}{x} & \frac{d}{dx} \ln^2(x) &= \frac{2 \cdot \ln(x)}{x} \\ \frac{d}{dx} x^k &= k \cdot x^{k-1} & \frac{d}{dx} \sqrt{x} &= \frac{d}{dx} x^{\frac{1}{2}} = \frac{1}{2} x^{-\frac{1}{2}} = \frac{1}{2\sqrt{x}} \\ \frac{d}{dx} e^x &= e^x \\ \frac{d}{dx} 2^x &= \frac{d}{dx} (e^c)^x = \frac{d}{dx} e^{c \cdot x} = e^{c \cdot x} \cdot c = (e^c)^x \cdot \ln(2) = 2^x \cdot \ln(2)\end{aligned}$$

$$\mathcal{O}(\ln^2(n)) \subset \mathcal{O}(\sqrt{n})$$

Wir wenden die Regel von l'Hospital an und beschränken uns auf den natürlichen Logarithmus:

$$\begin{aligned}\lim_{x \rightarrow \infty} \frac{\ln^2(x)}{\sqrt{x}} &= \lim_{x \rightarrow \infty} \frac{\frac{d}{dx} \ln^2(x)}{\frac{d}{dx} \sqrt{x}} = \lim_{x \rightarrow \infty} \frac{2 \ln(x) \cdot 2\sqrt{x}}{x} \\&= \lim_{x \rightarrow \infty} \frac{4 \ln(x)}{\sqrt{x}} = 4 \cdot \lim_{x \rightarrow \infty} \frac{\frac{d}{dx} \ln(x)}{\frac{d}{dx} \sqrt{x}} \\&= 4 \cdot \lim_{x \rightarrow \infty} \frac{2\sqrt{x}}{x} = 8 \cdot \lim_{x \rightarrow \infty} \frac{1}{\sqrt{x}} = 0\end{aligned}$$

Also wächst die Wurzelfunktion stärker als der Logarithmus, und daher gilt  $\mathcal{O}(\log^2(n)) \subset \mathcal{O}(\sqrt{n})$ .

$$\mathcal{O}(n^k) \subset \mathcal{O}(2^n)$$

Wir wenden die Regel von l'Hospital mehrmals an:

$$\begin{aligned}\lim_{x \rightarrow \infty} \frac{x^k}{2^x} &= \lim_{x \rightarrow \infty} \frac{\frac{d}{dx} x^k}{\frac{d}{dx} 2^x} = \lim_{x \rightarrow \infty} \frac{k \cdot x^{k-1}}{\ln(2) \cdot 2^x} \\&= \frac{k}{\ln(2)} \cdot \lim_{x \rightarrow \infty} \frac{\frac{d}{dx} x^{k-1}}{\frac{d}{dx} 2^x} = \frac{k}{\ln(2)} \cdot \lim_{x \rightarrow \infty} \frac{(k-1) \cdot x^{k-2}}{\ln(2) \cdot 2^x} \\&= \frac{k \cdot (k-1)}{\ln^2(2)} \cdot \lim_{x \rightarrow \infty} \frac{\frac{d}{dx} x^{k-2}}{\frac{d}{dx} 2^x} = \dots \\&= \frac{k!}{\ln^k(2)} \cdot \lim_{x \rightarrow \infty} \frac{1}{2^x} = 0\end{aligned}$$

Also wächst die Exponentialfunktion stärker als ein Polynom vom Grad  $k$ , und daher gilt  $\mathcal{O}(n^k) \subset \mathcal{O}(2^n)$ .

Weitere Rechenregeln:

- $f \in \mathcal{O}(g) \Rightarrow f + g \in \mathcal{O}(g)$
- $f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(h) \Rightarrow f \in \mathcal{O}(h)$
- für festes  $k \in \mathbb{N}$  gilt  $\Theta(\log(n^k)) = \Theta(\log(n))$ .

Oft benötigen wir asymptotische Aufwandsabschätzungen für mehrstellige Funktionen:

- naive Textsuche: verschiebe ein Muster der Länge  $m$  über einen Text der Länge  $n$  und prüfe stellenweise auf Gleichheit. Dabei ergibt sich als Laufzeit  $(n - (m - 1)) \cdot m$ .
- Bei der Tiefensuche auf einem Graphen mit  $n$  Knoten und  $m$  Kanten ergibt sich eine Laufzeit, die von  $n$  und  $m$  abhängt.

Wir erweitern daher die Groß-O-Notation auf zweistellige Funktionen  $g : \mathbb{N}_0^2 \rightarrow \mathbb{N}_0$ .

$$\mathcal{O}(g) := \left\{ f : \mathbb{N}_0^2 \rightarrow \mathbb{N}_0 \mid \begin{array}{l} \exists c \in \mathbb{N} \exists n_0 \in \mathbb{N} \forall x, y \in \mathbb{N} : \\ x, y \geq n_0 \Rightarrow f(x, y) \leq c \cdot g(x, y) \end{array} \right\}$$

Man unterscheidet die Laufzeit

- im besten Fall (best case),
- im Mittel (average case) und
- im schlechtesten Fall (worst case).

Auf den nächsten Folien: Lineare vs. binäre Suche.

Algorithmus	best case	average case	worst case
Lineare Suche	$\mathcal{O}(1)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$
Binäre Suche	$\mathcal{O}(1)$	$\mathcal{O}(\log(N))$	$\mathcal{O}(\log(N))$



*Definition:* (Worst-Case Komplexität)

$W_n$ : Menge der zulässigen Eingaben der Länge  $n$ .

$A(w)$ : Anzahl Schritte von Algorithmus  $A$  für Eingabe  $w$ .

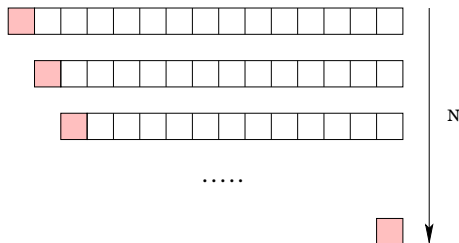
*Worst-Case Komplexität* (im schlechtesten Fall):

$$T_A(n) = \sup\{A(w) \mid w \in W_n\}$$

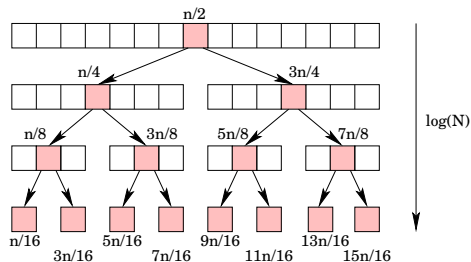
ist eine obere Schranke für die maximale Anzahl der Schritte, die Algorithmus  $A$  benötigt, um Eingaben der Größe  $n$  zu bearbeiten.

# Worst-Case Komplexität: Beispiel

## Lineare Suche:



## Binäre Suche:



## Zum Vergleich:

$N$	$\log(N)$
1.000.000	20
1.000.000.000	30
1.000.000.000.000	40

*Definition:* (Average-Case Komplexität)

$W_n$ : Menge der zulässigen Eingaben der Länge  $n$ .

$A(w)$ : Anzahl Schritte von Algorithmus  $A$  für Eingabe  $w$ .

*Average-Case Komplexität* (erwarteter Aufwand):

$$\overline{T}_A(n) = \frac{1}{|W_n|} \cdot \sum_{w \in W_n} A(w)$$

ist die mittlere Anzahl von Schritten, die Algorithmus  $A$  benötigt, um eine Eingabe der Größe  $n$  zu bearbeiten.

Wir setzen hier und im Verlauf dieser Veranstaltung eine Gleichverteilung voraus.

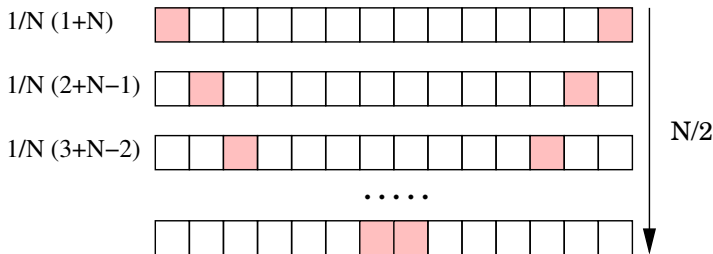
→ arithmetischer Mittelwert

# Average-Case Komplexität: Beispiel

## Lineare Suche:

Kosten:  $1, \dots, N$  Vergleiche

erwartete Kosten:  $\frac{1}{N} \cdot (1 + 2 + 3 + \dots + N)$



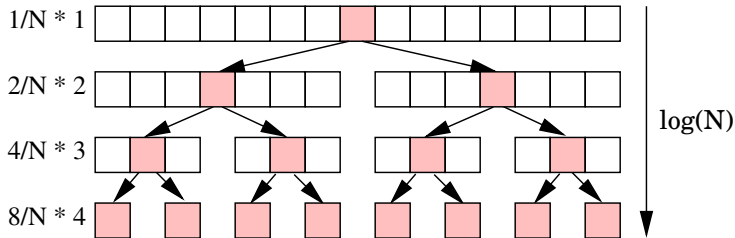
$$\frac{1}{N} \cdot (1 + 2 + 3 + \dots + N) = \frac{1}{N} \cdot \frac{N(N+1)}{2} = \frac{N+1}{2}$$

## Binäre Suche:

Kosten:  $1, \dots, \log(N)$  Vergleiche

Vereinfachung:  $N = 2^x - 1$

erwartete Kosten:  $\frac{1}{N} \cdot (1 \cdot 1 + 2 \cdot 2 + 4 \cdot 3 + \dots + 2^{x-1} \cdot x)$



Behauptung:

$$\sum_{i=1}^x i \cdot 2^{i-1} = (x-1) \cdot 2^x + 1$$

Beweis mittels vollständiger Induktion:

I.A.  $x = 1$ :

$$1 \cdot 2^0 = 1 \cdot 1 = 1 \stackrel{!}{=} 0 \cdot 2^1 + 1 = 1$$

I.S.  $x \rightarrow x+1$ :

$$\begin{aligned} \sum_{i=1}^{x+1} i \cdot 2^{i-1} &= \sum_{i=1}^x i \cdot 2^{i-1} + (x+1) \cdot 2^x \\ &\stackrel{I.V.}{=} (x-1) \cdot 2^x + 1 + (x+1) \cdot 2^x \\ &= 2x \cdot 2^x + 1 = x \cdot 2^{x+1} + 1 \end{aligned}$$

Aus der Annahme  $N = 2^x - 1$  folgt:  $\log_2(N + 1) = x$

Somit ergibt sich:

$$\begin{aligned}\frac{1}{N} \cdot \sum_{i=1}^x i \cdot 2^{i-1} &= \frac{1}{N} \cdot [(x - 1) \cdot 2^x + 1] \\ &= \frac{1}{N} \cdot [(\log_2(N + 1) - 1) \cdot (N + 1) + 1] \\ &= \frac{1}{N} \cdot [(N + 1) \cdot \log_2(N + 1) - N] \\ &\approx \log_2(N + 1) - 1 \text{ für große } N\end{aligned}$$

Im Mittel verursacht binäres Suchen also nur etwa eine Kosteneinheit weniger als im schlechtesten Fall.

## *Fragen / Probleme:*

- Worüber bildet man den Durchschnitt?
- Sind alle Eingaben der Länge  $N$  gleich wahrscheinlich? oder: Binomial-, Normal-, Poissonverteilung?
- Technisch oft sehr viel schwieriger durchzuführen als die worst-case Analyse.

*Murphys Gesetz:* Alles was schiefgehen kann, wird auch schiefgehen! Für uns heißt das: Immer wenn ich das Programm ausführe, warte ich ewig.

Average-Case Untersuchungen sind ungeeignet für kritische Anwendungen, bei denen maximale Reaktionszeiten garantiert werden müssen! → Vorlesung *Echtzeitsysteme*

## *Warum worst-case Komplexität?*

- Obere Schranke der Laufzeit: Wir können uns sicher sein, dass das Programm nach einer gewissen Zeit fertig ist.
- Der schlimmste Fall kommt bei manchen Algorithmen oft vor, z.B. beim Suchen nach Daten, die in einer Datenbank nicht vorhanden sind.
- Der durchschnittliche Fall ist oft fast genauso schlecht wie der schlimmste Fall.



## *Einleitung*

- Bewertung von Algorithmen
- *Berechenbarkeitstheorie*
- Komplexitätstheorie
- Exakte Algorithmen für schwere Probleme

*Definition:* Ein **Alphabet** ist eine endliche, nichtleere Menge von Zeichen (Symbole oder Buchstaben genannt).

*Definition:* Endliche Folgen  $(x_1, \dots, x_k)$  mit  $x_i \in A$  heißen **Wörter** der Länge  $k$  über dem Alphabet  $A$ .

*Definition:* Die Menge aller Wörter über einem Alphabet  $A$  wird mit  $A^*$  bezeichnet. Das leere Wort wird durch das Symbol  $\epsilon$  dargestellt.

Eine Teilmenge von  $A^*$  wird als **Sprache** bezeichnet.

*Satz:* Die Menge  $A^*$  aller Wörter über einem Alphabet  $A$  ist abzählbar.

*Beweis:*  $\rightarrow$  Übung

Die Menge  $A^*$  aller Wörter über einem Alphabet  $A$  ist abzählbar.

*Idee:* Sei  $A = \{a_1, \dots, a_k\}$  ein endliches Alphabet. Sei  $A^\ell$  die Menge aller Wörter der Länge  $\ell$  über dem Alphabet  $A$ . Dann gilt:

$$A^{\ell+1} = \{a_1, \dots, a_k\} \times A^\ell$$

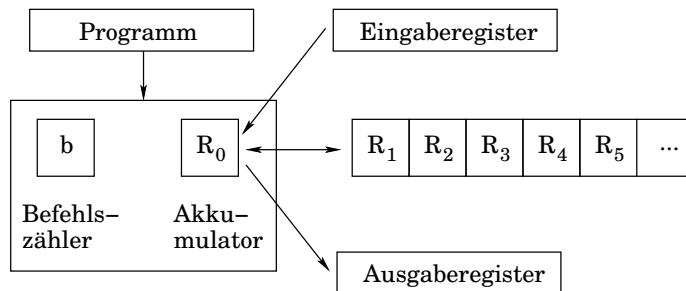
Daraus lässt sich direkt ein Algorithmus ableiten, der  $A^*$  aufzählt:

```
 $A^0 := \{\epsilon\}$   
for  $\ell := 1 \dots \infty$  do  
   $A^\ell := \emptyset$   
  for all  $a \in A$  do  
    for all  $w \in A^{\ell-1}$  do  
       $A^\ell := A^\ell \cup \{a \circ w\}$ 
```

Dabei bezeichnet  $a \circ w$  das Aneinanderfügen von  $a$  und  $w$ , also die String-Konkatenation.

## *Random Access Machine (RAM)* (Registermaschine)

- fester Befehlssatz: read, write, add, sub, goto, if goto, ...
- abzählbar unendlich viele Speicherzellen  $R_0, R_1, R_2, \dots$
- spezielles Register  $R_0$  für Arithmetik  $\rightarrow$  Akkumulator
- spezielles Register für Eingabe
- spezielles Register für Ausgabe



*einige Befehle:*

- *READ*:  $R_0 := \text{head}(e)$ ,  $e := \text{tail}(e)$ ,  $b := b + 1$
- *WRITE*:  $a := a \oplus R_0$ ,  $b := b + 1$
- *ADD  $R_i$* :  $R_0 := R_0 + R_i$ ,  $b := b + 1$
- *ADD  $i$* :  $R_0 := R_0 + i$ ,  $b := b + 1$
- *LOAD  $R_i$* :  $R_0 := R_i$ ,  $b := b + 1$
- *ILOAD  $R_i$* :  $R_0 := R_{R_i}$ ,  $b := b + 1$

*Kostenmaße:*

- *uniform*: Anzahl der ausgeführten Befehle bzw. der benutzten Speicherzellen.
- *logarithmisch*: Berücksichtige die binäre Länge der benutzten Operanden bei den jeweiligen Befehlen.

*Beispiel:*  $\text{ggT}(x, y)$  berechnen, zu Beginn  $R_1 = x$ ,  $R_2 = y$

Zeile	Anweisung	Kommentar
1:	LOAD (ACC FROM) $R_1$	$R_0 := R_1$
2:	STORE (ACC TO) $R_3$	$R_3 := R_0$
3:	SUB $R_2$ (FROM ACC)	$R_0 := R_0 - R_2$
4:	JLZ 6	jump to Z6 if $R_0 < 0$
5:	STORE (ACC TO) $R_3$	$R_3 := R_0$
6:	LOAD (ACC FROM) $R_2$	$R_0 := R_2$
7:	STORE (ACC TO) $R_1$	$R_1 := R_0$
8:	LOAD (ACC FROM) $R_3$	$R_0 := R_3$
9:	STORE (ACC TO) $R_2$	$R_2 := R_0$
10:	JNEZ 1	jump to Z1 if $R_0 \neq 0$
11:	HALT	

Am Ende steht das Ergebnis in  $R_1$ .

Veranschaulichung: Zu Beginn gelte  $R_1 = x$  und  $R_2 = y$ .

- |     |                    |               |                      |
|-----|--------------------|---------------|----------------------|
| 1.  | $R_0 := R_1$       | }             | $R_3 := x$           |
| 2.  | $R_3 := R_0$       |               |                      |
| 3.  | $R_0 := R_0 - R_2$ | }             | Z6 falls $x - y < 0$ |
| 4.  | Z6 if $R_0 < 0$    |               |                      |
| 5.  | $R_3 := R_0$       | $\rightarrow$ | $R_3 := x - y$       |
| 6.  | $R_0 := R_2$       | }             | $R_1 := y$           |
| 7.  | $R_1 := R_0$       |               |                      |
| 8.  | $R_0 := R_3$       | }             | $R_2 := R_3$         |
| 9.  | $R_2 := R_0$       |               |                      |
| 10. | Z1 if $R_0 \neq 0$ | $\rightarrow$ | Z1, falls $x \neq y$ |

als Pseudo-Code:

```
repeat
     $t := x$ 
    if  $x \geq y$  then
         $t := x - y$ 
     $x := y$ 
     $y := t$ 
until  $t = 0$ 
```

Sobald Zeile 9 abgearbeitet ist, gilt

- $\text{ggT}(x, y) = \text{ggT}(y, x - y)$ , falls  $x \geq y$  ist,
- andernfalls gilt  $\text{ggT}(x, y) = \text{ggT}(y, x)$ .

Korrektheit: Offensichtlich gilt  $\text{ggT}(a, b) = \text{ggT}(b, a)$ , also bleibt nur noch  $\text{ggT}(a, b) = \text{ggT}(a - b, b)$  zu zeigen.

- Sei  $t_1 = \text{ggT}(a, b)$ , dann gilt  $t_1 \mid a$  und  $t_1 \mid b$ , und deshalb:
  - $t_1 \mid a \iff \exists c_1 \in \mathbb{N} : a = t_1 \cdot c_1$
  - $t_1 \mid b \iff \exists c_2 \in \mathbb{N} : b = t_1 \cdot c_2$
  - $a > b \implies c_1 > c_2$

Wir halten fest:  $a - b = t_1 \cdot (c_1 - c_2)$  und damit  $t_1 \mid a - b$ .

Da  $t_1 \mid b$  und  $t_1 \mid a - b$  gilt, ist  $t_1$  offenbar ein gemeinsamer Teiler von  $b$  und  $a - b$ , und somit  $t_1 \leq \text{ggT}(a - b, b)$ , denn  $t_1$  kann nicht größer als der größte gemeinsame Teiler sein.

- Sei  $t_2 = \text{ggT}(a - b, b)$ , dann gilt  $t_2 \mid a - b$  und  $t_2 \mid b$ . Also:
  - $t_2 \mid a - b \iff \exists d_1 \in \mathbb{N} : a - b = t_2 \cdot d_1$
  - $t_2 \mid b \iff \exists d_2 \in \mathbb{N} : b = t_2 \cdot d_2$
  - und wegen  $(a - b) + b = t_2 \cdot (d_1 + d_2)$  gilt:  $t_2 \mid a$

Es gilt also analog zu oben:  $t_2 \leq \text{ggT}(a, b)$



Schließlich zeigen wir noch die Gleichheit. Wir hatten festgestellt:

- $t_1 = ggT(a, b) \leq ggT(a - b, b)$  und
- $t_2 = ggT(a - b, b) \leq ggT(a, b)$ .

Oder anders gesagt:  $t_1 \leq t_2$  und  $t_2 \leq t_1$ . Beide Ungleichungen können natürlich nur dann gelten, wenn  $t_1 = t_2$  gilt.

Könnte die RAM auch multiplizieren und dividieren, dann könnte die Berechnung beschleunigt werden, denn es gilt:

$$ggT(a, b) = ggT(a \bmod b, b)$$

Wegen unserer obigen Überlegung gilt:

$$ggT(a, b) = ggT(a - b, b) = ggT(a - 2b, b) = \dots = ggT(a - kb, b)$$

Sei  $r = a \bmod b$ . Dann existiert ein  $k$ , so dass  $a = kb + r$  gilt, und daher  $a - kb = r = a \bmod b$ , womit die Aussage gezeigt ist.

*Feststellung:* Die RAM ähnelt unseren heutigen Computern und entspricht intuitiv unserem Berechenbarkeitsbegriff:

*Eine Funktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  ist berechenbar, falls es ein RAM-Programm  $p$  gibt, so dass gilt:  $p$  berechnet  $m$  zur Eingabe  $n_1, \dots, n_k$  genau dann wenn  $f(n_1, \dots, n_k) = m$ .*

*Anmerkung:* Anstelle eines RAM-Programms können wir auch ein Programm in einer höheren Programmiersprache wie C/C++, Java oder Python verwenden, da diese Programme durch einen Compiler in Assembler-Code umgesetzt werden.

Die folgenden Beispiele sind dem sehr empfehlenswerten Buch von Uwe Schöning entnommen: Theoretische Informatik – kurzgefasst.

*Übung:* Ist folgende Funktion berechenbar?

$$f(n) = \begin{cases} 1, & \text{falls die Dezimalbruchentwicklung von } \pi \\ & \text{mit } n \text{ beginnt} \\ 0, & \text{sonst} \end{cases}$$

Beispiel:  $f(314) = 1$ ,  $f(31417) = 0$ ,  $f(31415) = 1$

*Übung:* Ist folgende Funktion berechenbar?

$$f(n) = \begin{cases} 1, & \text{falls die Dezimalbruchentwicklung von } \pi \\ & \text{mit } n \text{ beginnt} \\ 0, & \text{sonst} \end{cases}$$

Beispiel:  $f(314) = 1$ ,  $f(31417) = 0$ ,  $f(31415) = 1$

*Antwort:* Ja!

- Es gibt Näherungsverfahren für die Zahl  $\pi$ , die auch eine Abschätzung des Fehlers liefern.
- Diese werden solange ausgeführt, bis die entsprechende Genauigkeit erreicht ist.

*Übung:* Ist folgende Funktion berechenbar?

$$g(n) = \begin{cases} 1, & \text{in der Dezimalbruchentwicklung von } \pi \\ & \text{kommt irgendwo } n \text{ vor} \\ 0, & \text{sonst} \end{cases}$$

*Übung:* Ist folgende Funktion berechenbar?

$$g(n) = \begin{cases} 1, & \text{in der Dezimalbruchentwicklung von } \pi \\ & \text{kommt irgendwo } n \text{ vor} \\ 0, & \text{sonst} \end{cases}$$

*Antwort:* Vielleicht nicht!

- Unser bisheriges Wissen über die Zahl  $\pi$  reicht nicht aus, um die Frage zu beantworten.
- Falls  $\pi$  zufällig genug ist, dass jede Zahlenfolge irgendwann mal vorkommt, wäre  $g(n) = 1$  für alle  $n$  und damit wäre  $g$  berechenbar.

*Übung:* Ist folgende Funktion berechenbar?

$$h(n) = \begin{cases} 1, & \text{die Dezimalbruchentwicklung von } \pi \\ & \text{enthält irgendwo } n\text{-mal hintereinander eine } 7 \\ 0, & \text{sonst} \end{cases}$$

*Übung:* Ist folgende Funktion berechenbar?

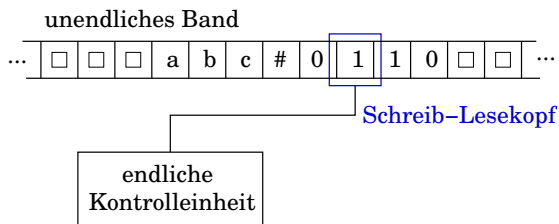
$$h(n) = \begin{cases} 1, & \text{die Dezimalbruchentwicklung von } \pi \\ & \text{enthält irgendwo } n\text{-mal hintereinander eine } 7 \\ 0, & \text{sonst} \end{cases}$$

*Antwort:* Ja!

- Falls beliebig lange 7-er Folgen in  $\pi$  vorkommen, dann ist  $h(n) = 1$  für alle  $n$ , also ist  $h$  für diesen Fall berechenbar.
- Falls aber nur 7-er Folgen bis zur Länge  $n_0$  enthalten sind, ist  $h(n) = 1$  für  $n \leq n_0$  und  $h(n) = 0$  sonst, also ist  $h$  auch in diesem Fall berechenbar.
- In jedem Fall *gibt es einen* Algorithmus zur Berechnung von  $h$ , auch wenn wir den Wert von  $n_0$  nicht kennen.



*deterministische  
Turing-Maschine (dTM)*



- Das potentiell unendliche Band ist in Felder unterteilt.
- Jedes Feld kann ein einzelnes Zeichen des Arbeitsalphabets der Maschine enthalten.
- Auf dem Band kann sich der Schreib-Lesekopf bewegen.
- Nur das Zeichen, auf dem sich dieser Kopf gerade befindet, kann im momentanen Rechenschritt verändert werden.
- Der Kopf kann in einem Rechenschritt nur um maximal eine Position nach links oder rechts bewegt werden.
- Noch nicht besuchte Felder enthalten das Blank-Symbol  $\square$ .

*Formal:* Eine Turingmaschine  $M$  ist gegeben durch ein 7-Tupel  $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$ .

- $Z$  ist die Zustandsmenge.
- $\Sigma$  ist das Eingabealphabet.
- $\Gamma$  ist das Ausgabealphabet mit  $\Sigma \subseteq \Gamma$ .
- $\delta : Z \times \Gamma \rightarrow Z \times \Gamma \times \{L, N, R\}$  ist die partielle Übergangsfunktion, die (u.a.) für keinen Wert aus  $E \times \Gamma$  definiert ist.
- $z_0 \in Z$  ist der Startzustand.
- $\square$  ist das Blank-Symbol.
- $E \subsetneq Z$  ist die Menge der akzeptierenden Endzustände.

Informal bedeutet  $\delta(z, a) = (z', b, x)$  folgendes:

Wenn sich Maschine  $M$

- im Zustand  $z$  befindet und
- unter dem Schreib-Lesekopf das Zeichen  $a$  steht,

dann

- geht  $M$  im nächsten Schritt in den Zustand  $z'$  über,
- schreibt auf den Platz von  $a$  das Zeichen  $b$  auf das Band
- und führt danach die Kopfbewegung  $x \in \{L, N, R\}$  aus mit  $L$  für links,  $N$  für neutral (stehenbleiben) und  $R$  für rechts.

*Definition:* Eine Konfiguration  $\alpha z \beta$  ist eine *Momentaufnahme* der Turingmaschine.

- $\alpha \beta \in \Gamma^*$  ist der nicht-leere, schon besuchte Teil des Bandes.
- Der Schreib-Lesekopf steht auf dem ersten Zeichen von  $\beta$ .
- $z$  ist der Zustand, in dem sich die Maschine befindet.

*Definition:* Startkonfiguration  $z_0 x$

- Die Eingabe  $x \in \Sigma^*$  steht schon auf dem Band.
- Der Schreib-Lesekopf steht auf dem ersten Zeichen der Eingabe  $x$ .
- Die Maschine ist im Startzustand  $z_0$ .

*Übung:* Geben Sie eine Turingmaschine an, die auf eine Eingabe, interpretiert als Binärzahl, eine 1 hinzu addiert.

Sei  $T = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$  die deterministische Turingmaschine mit

$$Z = \{z_0, z_1, z_2\} \quad E = \{z_2\} \quad \Sigma = \{0, 1\} \quad \Gamma = \{0, 1, \square\}$$

und folgender Übergangsfunktion:

$(z_0, 0)$	$\rightarrow$	$(z_0, 0, R)$
$(z_0, 1)$	$\rightarrow$	$(z_0, 1, R)$
$(z_0, \square)$	$\rightarrow$	$(z_1, \square, L)$
$(z_1, 1)$	$\rightarrow$	$(z_1, 0, L)$
$(z_1, 0)$	$\rightarrow$	$(z_2, 1, N)$
$(z_1, \square)$	$\rightarrow$	$(z_2, 1, N)$

*Definition:* Die von einer Turingmaschine  $M$  **akzeptierte Sprache** ist wie folgt definiert:

$$T(M) = \{x \in \Sigma^* \mid z_0x \vdash^* \alpha z \beta; \alpha, \beta \in \Gamma^*; z \in E\}$$

Dabei bedeutet  $z_0x \vdash^* \alpha z \beta$ , dass die Maschine ausgehend von der Startkonfiguration  $z_0x$  nach endlich vielen Schritten eine Konfiguration  $\alpha z \beta$  erreicht und hält.

*Beachte:* Für ein  $x \notin T(M)$  darf die Maschine  $M$  in eine Endlosschleife gehen!

*Definition:*  $L \subseteq \Sigma^*$  heißt **rekursiv aufzählbar**, wenn es eine dTM gibt, die  $L$  akzeptiert.

*Definition:* Falls  $M$  die Sprache  $L$  akzeptiert und für alle  $x \in \Sigma^*$  hält, so **entscheidet**  $M$  die Sprache  $L$ .

$L \subseteq \Sigma^*$  heißt **entscheidbar** oder auch **rekursiv**, wenn es eine dTM gibt, die  $L$  entscheidet.

*Übung:* Geben Sie eine Turingmaschine an, die  $L = \{0^n 1^n \mid n \geq 1\}$  entscheidet.

Für die Sprache  $L = \{0^n 1^n \mid n \geq 1\}$  sei  $T = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$  die folgende deterministische Turingmaschine:

$$Z = \{z_0, z_1, z_2, \dots, z_7\} \quad E = \{z_7\} \quad \Sigma = \{0, 1\} \quad \Gamma = \{0, 1, \square\}$$

Die Idee der Übergangsfunktion:

- $z_0, z_1$ : Prüfe, ob  $x = 0^i 1^j$  ist. Falls nicht, gibt es keinen Nachfolgezustand.

$\delta$	0	1	$\square$
$z_0$	$(z_0, 0, R)$	$(z_1, 1, R)$	— — —
$z_1$	— — —	$(z_1, 1, R)$	$(z_2, \square, L)$

- Prüfe, ob  $i = j$  gilt: Falls rechts eine 1 gelesen wird, lösche diese ( $z_2$ ), laufe nach links ( $z_3$ ) und suche dort eine entsprechende 0 und streiche diese ebenfalls ( $z_4$ ).

$\delta$	0	1	$\square$
$z_2$	— — —	$(z_3, \square, L)$	— — —
$z_3$	$(z_3, 0, L)$	$(z_3, 1, L)$	$(z_4, \square, R)$
$z_4$	$(z_5, \square, R)$	— — —	— — —

- Prüfe in  $z_5$ , ob die Eingabe komplett abgearbeitet wurde. Falls nicht, laufe in  $z_6$  zurück nach rechts.

$\delta$	0	1	$\square$
$z_5$	$(z_6, 0, R)$	$(z_6, 1, R)$	$(z_7, \square, N)$
$z_6$	$(z_6, 0, R)$	$(z_6, 1, R)$	$(z_2, \square, L)$



Bei der RAM haben wir von *Berechnung einer Funktion* gesprochen, bei der dTM sprechen wir von *akzeptierten Sprachen*. Wo ist da der Zusammenhang?

Eine dTM  $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$  berechnet eine Funktion  $f : \Sigma^* \rightarrow \Sigma^*$  genau dann, wenn für ein  $z_e \in E$  gilt:

$$z_0 x \vdash^* z_e y \iff f(x) = y$$

Die Turingmaschine schreibt also  $y$  auf das Band und geht in einen akzeptierenden Endzustand.

Für Funktionen  $f : \mathbb{N}^r \rightarrow \mathbb{N}^s$ , die auf natürlichen Zahlen definiert sind, kann man entsprechend definieren: Die dTM  $M$  berechnet  $f$  genau dann, wenn  $M$  gestartet mit  $x = 0^{n_1+1}10^{n_2+1}1 \dots 10^{n_r+1}$  den Wert  $y = 0^{m_1+1}10^{m_2+1}1 \dots 10^{m_s+1}$  auf das Band schreibt<sup>1</sup>, falls  $f(n_1, \dots, n_r) = (m_1, \dots, m_s)$  ist.

---

<sup>1</sup>Hier werden die Zahlenwerte unär kodiert, damit man  $\Sigma = \{0, 1\}$  wählen kann. Man könnte auch eine binäre Kodierung der Zahlen nutzen und ein Symbol wie  $\#$  verwenden, um die Zahlen voneinander zu trennen, dann wäre  $\Sigma = \{0, 1, \#\}$ .

Das *Akzeptieren von Sprachen* und das *Berechnen von Funktionen* ist sehr ähnlich. Sei  $M$  eine dTM, die die Funktion  $f : \Sigma^* \rightarrow \Sigma^*$  berechnet. Dann kann sehr einfach eine dTM  $M'$  konstruiert werden, die die Sprache  $L = \{(x, f(x))\}$  akzeptiert und dafür  $M$  nutzt.

Beispiel: Eine dTM  $M$  berechne die Funktion  $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  mit  $f(x, y) = x + y$ . Die zur Funktion  $f$  entsprechende Sprache ist

$$L = \{(x, y, z) \mid x + y = z\}$$

Eine dTM  $M'$ , die die Sprache  $L$  akzeptiert, nutzt die dTM  $M$  wie folgt:

- $M'$  verhält sich wie  $M$  gestartet mit  $0^{x+1}10^{y+1}$ .
- $M'$  akzeptiert genau dann, wenn das Ergebnis gleich  $0^{z+1}$  ist.

Die umgekehrte Richtung ist etwas problematischer. Sei  $M$  eine dTM, die die Sprache  $L$  akzeptiert. Dann kann eine dTM  $M'$  konstruiert werden, die die *charakteristische Funktion*  $\chi_L$  berechnet:

$$\chi_L(x) = 1 \iff x \in L$$

Die dTM  $M'$  verhält sich wie  $M$  gestartet mit  $x$ .

- Wenn  $M$  die Eingabe  $x$  akzeptiert, löscht  $M'$  das Band und schreibt eine 1.
- Wenn  $M$  die Eingabe  $x$  nicht akzeptiert, löscht  $M'$  das Band und schreibt eine 0.

Problem: Wenn die Maschine  $M$  die Eingabe  $x$  nicht akzeptiert, kann es sein, dass  $M$  nicht hält. In diesem Fall kann  $M'$  das Band nicht löschen und eine 0 schreiben.

Wenn  $M$  die Sprache  $L$  entscheidet, also insbesondere für jede Eingabe hält, so kann die charakteristische Funktion  $\chi_L$  berechnet werden.

Im weiteren wollen wir die Begriffe Akzeptanz und Entscheidung nutzen.

*Satz:* Die Klasse der rekursiven Sprachen ist abgeschlossen gegen Komplementbildung.

*Beweisidee:* Sei  $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$  eine dTM, die  $L$  entscheidet.

Folgende dTM  $\bar{M}$  akzeptiert  $\bar{L}$ :

$$\bar{M} = (Q \cup \{q^*\}, \Sigma, \Gamma, \bar{\delta}, q_0, \square, \{q^*\})$$

- Die akzeptierenden Endzustände von  $M$  sind bei  $\bar{M}$  nicht akzeptierend.
- Stattdessen gibt es bei  $\bar{M}$  einen neuen akzeptierenden Endzustand  $q^*$ .
- $\bar{\delta}$  ist definiert wie  $\delta$ , enthält aber Zusatzregeln: Falls  $M$  im Zustand  $q \notin F$  hält, geht  $\bar{M}$  erst in den Zustand  $q^*$  und hält dann.

*Satz:* Eine Sprache  $L$  ist entscheidbar, genau dann wenn  $L$  und  $\bar{L}$  rekursiv aufzählbar sind.

*Beweisidee:*

⇒ Da  $L$  entscheidbar ist, ist auch  $\bar{L}$  entscheidbar.

Außerdem gilt: Jede entscheidbare Sprache ist insbesondere rekursiv aufzählbar.

⇐ Seien  $M_1$  und  $M_2$  die TMs, die  $L$  bzw.  $\bar{L}$  akzeptieren.

Die dTM  $M$  führt jeweils einen Schritt von  $M_1$  und  $M_2$  aus und hält, falls eine der beiden Maschinen hält.  $M$  akzeptiert, falls  $M_1$  akzeptiert, und verwirft, falls  $M_2$  akzeptiert.

## *Kostenmaße:*

- *Laufzeit*: Die Anzahl ausgeführter Konfigurationsübergänge.
- *Speicherplatz*: Die maximale Länge einer Konfiguration während einer Rechnung.

## *Programmiertechniken für Turingmaschinen:*

- Im Zustand merken.
- Nutzen mehrerer Spuren.
- Nutzen mehrerer Bänder.
- Unterprogramme.

*Im Zustand merken:* Man kann eine von endlich vielen Informationen aus einer endlichen Menge  $A$  speichern, indem man sie sich „im Zustand merkt“. Dazu ersetzt man  $Q$  durch  $Q \times A$ .

*Beispiel:* Wir wollen testen, ob bei Eingabe  $x_1 \dots x_n \in \Sigma^+$  der Buchstabe  $x_1$  in  $x_2 \dots x_n$  vorkommt. Wir merken uns  $x_1$  im Zustand.

Sei  $\Gamma = \Sigma \cup \{\square\}$ ,  $Q = (\{q_0\} \times \Sigma) \cup \{q_0, q_1\}$ , Startzustand  $q_0$ ,  $F = \{q_1\}$ .

$$\begin{aligned}\delta(q_0, a) &= ([q_0, a], a, R) && \forall a \in \Sigma \\ \delta([q_0, a], a) &= (q_1, a, N) && \forall a \in \Sigma \\ \delta([q_0, a], b) &= ([q_0, a], b, R) && \forall a, b \in \Sigma, a \neq b\end{aligned}$$

*Nutzen mehrerer Spuren:* Um auf  $k$  Spuren unterschiedliche Worte schreiben zu können, benutzen wir als Bandalphabet  $\Gamma^k$  anstatt  $\Gamma$ .

E	R	I	K
M	A	G	
L	E	N	A

↑

Im obigen Beispiel ist  $k = 3$ , der Bandinhalt der Kopfposition ist  $[RAE] \in \{A, \dots, Z\}^3$ .

Anwendung dieser Technik im Beweis zu folgendem Satz:

*Satz:* Jede  $t(n)$ -zeit-,  $s(n)$ -platzbeschränkte  $k$ -Band-dTM kann durch eine 1-Band-dTM in Zeit  $O(t(n) \cdot s(n))$  auf Platz  $O(s(n))$  simuliert werden.



*k-Band-dTM*: Eine  $k$ -Band Turingmaschine hat nicht nur ein Band und einen Kopf, sondern  $k$  Bänder mit je einem Kopf.

- Die Übergangsfunktion ist von der Form  $\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{R, L, N\}^k$ .
- Zu Beginn steht die Eingabe auf Band 1, sonst stehen überall Blanks.
- Die Arbeitsweise ist analog zu 1-Band-dTMs definiert.

Mehrband-dTM verwenden wir um Unterprogramme zu realisieren (siehe dazu nächsten Abschnitt).

*Übung*: Geben Sie jeweils eine 1-Band- und eine 2-Band-dTM an, die die Sprache  $\{ww^R \mid w \in \{0, 1\}^*\}$  entscheidet.

$$L = \{ww^R \mid w \in \{0,1\}^*\} = \{00, 11, 0110, 1001, 1111, \dots\}$$

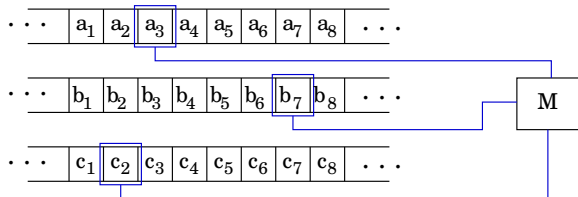
*1-Band-TM:* Da wir die Mitte des Wortes nicht kennen, gehen wir vor wie bei der Sprache  $\{0^n1^n \mid n \geq 1\}$ :

- Merke im Zustand das erste Zeichen, lösche das Symbol und laufe nach rechts.
- Falls das letzte Zeichen gleich dem ersten Zeichen ist, lösche das letzte Zeichen und laufe zurück zum Anfang.

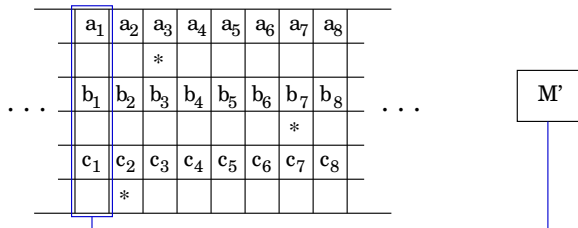
*2-Band-TM:*

- Kopiere die Eingabe auf das zweite Band,
- laufe auf dem ersten Band an den Anfang zurück,
- prüfe, ob die Ziffern auf den Bändern gleich sind
- und gehe zur nächsten Ziffer (Band 1 rechts, Band 2 links).

*Beweis:* Sei  $k$  die Anzahl der Bänder und  $\Gamma$  das Arbeitsalphabet von  $M$ . Wir unterteilen das Band von  $M'$  in  $2k$  Spuren, so dass eine Konfiguration von  $M$



simuliert wird durch:



Als Arbeitsalphabet von  $M'$  wählen wir  $\Gamma' = \Gamma \cup (\Gamma \cup \{\star\})^{2k}$ .

$M'$  simuliert einen Schritt von  $M$  wie folgt:

- Der Kopf von  $M'$  steht zu Beginn des Simulationsschrittes links von allen  $\star$ -Marken.
- $M'$  durchläuft das Band bis alle  $\star$ -Marken überschritten wurden und merkt sich die von  $M'$  gelesenen Zeichen an den  $\star$ -Positionen im Zustand. Dazu wird nur ein endlicher Speicher benötigt, also  $Q' = Q \times (\Gamma^k \times Q)$ .
- Nun weiß  $M'$ , welche Zeile der  $\delta$ -Funktion von  $M$  anzuwenden ist.
- $M'$  läuft wieder nach links über alle  $\star$ -Marken hinweg und führt alle entsprechenden Änderungen aus (Änderung der Inschriften auf ungeraden Spuren und Versetzen der  $\star$ -Marken) und merkt sich den neuen Zustand von  $M$  im Zustand.

## *Unterprogramme:*

- hier nur für Mehrband-dTM -

- Eine Teilmenge von Zuständen, die für das UP reserviert sind.
- Das UP wird von einem bestimmten Zustand der dTM aufgerufen.
- Die zu übergebenden Daten werden auf ein für UPe reserviertes Band der dTM kopiert.
- Das UP arbeitet nur auf diesem Band und das Ergebnis der Berechnung wird am Ende des UPs auf andere Bänder der dTM kopiert, um für weitere Berechnungen zur Verfügung zu stehen.
- Die Maschine geht nach Beendigung des UPs in einen Zustand über, der nicht zur Menge der für das UP reservierten Zustände gehört.

*Beispiel:* In der Turingmaschine für die Sprache  $L = \{0^n 1^n \mid n \geq 1\}$  können die Zustände  $z_0, z_1$  als UP aufgefasst werden.

*Satz:* dTMs und RAMs können mit polynomielltem Zeitverlust gegenseitig simuliert werden.

*Beweisidee:* Schalte mehrere Turingmaschinen hintereinander.

$M_1 = (Z_1, \Sigma, \Gamma_1, \delta_1, z_1, \square, E_1)$  und  $M_2 = (Z_2, \Sigma, \Gamma_2, \delta_2, z_2, \square, E_2)$  seien zwei Turingmaschinen. Dann bezeichnet

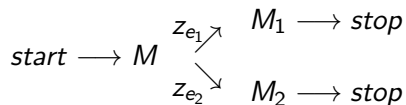
$$start \longrightarrow M_1 \longrightarrow M_2 \longrightarrow stop$$

eine neue TM  $M = (Z_1 \cup Z_2, \Sigma, \Gamma_1 \cup \Gamma_2, \delta, \square, E_2)$  wobei

- o.B.d.A.  $Z_1 \cap Z_2 = \emptyset$  und
- $\delta = \delta_1 \cup \delta_2 \cup \{(z_e, a, z_2, a, N) \mid z_e \in E_1, a \in \Gamma_1\}$

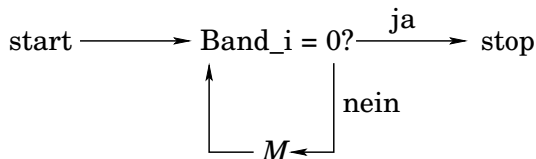
## Beweisidee: (Fortsetzung)

- Bezeichne  $\text{Band} := \text{Band} + 1$  unsere Addiermaschine.
- Daraus bauen wir eine  $k$ -Band TM  $\text{Band\_i} := \text{Band\_i} + 1$ , bei der die Aktionen nur auf Band  $i$  ablaufen und alle anderen Bänder unverändert bleiben.
- In ähnlicher Weise können wir Maschinen konstruieren für:
  - $\text{Band\_i} := \text{Band\_i} - 1$
  - $\text{Band\_i} := 0$
  - $\text{Band\_i} := \text{Band\_j}$
  - $\text{Band\_i} = 0?$
- Wir konstruieren eine TM, die vom Endzustand  $z_{e_1}$  von  $M$  nach  $M_1$  übergeht und von  $z_{e_2}$  aus nach  $M_2$  (Verzweigung):



## Beweisidee: (Fortsetzung)

- Wir können auch Schleifen realisieren:



Man erkennt, dass wir einfache Programmiersprachen-ähnliche Konzepte mit einer Mehrband-TM simulieren können:

- Die Bandinhalte können als Variablenwerte angesehen werden,
- es gibt einfache Wertzuweisungen,
- die Hintereinanderausführung von Programmen ist möglich,
- Verzweigungen und Schleifen können programmiert werden.



*Church These (1936)*: Die im intuitiven Sinne berechenbaren Funktionen sind genau die, die durch Turingmaschinen berechenbar sind.

- Church:  $\lambda$ -Kalkül
- Kleene:  $\mu$ -Rekursion
- Markov: Markov-Algorithmen

Alle vorgeschlagenen Formalisierungen von Berechenbarkeit sind äquivalent zur Berechenbarkeit durch eine Turingmaschine.

## *nichtdeterministische Turing-Maschine (nTM)*

- Eine nTM besitzt anstelle der Übergangsfunktion eine Übergangsrelation, d.h. jede Konfiguration hat mehrere mögliche Nachfolgekonfigurationen. Daher sind bei einer festgelegten Maschine und einer festgewählten Eingabe mehrere Resultate möglich.
- Eine nTM akzeptiert eine Eingabe gdw. *es existiert eine* Berechnung von der Startkonfiguration in eine akzeptierende Endkonfiguration.

*Übung:* Geben Sie eine (2-Band) nTM an,

- die alle geradlängigen Palindrome über dem Alphabet  $\{0, 1\}$  erkennt, also die Sprache  $L = \{ww^R \mid w \in \{0, 1\}^*\}$  akzeptiert.
- die die Sprache  $L = \{ww \mid w \in \{0, 1\}^*\}$  akzeptiert.

$$L = \{ww^R \mid w \in \{0,1\}^*\}$$

- Kopiere einen beliebig langen Anfang des Wortes auf das zweite Band:

$$\delta(q_0, a, \square) = \{(q_0, a, a, R, R), (q_1, a, a, R, N)\}$$

„Rate“, wo  $w$  zu Ende ist durch Übergang nach  $q_1$ .

- Dann laufen die Köpfe in verschiedene Richtungen und prüfen zeichenweise, ob der Rest auf Band 1 mit dem Inhalt von Band 2 übereinstimmt:

$$\delta(q_1, a, a) = (q_1, a, a, R, L)$$

- Wurde zu Beginn richtig „geraten“, dann akzeptiere:

$$\delta(q_1, \square, \square) = (q_2, \square, \square, N, N)$$

$\Rightarrow T = (Z, \Sigma, \Gamma, \delta, q_0, \square, E)$  nTM mit

$$Z = \{q_0, q_1, q_2\} \quad E = \{q_2\} \quad \Sigma = \{0, 1\} \quad \Gamma = \{0, 1, \square\}$$

$$L = \{ww \mid w \in \{0,1\}^*\}$$

- Kopiere einen beliebig langen Anfang des Wortes auf das zweite Band:

$$\delta(q_0, a, \square) = \{(q_0, a, a, R, R), (q_1, a, a, R, L)\}$$

„Rate“, wo  $w$  zu Ende ist durch Übergang nach  $q_1$ .

- Laufe auf dem zweiten Band bis an den Anfang zurück:

$$\delta(q_1, a, b) = (q_1, a, b, N, L) \quad \text{und} \quad \delta(q_1, a, \square) = (q_2, a, \square, N, R)$$

- Laufe auf beiden Bändern nach rechts und vergleiche die Zeichen:

$$\delta(q_2, a, a) = (q_2, a, a, R, R)$$

- Wurde zu Beginn richtig „geraten“, akzeptiere:  $\delta(q_2, \square, \square) = (q_3, \square, \square, N, N)$

$\Rightarrow T = (Z, \Sigma, \Gamma, \delta, q_0, \square, E)$  nTM mit

$$Z = \{q_0, q_1, q_2, q_3\} \quad E = \{q_3\} \quad \Sigma = \{0, 1\} \quad \Gamma = \{0, 1, \square\}$$

Der folgende Satz zeigt, dass der Berechenbarkeitsbegriff durch den Nichtdeterminismus nicht erweitert wird!

*Satz:* Eine polynomiell zeitbeschränkte nTM kann durch eine exponentiell zeitbeschränkte dTM simuliert werden.

*Beweis:* → Übung

zur Erinnerung:

- Für eine Menge  $M$  ist  $\mathcal{P}(M) := \{A \mid A \subseteq M\}$  die Potenzmenge von  $M$ .
- Beispiel: Für  $M = \{1, 2, 3\}$  ergibt sich  
 $\mathcal{P}(M) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$
- Für eine natürliche Zahl  $n$  bezeichnet  $[n]$  die Menge  $\{1, 2, 3, \dots, n\}$ .

*Beweisidee:* Beschränkung auf *nichtdeterministische 1-Band TM*

$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, E)$ , die  $p(n)$  zeitbeschränkt ist, wobei  $p$  ein Polynom ist und  $\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{R, L, N\})$ .

Sei  $b$  die maximale Anzahl von Nachfolgekonfigurationen, die eine Konfiguration haben kann, also  $b = \max\{|\delta(q, a)| : q \in Q, a \in \Gamma\}$ .

Nummeriere die möglichen Zustandsübergänge von  $\delta(q, a)$  durch.

Die deterministische 3-Band TM verhält sich bei Eingabe  $x$  so:

- Zähle auf Band 2 nacheinander alle möglichen Folgen  $(v_1, v_2, \dots, v_T)$  mit  $v_i \in [b]$  auf, wobei  $T := p(|x|)$  ist.
- Kopiere die Eingabe  $x$  von Band 1 auf Band 3 und führe dann auf Band 3 die Berechnung von  $M$  aus, wobei im Schritt  $i$  der  $v_i$ -te Zustandsübergang ausgeführt wird.
- Akzeptiere, falls bei einer der Berechnungen  $M$  akzeptiert.

*Satz:* Es gibt Funktionen  $f : \{0, 1\}^* \rightarrow \{0, 1\}$ , die nicht durch ein C-Programm berechenbar sind.

*Beweisidee:*

- Die Menge  $P$  der C-Programme ist abzählbar unendlich: endlicher Text über einem endlichen Alphabet  $A$  mit Syntax-Check durch Compiler.
  - Jedes Programm berechnet genau eine Funktion:  
 $f : A^* \rightarrow A^* \cup \{\text{Programm hält nicht}\}$
  - $P$  ist abzählbar, die Menge  $F = \{f : \{0, 1\}^* \rightarrow \{0, 1\}\}$  ist überabzählbar.
- $\Rightarrow$  Es existiert ein  $f \in F$  das nicht von einem Programm aus  $P$  berechnet wird.

*Frage:* Warum ist  $F$  überabzählbar?

# Cantor'sches Diagonalisierungsverfahren

Annahme:  $F = \{f_1, f_2, f_3, \dots\}$  wäre aufzählbar. Dann erstellen wir eine Tabelle, in der wir die Funktionen gegen alle möglichen Eingaben auftragen:

	$f_1$	$f_2$	$f_3$	$f_4$	$f_5$	$\dots$		$f_1$	$f_2$	$f_3$	$f_4$	$f_5$	$\dots$	
$\epsilon$	0	1	0	0	1	$\dots$	$\longrightarrow$	$\epsilon$	1	1	0	0	1	$\dots$
0	1	1	0	1	0	$\dots$		0	1	0	0	1	0	$\dots$
1	1	0	0	0	0	$\dots$		1	1	0	1	0	0	$\dots$
00	0	1	1	1	1	$\dots$		00	0	1	1	0	1	$\dots$
01	0	1	1	0	0	$\dots$		01	0	1	1	0	1	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$		$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Die Funktion, deren Werte sich aus den negierten Werten der Diagonalen ergeben, kann nicht in der Aufzählung enthalten sein. ⚡

Obiger Beweis ist nicht konstruktiv. Wir wollen nun untersuchen, wie solche nicht berechenbaren Funktionen aussehen.



Bisher: Special Purpose Machines lösen nur genau ein Problem.

Wie können wir programmierbare (universelle) Computer darstellen? Das Programm soll dabei eine Beschreibung einer special purpose machine sein.

*Gödelnummer:* Sei  $M$  eine 1-Band-dTM mit  $Q = \{q_1, \dots, q_m\}$ ,  $\Sigma = \{0, 1\}$  und  $\Gamma = \{0, 1, \square\}$ .

Sei  $X_1 \hat{=} 0$ ,  $X_2 \hat{=} 1$ ,  $X_3 \hat{=} \square$ ,  $D_1 \hat{=} L$ ,  $D_2 \hat{=} R$ ,  $D_3 \hat{=} N$ .

Codiere  $\delta(q_i, X_j) = (q_k, X_l, D_m)$  durch  $0^i 10^j 10^k 10^l 10^m$ .

$\delta$  ist eine endliche Funktionstabelle, daher sei  $Code_t$  die Codierung der  $t$ -ten Zeile,  $1 \leq t \leq g$ . Die *Gödelnummer* von  $M$  ist

$$\langle M \rangle = 111Code_111Code_211Code_3 \dots 11Code_g111$$

wobei  $g \leq m \cdot |\Gamma|$  die Größe von  $\delta$  ist.

*Definition:* Eine TM  $M_0$  heißt *universell*, falls für jede 1-Band-dTM  $M$  und jedes  $x \in \{0,1\}^*$  gilt:

- $M_0$  gestartet mit  $\langle M \rangle x$  hält genau dann, wenn  $M$  gestartet mit  $x$  hält.
- Falls  $M$  gestartet mit  $x$  hält, berechnet  $M_0$  gestartet mit  $\langle M \rangle x$  den gleichen Output wie  $M$  gestartet mit  $x$ .
- Insbesondere akzeptiert  $M_0$  den Input  $\langle M \rangle x$  genau dann, wenn  $M$  den Input  $x$  akzeptiert.

*Satz:* Es gibt eine universelle 2-Band-dTM, die jede  $t(n)$ -zeit- und  $s(n)$ -platzbeschränkte 1-Band-dTM in Zeit  $O(t(n))$  auf Platz  $O(s(n))$  simuliert.

Die Länge des Programms, also die Länge von  $\langle M \rangle$ , wird in der Groß-O-Notation als Konstante angenommen.

## *Beweisidee Teil 1: Codierung einer Konfiguration*

Eine Konfiguration  $\alpha q_i X_j \beta$  von  $M$  wird von  $M_0$  wie folgt codiert:

- Auf Band 1 steht  $\langle M \rangle$  und der Zustand  $q_i$ , codiert durch  $0^i$ .
- Auf Band 2 steht die Bandinschrift  $\alpha X_j \beta$  von  $M$ .
- Der Kopf von Band 2 steht auf  $X_j$ .

Zu Beginn gilt:

- Auf Band 2 steht die Eingabe  $x$ , der Kopf befindet sich auf dem ersten Zeichen von  $x$ .
- Der Startzustand ist auf Band 1 notiert.

Am Ende gilt:

- Halte, falls keine Nachfolgekonfiguration existiert.
- Akzeptiere, falls der akzeptierende Endzustand von  $M$  auf Band 1 hinter  $\langle M \rangle$  steht.

## Beweisidee Teil 2: *Simulation eines Schrittes*

Die TM  $M$  befinde sich in der Konfiguration  $\alpha q_i X_j \beta$ :

- Suche auf Band 1 in  $\langle M \rangle$  die Zeichenreihe  $110^i 10^j$ , also den Anfang der Codierung des Übergangs  $\delta(q_i, X_j)$ .
  - $j \in \{1, 2, 3\}$  kann im Zustand gespeichert werden.
  - $0^i$  wird durch Vergleich mit der Codierung von  $q_i \hat{=} 0^i$  auf Band 1 gefunden. (Warum  $q_i$  nicht im Zustand speichern?)
- Lese die dahinterstehende Zeichenreihe der Form  $10^k 10^l 10^m$ .
  - Ersetze die Codierung  $0^i$  von  $q_i$  auf Band 1 durch  $0^k$ , also durch die Codierung des neuen Zustands  $q_k$ .
  - Speichere die Information  $l$  und  $m$  über den auszuführenden nächsten Schritt im Zustand: „Überschreibe die Zelle der Kopfposition mit  $X_l$  und bewege den Kopf gemäß  $D_m$ “.
- Verändere Band 2 entsprechend dem im Zustand gespeicherten Befehl.

*Definition:* Die **Diagonalsprache**  $DIAG$  ist wie folgt definiert:

$$DIAG = \{ \langle M \rangle \mid M \text{ ist dTM, die } \langle M \rangle \text{ nicht akzeptiert} \}$$

*Satz:*  $DIAG$  ist nicht rekursiv aufzählbar.

*Beweis* durch Widerspruch.

Angenommen, die dTM  $\bar{M}$  akzeptiert  $DIAG$ , also:  $\bar{M}$  akzeptiert  $x$  genau dann wenn  $x \in DIAG$ . Was macht  $\bar{M}$  mit Eingabe  $\langle \bar{M} \rangle$ ?

- Fall 1:
  - $\langle \bar{M} \rangle \in DIAG \Rightarrow \bar{M}$  akzeptiert die Eingabe  $\langle \bar{M} \rangle$  nicht.
  - $\bar{M}$  akzeptiert  $DIAG \Rightarrow \bar{M}$  akzeptiert  $\langle \bar{M} \rangle$ , wenn  $\langle \bar{M} \rangle \in DIAG$ ; ein Widerspruch.
- Fall 2:
  - $\langle \bar{M} \rangle \notin DIAG \Rightarrow \bar{M}$  akzeptiert die Eingabe  $\langle \bar{M} \rangle$ .
  - $\bar{M}$  akzeptiert  $DIAG \Rightarrow \bar{M}$  akzeptiert  $\langle \bar{M} \rangle$  nicht, wenn  $\langle \bar{M} \rangle \notin DIAG$ ; ein Widerspruch.

## *Explizite Darstellung des Beweises als Diagonalisierung*

Sei  $M_1, M_2, \dots$  die Folge aller in der Reihenfolge ihrer Gödelnummern aufgezählten dTMs.

Betrachte die unendliche Matrix, deren Zeilen mit  $\langle M_1 \rangle, \langle M_2 \rangle, \dots$ , und deren Spalten mit  $M_1, M_2, \dots$  nummeriert sind.

An der Position  $M_i, \langle M_j \rangle$  wird eingetragen, ob  $M_i$  die Eingabe  $\langle M_j \rangle$  akzeptiert (Eintrag „a“) oder nicht akzeptiert (Eintrag „na“).

	$M_1$	$M_2$	$M_3$	$\dots$
$\langle M_1 \rangle$	a	na	a	$\dots$
$\langle M_2 \rangle$	na	na	a	$\dots$
$\langle M_3 \rangle$	na	na	a	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Die Spalte unter  $M_i$  heißt *Akzeptanz-Folge* von  $M_i$ .

Wir nehmen wieder die Existenz einer dTM  $\bar{M}$  an, die *DIAG* akzeptiert.

	$M_1$	$M_2$	$M_3$	$\dots$		<i>DIAG</i>	$M_1$	$M_2$	$M_3$	$\dots$
$\langle M_1 \rangle$	$a$	$na$	$a$	$\dots$	$\Rightarrow$	$\langle M_1 \rangle$	$na$	$na$	$a$	$\dots$
$\langle M_2 \rangle$	$na$	$na$	$a$	$\dots$		$\langle M_2 \rangle$	$na$	$a$	$a$	$\dots$
$\langle M_3 \rangle$	$na$	$na$	$a$	$\dots$		$\langle M_3 \rangle$	$na$	$na$	$na$	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$		$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Wie sieht die Akzeptanz-Folge von  $\bar{M}$  aus?

- Definition von *DIAG*: An Stelle  $i$  hat sie den Eintrag  $a$  falls  $M_i$  die Eingabe  $\langle M_i \rangle$  nicht akzeptiert, sonst den Eintrag  $na$ .
- Somit ist  $\bar{M}$  keine der dTMs  $M_1, M_2, \dots$ , da sich die Akzeptanz-Folge von  $M_i$  an Stelle  $i$  von der von  $\bar{M}$  unterscheidet.
- Somit kann es  $\bar{M}$  nicht geben, da wir ja in  $M_1, M_2, \dots$  alle dTMs aufgezählt haben. Widerspruch!

*Halteproblem:* Stoppt ein Programm zu einer bestimmten Eingabe?

$$H = \{\langle M \rangle x \mid M \text{ ist dTM, die gestartet mit Eingabe } x \text{ hält}\}$$

*Halteproblem bei leerem Band:*

$$H_0 = \{\langle M \rangle \mid M \text{ ist dTM, die gestartet mit Input } \epsilon \text{ hält}\}$$

*Satz:*  $H$  und  $H_0$  sind rekursiv aufzählbar.

- $H$  ist rekursiv aufzählbar, da die universelle dTM  $H$  akzeptiert, falls wir sie so modifizieren, dass sie immer akzeptiert, falls sie hält.
- $H_0$  ist daher natürlich auch rekursiv aufzählbar.



Um zu zeigen, dass  $H$  und  $H_0$  nicht entscheidbar sind, nutzen wir keinen Diagonalisierungsbeweis, sondern eine andere Methode, die anwendet, dass  $DIAG$  nicht rekursiv ist.

*Vorüberlegung:*  $\overline{DIAG}$  ist nicht entscheidbar, da die Klasse der entscheidbaren Sprachen abgeschlossen ist gegenüber Komplementbildung.

$$\overline{DIAG} = \{w \in \{0,1\}^* \mid w \text{ ist keine Codierung einer TM, oder } w = \langle M \rangle \text{ und } M \text{ akzeptiert } \langle M \rangle\}$$

$\overline{DIAG}$  nennt man auch *Selbstanwendbarkeitsproblem* oder *spezielles Halteproblem*.

*Definition:* Eine Sprache  $L$  heißt reduzierbar auf  $L'$ , falls es eine totale berechenbare Funktion  $f : \Sigma^* \longrightarrow \Sigma^*$  gibt mit  $x \in L \iff f(x) \in L'$ .

- Wir schreiben dann:  $L \leq L'$  (mittels  $f$ )
- Idee:  $L$  ist nicht schwieriger zu lösen als  $L'$ .

*Satz:* Falls  $L$  nicht entscheidbar ist und  $L \leq L'$  (mittels  $f$ ) gilt, dann ist auch  $L'$  nicht entscheidbar.

*Beweis:* Wäre  $L'$  entscheidbar, dann wäre auch  $L$  entscheidbar.

- Berechne für eine beliebige Eingabe  $x$  den Wert  $f(x)$  und entscheide, ob  $f(x) \in L'$  gilt.
- Daher wäre mittels  $f$  auch entschieden, ob  $x \in L$  ist.

*Satz:*  $H$  ist nicht entscheidbar, denn es gilt  $\overline{DIAG} \leq H$  mittels der folgenden, berechenbaren Funktion  $f$ :

- Falls  $w$  keine Gödelisierung einer Turingmaschine ist:

Bilde  $w$  auf  $\langle \tilde{M} \rangle \epsilon$  ab, wobei  $\tilde{M}$  eine feste TM ist, die bei leerer Eingabe hält.

$$w \neq \langle M \rangle \wedge w \in \overline{DIAG} \iff f(w) = \langle \tilde{M} \rangle \epsilon \in H$$

- Falls  $w = \langle M \rangle$  gilt:

Bilde  $w$  auf  $\langle M' \rangle \langle M \rangle$  ab, wobei  $M'$  die dTM ist, die aus  $M$  entsteht, wenn wir jede nicht akzeptierende Rechnung von  $M$  zu einer Endlosschleife erweitern.

$$\begin{aligned} w = \langle M \rangle \wedge w \in \overline{DIAG} &\iff M \text{ akzeptiert } \langle M \rangle \\ &\iff M' \text{ gestartet mit } \langle M \rangle \text{ hält} \\ &\iff f(w) = \langle M' \rangle \langle M \rangle \in H \end{aligned}$$

*Satz:*  $H_0$  ist nicht entscheidbar, denn es gilt  $H \leq H_0$ :

- Zu  $\langle M \rangle_x$  sei  $\bar{M}(M, x)$  die TM, die gestartet mit leerem Band zuerst  $x$  schreibt, und sich dann wie  $M$  gestartet mit  $x$  verhält.
- $f$  bildet  $\langle M \rangle_x$  auf  $\langle \bar{M}(M, x) \rangle$  ab und ist berechenbar.
- Es gilt:  $M$  gestartet mit  $x$  hält  $\iff \bar{M}(M, x)$  gestartet mit leerem Band hält.

*Folgerung:* Die Klasse der rekursiv aufzählbaren Sprachen ist nicht gegen Komplementbildung abgeschlossen.

- $H$  ist rekursiv aufzählbar,  $\bar{H}$  aber nicht, denn
- wäre  $\bar{H}$  rekursiv aufzählbar, dann wäre  $H$  entscheidbar.

*Übung:* Zeigen Sie, dass die folgenden Probleme nicht entscheidbar sind.

- *Totalitätsproblem:*

$$\text{TOTAL} := \{ \langle M \rangle \mid M \text{ hält für jeden Input} \}$$

- *Endlichkeitsproblem:*

$$\text{ENDLICH} := \{ \langle M \rangle \mid M \text{ hält für endlich viele Inputs} \}$$

- *Äquivalenzproblem:*

$$\text{ÄQUIV} := \{ \langle M \rangle, \langle M' \rangle \mid L(M) = L(M') \}$$

*Totalitätsproblem:* Wir zeigen  $H_0 \leq \text{TOTAL}$ .

Sei  $\tilde{M}_M$  die Maschine, die sich wie  $M$  verhält, aber zu Beginn das Band löscht.

Sei  $\hat{M}$  eine Maschine, die nie hält.

$$f(x) := \begin{cases} \langle \tilde{M}_M \rangle, & \text{falls } x = \langle M \rangle \text{ für eine Turingmaschine } M \\ \langle \hat{M} \rangle, & \text{sonst} \end{cases}$$

$f$  ist berechenbar, und es gilt:

$$x \in H_0 \Rightarrow x = \langle M \rangle \text{ und } M \text{ hält bei leerem Input}$$

$$\Rightarrow f(x) = \langle \tilde{M}_M \rangle \in \text{TOTAL}$$

$$x \notin H_0 \stackrel{1.}{\Rightarrow} x \neq \langle M \rangle \Rightarrow f(x) = \langle \hat{M} \rangle \notin \text{TOTAL}$$

$$x \notin H_0 \stackrel{2.}{\Rightarrow} x = \langle M \rangle \text{ und } M \text{ gestartet mit } \epsilon \text{ hält nicht}$$

$$\Rightarrow f(x) = \langle \tilde{M}_M \rangle \text{ und } \tilde{M}_M \text{ hält für keine Eingabe}$$

$$\Rightarrow f(x) \notin \text{TOTAL}$$

*Äquivalenzproblem:* Wir zeigen  $H_0 \leq \text{ÄQIV}$ .

Seien  $\tilde{M}_M$  und  $\hat{M}$  wie im obigen Beweis definiert, und sei  $\bar{M}$  eine Maschine, die bei jeder Eingabe hält.

$$f(x) := \begin{cases} \langle \tilde{M}_M \rangle \langle \bar{M} \rangle, & \text{falls } x = \langle M \rangle \text{ für eine Maschine } M \\ \langle \hat{M} \rangle \langle \bar{M} \rangle, & \text{sonst} \end{cases}$$

$f$  ist berechenbar, und es gilt:

$$\begin{aligned} x \in H_0 &\Rightarrow x = \langle M \rangle \text{ und } M \text{ hält bei leerem Input} \\ &\Rightarrow f(x) = \langle \tilde{M}_M \rangle \langle \bar{M} \rangle \in \text{ÄQIV} \\ x \notin H_0 &\stackrel{1.}{\Rightarrow} x \neq \langle M \rangle \Rightarrow f(x) = \langle \hat{M} \rangle \langle \bar{M} \rangle \notin \text{ÄQIV} \\ x \notin H_0 &\stackrel{2.}{\Rightarrow} x = \langle M \rangle \text{ und } M \text{ gestartet mit } \epsilon \text{ hält nicht} \\ &\Rightarrow f(x) = \langle \tilde{M}_M \rangle \langle \bar{M} \rangle \text{ und } \tilde{M}_M \text{ hält für keine Eingabe} \\ &\Rightarrow f(x) \notin \text{ÄQIV} \end{aligned}$$

*Endlichkeitsproblem:* Wir zeigen  $\bar{H} \leq \text{GENAU-}K$ , dabei ist  
 $\text{GENAU-}K := \{\langle M \rangle \mid M \text{ hält für genau } K \text{ Eingaben}\} \subset \text{ENDLICH}$ .

Sei  $M_K$  eine fest gewählte dTM, die für genau  $K$  Eingaben hält. Sei  $M'(M, y)$  die dTM, die für genau  $K$  Eingaben hält, und sich für alle anderen Eingaben wie  $M$  gestartet mit  $y$  verhält.

$$f(x) := \begin{cases} \langle M'(M, y) \rangle & \text{falls } x = \langle M \rangle y \text{ für eine Maschine } M \\ \langle M_K \rangle & \text{sonst} \end{cases}$$

$f$  ist berechenbar, und es gilt:

$$\begin{aligned} x \in \bar{H} &\stackrel{1.}{\Rightarrow} x = \langle M \rangle y \text{ und } M \text{ gestartet mit } y \text{ hält nicht} \\ &\Rightarrow f(x) = \langle M'(M, y) \rangle \text{ hält für genau } K \text{ Eingaben} \\ &\Rightarrow f(x) \in \text{GENAU-}K \\ x \in \bar{H} &\stackrel{2.}{\Rightarrow} x \neq \langle M \rangle y \\ &\Rightarrow f(x) = \langle M_K \rangle \\ &\Rightarrow f(x) \in \text{GENAU-}K \end{aligned}$$



Fortsetzung:

$$\begin{aligned}x \notin \bar{H} &\Rightarrow x = \langle M \rangle y \text{ und } M \text{ gestartet mit } y \text{ hält} \\&\Rightarrow f(x) = \langle M'(M, y) \rangle \text{ hält für unendlich viele Eingaben} \\&\Rightarrow f(x) \notin \text{GENAU-}K\end{aligned}$$

Da  $\text{GENAU-}K \subset \text{ENDLICH}$  gilt, und da bereits  $\text{GENAU-}K$  nicht entscheidbar ist, muss auch  $\text{ENDLICH}$  nicht entscheidbar sein.

*Allgemein gilt der Satz von Rice:* Sei  $\mathcal{R}$  die Menge der von dTM berechenbaren partiellen Funktionen und  $S$  eine Teilmenge von  $\mathcal{R}$  mit  $\emptyset \neq S \neq \mathcal{R}$ . Dann ist die Sprache

$$L(S) = \{\langle M \rangle \mid M \text{ berechnet eine Funktion aus } S\}$$

nicht entscheidbar.

In anderen Worten: Aussagen über die von einer TM berechneten Funktion sind nicht entscheidbar.

## Postisches Korrespondenzproblem:

**Gegeben:** Eine endliche Menge von Wortpaaren  $(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_k, y_k)$ , wobei  $x_i, y_i \in \Sigma^+$  gilt. Diese Wortpaare sind vorstellbar als verschiedene Dominosteine.

**Gefragt:** Gibt es eine Folge von Indizes  $i_1, i_2, \dots, i_n \in \{1, 2, \dots, k\}$ ,  $n \geq 1$ , mit  $x_{i_1} x_{i_2} \dots x_{i_n} = y_{i_1} y_{i_2} \dots y_{i_n}$ ?

**Beispiel:**  $K = ((1, 101), (10, 00), (011, 11))$  besitzt die Lösung  $(1, 3, 2, 3)$ , denn es gilt

$$\underbrace{1}_{x_1} \underbrace{011}_{x_3} \underbrace{10}_{x_2} \underbrace{011}_{x_3} = \underbrace{101}_{y_1} \underbrace{11}_{y_3} \underbrace{00}_{y_2} \underbrace{11}_{y_3}$$

**Ohne Beweis:** Das Postsche Korrespondenzproblem ist nicht entscheidbar.

*Ablauf bei Reduktionen: Zeige  $ALT \leq NEU$  mittels  $f$ .*

- Erstelle zunächst eine Funktion  $f$ , die aus einer Eingabe  $x$  von  $ALT$  eine Eingabe  $f(x)$  von  $NEU$  macht, wobei gelten muss:

$$x \in ALT \iff f(x) \in NEU$$

- Wäre  $NEU$  berechenbar, dann wäre auch  $ALT$  berechenbar.

$$x \stackrel{?}{\in} ALT \rightsquigarrow \begin{cases} f(x) \in NEU & \Rightarrow x \in ALT \\ f(x) \notin NEU & \Rightarrow x \notin ALT \end{cases}$$

Um  $x \stackrel{?}{\in} ALT$  zu beantworten, nutzen wir also  $f$  und den (fiktiven) Algorithmus für  $NEU$ .

## *Einleitung*

- Bewertung von Algorithmen
- Berechenbarkeitstheorie
- *Komplexitätstheorie*
- Exakte Algorithmen für schwere Probleme

## *Komplexitätsklassen*

- P vs. NP: Probleme, für die eine polynomiell zeitbeschränkte dTM bzw. nTM existiert.
- L vs. NL: Probleme, für die eine logarithmisch platzbeschränkte dTM bzw. nTM existiert.
- PSPACE vs. NPSPACE: Probleme, für die eine polynomiell platzbeschränkte dTM bzw. nTM existiert.
- Es gibt viele weitere Komplexitätsklassen.

Es gilt folgende Hierarchie:

$$L \subset NL \subset P \subset NP \subset PSPACE \subset NPSPACE$$

**CLIQUE:** Finde zu einem gegebenen Graphen  $G = (V, E)$  und einem  $k \in \mathbb{N}$  eine Teilmenge  $V' \subseteq V$  der Knoten, sodass je zwei Knoten in  $V'$  durch eine Kante in  $E$  verbunden sind und  $|V'| \geq k$  gilt.

*Algorithmus:*

- „Rate“ nichtdeterministisch eine Menge  $V' \subseteq V$  mit  $|V'| \geq k$
- und verifiziere, ob  $V'$  eine Clique ist.

**HAMILTON-KREIS:** Finde eine Route in einem gegebenen Graphen  $G = (V, E)$ , die genau einmal durch jeden Knoten in  $V$  läuft und wieder am Startknoten endet.

*Algorithmus:*

- „Rate“ nichtdeterministisch eine Menge  $E' \subseteq E$
- und verifiziere, ob  $E'$  eine Rundreise ist.

**INDEPENDENT SET:** Finde in einem gegebenen Graphen  $G = (V, E)$  eine Menge  $V' \subseteq V$  mit  $|V'| \geq k$ , sodass für alle  $u, v \in V'$  gilt:  $\{u, v\} \notin E$ .

- „Rate“ nichtdeterministisch eine Menge  $V' \subseteq V$  mit  $|V'| \geq k$
- und verifiziere, ob  $V'$  eine unabhängige Menge ist.

**DOMINATING SET:** Finde in einem gegebenen Graphen  $G = (V, E)$  eine Menge  $V' \subseteq V$  mit  $|V'| \leq k$ , sodass für alle  $v \in V$  der Knoten  $v$  oder einer seiner Nachbarn in  $V'$  enthalten ist.

- „Rate“ nichtdeterministisch eine Menge  $V' \subseteq V$  mit  $|V'| \leq k$
- und verifiziere, ob  $V'$  eine dominierende Menge ist.

**Ohne Beweis:** Der Nichtdeterminismus kann auf eine initiale „Ratephase“ beschränkt werden.

Die obigen Probleme können deterministisch sehr einfach, aber (bisher) nicht effizient gelöst werden:

- Zähle alle  $\binom{n}{k}$  Teilmengen  $V'$  mit  $|V'| = k$  auf und teste die gesuchte Eigenschaft.  
Zur Erinnerung:

$$\begin{aligned}\binom{n}{k} &= \frac{n!}{k! \cdot (n-k)!} = \frac{n \cdot (n-1) \cdot \dots \cdot (n-k+1)}{k!} \\ &\leq \frac{n \cdot n \cdot \dots \cdot n}{k!} \leq n^k\end{aligned}$$

- Laufzeit  $\mathcal{O}(n^k \cdot (n+m))$  ist für konstantes  $k$  polynomiell.

Dabei ist  $n$  die Anzahl der Knoten und  $m$  die Anzahl der Kanten.

*Frage:* Warum wird zusätzlich zu den Graphen auch die Größe  $k$  in der Eingabe angegeben?



Wir unterscheiden bei vielen Problemen die Entscheidungs-, Optimierungs- und Suchvariante, hier am Beispiel **CLIQUE**.

- **Entscheidungsvariante:** Gibt es in einem gegebenen Graphen  $G = (V, E)$  eine Clique der Größe mindestens  $k$ ? Dabei ist  $k$  ein Teil der Eingabe.
- **Optimierungsversion:** Aus wie vielen Knoten besteht eine Clique maximaler Größe im gegebenen Graphen  $G = (V, E)$ ?
- **Suchvariante:** Finde eine Knotenmenge  $V' \subseteq V$  des Graphen  $G = (V, E)$ , sodass  $V'$  eine Clique maximaler Größe ist.

Problem bei der Optimierungs- und der Suchvariante: Es reicht nicht, die „geratene“ Lösung zu verifizieren. Es muss zusätzlich noch sichergestellt sein, dass es keine größere Clique im Graphen gibt.

Siehe auch <http://www.csc.kth.se/~viggo/wwwcompendium/> für weitere schwere Optimierungsprobleme.

*Frage:* Wie schwer sind die verschiedenen Varianten in Bezug zueinander?

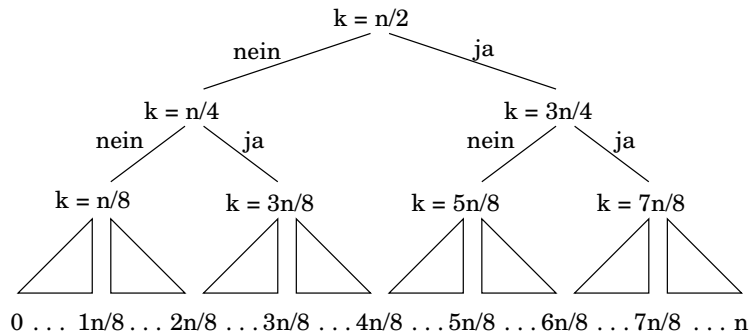
- Wenn wir einen Algorithmus für DEC-CLIQUE kennen, können wir damit eine Lösung für MAX-CLIQUE berechnen?
- Können wir mit einem Algorithmus für MAX-CLIQUE auch eine Lösung für SEARCH-CLIQUE berechnen?

DEC-CLIQUE bezeichnet die Entscheidungs-, MAX-CLIQUE die Optimierungs- und SEARCH-CLIQUE die Suchvariante.

Die anderen Richtungen sind klar:

- Ein Algorithmus für MAX-CLIQUE löst auch DEC-CLIQUE: Falls  $k$  kleiner oder gleich der optimalen Lösung ist, dann gib ja aus, sonst nein.
- Ein Algorithmus für SEARCH-CLIQUE löst auch MAX-CLIQUE: Gib die Größe der berechneten Teilmenge aus.

Mit einem Algorithmus für DEC-CLIQUE können wir auch eine Lösung für MAX-CLIQUE berechnen, indem wir einen Entscheidungsbaum wie bei der binären Suche nutzen:



Dabei bezeichne  $n$  die Anzahl der Knoten des Graphen.

Die Laufzeit gegenüber der Lösung des Entscheidungsproblems erhöht sich um einen logarithmischen Faktor.

Mit einem Algorithmus für MAX-CLIQUE können wir auch eine Lösung für SEARCH-CLIQUE berechnen:

```
function SEARCH-CLIQUE( $G = (V, E)$ )  
   $opt := \text{MAX-CLIQUE}(G)$   
   $V' := \emptyset$   
  for all  $v \in V$  do  
     $G' := G - \{v\}$   
    if  $\text{MAX-CLIQUE}(G') < opt$  then  
       $V' := V' \cup \{v\}$   
    else  
       $G := G'$   
  return  $V'$ 
```

$G - \{v\}$  bezeichnet den Graphen, der aus  $G$  entsteht, indem man den Knoten  $v$  und alle dazu inzidenten Kanten aus  $G$  entfernt.

Die Laufzeit erhöht sich um den Faktor  $n$ .

**$k$ -COLORING:** Sei  $f : V \rightarrow \{1, 2, \dots, k\}$  eine Abbildung der Knoten auf die ersten  $k$  natürlichen Zahlen. Dann nennt man  $f$  eine  **$k$ -Färbung** ( $k$ -coloring) eines Graphen  $G = (V, E)$ , wenn  $f(u) \neq f(v)$  für alle Kanten  $\{u, v\} \in E$  gilt.

- „Rate“ nichtdeterministisch für den Graphen  $G = (V, E)$  eine Zuordnung  $f : V \rightarrow [k]$  von Farben zu Knoten, wobei  $[k] := \{1, 2, \dots, k\}$  ist.
- Prüfe für jede Kante  $\{u, v\} \in E$ , ob  $f(u) \neq f(v)$  gilt.

Bei diesem Problem muss die Zahl  $k$  nicht Teil der Eingabe sein.

- Die obigen Probleme CLIQUE, INDEPENDENT SET und DOMINATING SET konnten für konstantes  $k$  in polynomieller Zeit  $\mathcal{O}(n^k)$  gelöst werden.
- Für das Problem  $k$ -COLORING ist bereits für  $k = 3$  kein Algorithmus mit polynomieller Laufzeit bekannt, jedenfalls nicht für allgemeine Graphen.

Varianten des Färbungsproblems:

- **Entscheidungsvariante:** Gibt es für einen gegebenen Graphen  $G = (V, E)$  eine Färbung der Knoten mit höchstens  $k$  Farben? Dabei ist  $k$  Teil der Eingabe.
- **Optimierungsversion:** Wie viele Farben reichen aus, um den Graphen zu färben?
- **Suchvariante:** Bestimme eine Abbildung  $f : V \rightarrow [k]$ , sodass  $f$  eine  $k$ -Färbung von  $G$  ist und keine Färbung mit weniger Farben existiert.

Drei Varianten können einfach ineinander umgerechnet werden.

- $\text{DEC-COL} \rightarrow \text{MIN-COL}$ : Nutze binären Entscheidungsbaum.
- $\text{MIN-COL} \rightarrow \text{DEC-COL}$ : Falls  $k$  größer oder gleich der optimalen Lösung ist, dann gib ja aus, sonst nein.
- $\text{SEARCH-COL} \rightarrow \text{MIN-COL}$ : Gib die Anzahl  $k$  der genutzten Farben aus.

Die Umrechnung von  $\text{MIN-COL} \rightarrow \text{SEARCH-COL}$  ist deutlich komplizierter.

## MIN-COL $\rightarrow$ SEARCH-COL

- Sei  $G = (V, E)$  ein Graph mit den Knoten  $V = \{v_1, \dots, v_n\}$ .
- Berechne zunächst das minimale  $k$ , für das eine  $k$ -Färbung von  $G$  existiert.
- Zum Bestimmen einer  $k$ -Färbung nutzen wir eine  $k$ -Clique  $C$ , die die Knoten  $x_1, \dots, x_k$  enthält.
- $C$  kann mit  $k$  vielen Farben gefärbt werden, wobei o.B.d.A. für die Färbung  $f$  angenommen werden kann, dass  $f(x_i) = i$  gilt.
- Der initiale Graph  $G_0$  besteht aus der Vereinigung von  $G$  und  $C$ .
- Berechne  $G_i$  aus  $G_{i-1}$  wie folgt:
  - Für  $j := 1 \dots k$  führe die folgenden zwei Schritte aus:
  - $E(G_i) := E(G_{i-1}) \cup \{(v_i, x) \mid x \in \{x_1, \dots, x_k\} - \{x_j\}\}$
  - Teste mittels MIN-COL, ob  $G_i$  immer noch  $k$ -färbbar ist. Falls ja, dann ist  $G_i$  gefunden, sonst fahre mit dem nächsten Wert von  $j$  fort.

Die Färbung können wir aus  $G_n$  ablesen: Knoten  $v_i$  erhält die Farbe  $j$ , wenn  $v_i$  nicht mit dem Knoten  $x_j$  durch eine Kante verbunden ist.

Für das Problem des Handlungsreisenden kann man auch eine Entscheidungs-, Optimierungs- und Suchvariante definieren.

Auch hier ist es recht einfach möglich, mittels eines Algorithmus für die eine Variante die Lösung einer anderen Variante zu berechnen.

- DEC-TSP  $\rightarrow$  MIN-TSP: Nutze binären Entscheidungsbaum.
- MIN-TSP  $\rightarrow$  DEC-TSP: Falls  $k$  größer oder gleich der optimalen Lösung ist, dann gib ja aus, sonst nein.
- SEARCH-TSP  $\rightarrow$  MIN-TSP: Addiere die Werte aller Kanten, die auf der optimalen Tour liegen und gib diese Summe aus.
- MIN-TSP  $\rightarrow$  SEARCH-TSP: Erhöhe den Wert einer Kante  $e$  um einen Wert  $c$ . Falls die optimale Länge unverändert ist, gehört die Kante  $e$  nicht zur optimalen Tour. Falls doch, setze den Wert der Kante zurück. Teste auf diese Weise alle Kanten.



*Definition:* Eine Sprache  $L$  ist *polynomiell reduzierbar* auf eine Sprache  $L'$ , wenn eine Funktion  $f$  existiert, mit

- $x \in L \iff f(x) \in L'$
- und  $f$  ist in polynomieller Zeit berechenbar.

Analog zur Berechenbarkeit schreiben wir dann  $L \leq_p L'$  und sagen:  $L$  ist nicht schwieriger zu berechnen als  $L'$ .

*Beachte:* Wir haben Probleme immer als Sprachen definiert, die von Turingmaschinen akzeptiert werden.

$\text{CLIQUE} := \{(G, k) \mid \text{Graph } G \text{ enthält Clique } C \text{ mit } |C| \geq k\}$

$\text{HCP} := \{G \mid \text{Graph } G \text{ enthält einen Hamilton-Kreis}\}$

$\text{ISP} := \{(G, k) \mid G \text{ enthält Independent-Set } I \text{ mit } |I| \geq k\}$

$\text{DSP} := \{(G, k) \mid G \text{ enthält Dominating-Set } D \text{ mit } |D| \leq k\}$

*Hinweis:* Polynomielle Reduktion ist transitiv.

$$L_1 \leq_p L_2 \text{ und } L_2 \leq_p L_3 \Rightarrow L_1 \leq_p L_3$$

*Idee:* Sei  $L_1 \leq_p L_2$  mittels  $f$  in Zeit  $p$ , und sei  $L_2 \leq_p L_3$  mittels  $g$  in Zeit  $q$ . Dann gilt  $L_1 \leq_p L_3$  mittels  $g \circ f$ :

$$x \in L_1 \iff f(x) \in L_2 \iff g(f(x)) \in L_3$$

$g(f(x))$  ist in polynomieller Zeit berechenbar:

- $f(x)$  hat polynomielle Länge  $p(|x|)$ , da eine Turingmaschine in Zeit  $p(|x|)$  nur  $p(|x|)$  Zeichen schreiben kann.
- Also kann  $g(f(x))$  in Zeit  $p(|x|) + q(p(|x|))$  berechnet werden, also in polynomieller Zeit.

*Definition:* Ein Problem  $L$  ist NP-vollständig, wenn gilt:

- Es existiert eine nTM, die das Problem in polynomieller Zeit löst, also  $L \in \text{NP}$ .
- Alle  $L' \in \text{NP}$  sind polynomiell reduzierbar auf  $L$ , also  $L' \leq_p L$ .

*Satz:* Ist auch nur ein einziges NP-vollständiges Problem deterministisch in polynomieller Zeit lösbar, dann sind alle NP-vollständigen Probleme in polynomieller Zeit lösbar!

*Beweis:*  $\rightarrow$  Übung

Leider ist bis heute für kein einziges NP-vollständiges Problem ein polynomiell zeitbeschränkter (deterministischer) Algorithmus bekannt.

Falls auch nur ein einziges NP-vollständiges Problem  $L$  in  $P$  liegt, dann gilt  $P = NP$ .

*Idee:* Sei  $L$  ein NP-vollständiges Problem und sei  $L$  in polynomieller Zeit  $p_1$  mittels einer deterministischen Turingmaschine  $M$  entscheidbar.

Für jedes NP-vollständige Problem  $L'$  gilt  $L' \leq_p L$  mittels einer Funktion  $f$  in Zeit  $p_2$  nach Definition, da  $L$  NP-vollständig ist.

Also kann  $x \in L'$  in polynomieller Zeit  $p_1 + p_2$  mittels der Maschine  $M$  entschieden werden:

$$x \in L' \iff f(x) \in L$$

*Frage:* Wie findet man ein erstes NP-vollständiges Problem?

*Erfüllbarkeitsproblem (SATisfiability):*

- Gegeben: Eine Formel  $F$  der Aussagenlogik.
- Gefragt: Ist die Formel  $F$  erfüllbar?
- $\text{SAT} := \{F \mid F \text{ ist erfüllbare aussagenlogische Formel}\}$

*Satz von Cook*<sup>2</sup>: SAT ist NP-vollständig

Wir werden im folgenden zeigen:

- $\text{SAT} \in \text{NP}$
- $\forall L \in \text{NP} : L \leq_p \text{SAT}$

---

<sup>2</sup>aus dem Buch von Uwe Schöning: Theoretische Informatik – kurz gefasst.

## SAT $\in$ NP

- Maschine  $M$  läuft einmal über die Eingabe und stellt fest, welche Variablen in der Formel  $F$  vorkommen – dies seien  $x_1, \dots, x_k$ .
- In der eigentlichen nichtdeterministischen Phase „rät“ die Maschine  $M$  eine Belegung  $a_1, \dots, a_k$  mit  $a_i \in \{0, 1\}$ .  
Zu diesem Zeitpunkt existieren  $2^k$  mögliche unabhängige Rechnungen.
- $M$  berechnet den Wert von  $F$  unter der betreffenden Belegung und akzeptiert genau dann, wenn der Wert 1 ist.
- $F \in \text{SAT}$  genau dann, wenn es eine Rechnung von  $M$  gibt, die  $F$  akzeptiert.
- Da  $k \leq |F|$  ist, ist die Rechenzeit polynomial.

$\forall L \in NP : L \leq_p \text{SAT}:$

- Sei  $M$  eine nTM, die  $L$  in polynomieller Zeit entscheidet.
- Sei  $\Gamma = \{a_1, \dots, a_\ell\}$  das Arbeitsalphabet von  $M$ .
- Sei  $Z = \{z_1, \dots, z_k\}$  die Zustandsmenge von  $M$ .
- Die  $\delta$ -Relation von  $M$  enthalte die Zeile  $\delta(z_e, a) \ni (z_e, a, N)$ .  
→ Ein einmal erreichter Endzustand wird also nie mehr verlassen.
- Sei  $p$  ein Polynom, das die Rechenzeit von  $M$  beschränkt.
- Sei  $x = x_1x_2 \dots x_n \in \Sigma^*$  eine Eingabe für  $M$ .

Wir werden nun eine Boolesche Formel  $F$  in Abhängigkeit von  $x$  angeben, sodass gilt:

$$x \in L \iff F(x) \text{ ist erfüllbar}$$

Die gesuchte Formel  $F$  enthält folgende Boolesche Variablen:

Variable	Indizes	Bedeutung
$zust_{t,z}$	$t = 0, \dots, p(n)$ und $z \in Z$	$zust_{t,z} = 1 \iff$ nach $t$ Schritten befindet sich $M$ im Zustand $z$
$pos_{t,i}$	$t = 0, \dots, p(n)$ und $i = -p(n), \dots, p(n)$	$pos_{t,i} = 1 \iff$ der Schreib-Lesekopf von $M$ befindet sich nach $t$ Schritten auf Position $i$
$band_{t,i,a}$	$t = 0, \dots, p(n)$ und $i = -p(n), \dots, p(n)$ und $a \in \Gamma$	$band_{t,i,a} = 1 \iff$ nach $t$ Schritten befindet sich auf Bandposition $i$ das Zeichen $a$

Dabei ist  $n$  die Länge der Eingabe und  $p$  das Polynom, das die Rechenzeit von  $M$  beschränkt.



Die Formel  $F$  ist aus mehreren Teilformeln aufgebaut:

$$F = R \wedge A \wedge U_1 \wedge U_2 \wedge E$$

- $R$ : Randbedingungen
- $A$ : Anfangsbedingung
- $U_1$  und  $U_2$ : Übergangsbedingungen
- $E$ : Endbedingungen

Im Folgenden wird immer wieder die Teilformel  $G(y_1, \dots, y_k)$  vorkommen:

$$G(y_1, \dots, y_k) = \left( \bigvee_{i=1}^k y_i \right) \wedge \left( \bigwedge_{i \neq j} (\neg y_i \vee \neg y_j) \right)$$

wird genau dann wahr, wenn genau eins der  $y_i$  wahr ist.

Randbedingung  $R$  drückt aus:

- $M$  ist zu jedem Zeitpunkt  $t$  in genau einem Zustand:

$$\bigwedge_t G(\text{zust}_{t,z_1}, \dots, \text{zust}_{t,z_k})$$

- Der Kopf von  $M$  befindet sich zu jedem Zeitpunkt  $t$  an genau einer Bandposition:

$$\bigwedge_t G(\text{pos}_{t,-p(n)}, \dots, \text{pos}_{t,p(n)})$$

- Zu jedem Zeitpunkt  $t$  und an jeder Bandposition  $i$  steht genau ein Zeichen:

$$\bigwedge_{t,i} G(\text{band}_{t,i,a_1}, \dots, \text{band}_{t,i,a_\ell})$$

Anfangsbedingung  $A$  drückt für  $t = 0$  aus:

- $M$  ist im Anfangszustand  $z_0$ .
- Der Schreib-Lesekopf befindet sich auf dem ersten Zeichen.
- Die ersten  $n$  Bandpositionen enthalten die Eingabe  $x_1, \dots, x_n$ .
- An allen anderen Bandpositionen steht ein Blank.

$$\begin{aligned} A = & \text{zust}_{0,z_0} \wedge \text{pos}_{0,1} \wedge \bigwedge_{j=1}^n \text{band}_{0,j,x_j} \\ & \wedge \bigwedge_{j=-p(n)}^0 \text{band}_{0,j,\square} \wedge \bigwedge_{j=n+1}^{p(n)} \text{band}_{0,j,\square} \end{aligned}$$

$U_1$  beschreibt den Übergang vom Zeitpunkt  $t$  nach  $t + 1$  an der Stelle, wo sich der Schreib-Lesekopf befindet:

$$U_1 = \bigwedge_{t,z,i,a} \left[ (zust_{t,z} \wedge pos_{t,i} \wedge band_{t,i,a}) \rightarrow \bigvee_{\delta(z,a) \ni (z',a',y)} (zust_{t+1,z'} \wedge pos_{t+1,i+y} \wedge band_{t+1,i,a'}) \right]$$

wobei  $y \in \{-1, 0, +1\}$  angenommen wird.

$U_2$  drückt aus, dass sich der Bandinhalt auf allen anderen Positionen nicht ändert:

$$U_2 = \bigwedge_{t,i,a} \left( (\neg pos_{t,i} \wedge band_{t,i,a}) \rightarrow band_{t+1,i,a} \right)$$

$E$  prüft, ob ein Endzustand erreicht wird. Nach unserer Annahme wird dies insbesondere zum Zeitpunkt  $p(n)$  erreicht.

$$E = \bigvee_{z \in E} \text{zust}_{p(n),z}$$

*Übung:*

- Zeigen Sie:  $x \in L \iff F(x)$  ist erfüllbar
- Zeigen Sie, dass alle Teilfolgen polynomielle Länge haben und daher die Reduktion in polynomieller Zeit berechenbar ist.

$x \in L \iff F(x)$  ist erfüllbar

- $x \in L$ : Dann gibt es eine Rechnung polynomieller Länge  $p(|x|)$  von einer nicht-deterministischen Maschine, und die Rechnung führt in einen akzeptierenden Endzustand.

Wenn alle Variablen entsprechend der Intention gesetzt werden, dann ist  $F(x)$  erfüllt.

- $F(x)$  ist erfüllbar: Dann ist durch eine erfüllende Belegung insbesondere die Teilformel  $R$  für die Randbedingungen erfüllt.

Daher sind die Variablenwerte für  $zust_{t,z}$ ,  $pos_{t,i}$  und  $band_{t,i,a}$  sinnvoll als Konfiguration einer Maschine  $M$  interpretierbar.

Da die Belegung auch die Teilformeln für die Anfangs-, Übergangs- und Endbedingung erfüllt, wird eine Berechnung der Länge  $p(|x|)$  beschrieben, die in einem akzeptierenden Endzustand endet. Daher gilt  $x \in L(M)$ .

Für die Länge der Teilformel  $G(y_1, \dots, y_k)$  gilt  $|G| \in \mathcal{O}(k^2)$ , also ist die Länge quadratisch in der Anzahl der Variablen:

$$G(y_1, \dots, y_k) = \left( \bigvee_{i=1}^k y_i \right) \wedge \left( \bigwedge_{i \neq j} (\neg y_i \vee \neg y_j) \right)$$

Wegen  $t = 0, \dots, p(n)$  und  $i = -p(n), \dots, +p(n)$  und aufgrund obiger Beobachtung gilt für die Randbedingungen:

- $|\bigwedge_t G(\text{zust}_{t,z_1}, \dots, \text{zust}_{t,z_k})| \in \mathcal{O}(p \cdot k^2)$
- $|\bigwedge_t G(\text{pos}_{t,-p(n)}, \dots, \text{pos}_{t,p(n)})| \in \mathcal{O}(p^3)$
- $|\bigwedge_{t,i} G(\text{band}_{t,i,a_1}, \dots, \text{band}_{t,i,a_\ell})| \in \mathcal{O}(p^2 \cdot \ell^2)$

Die Endbedingung hat konstante Länge:

$$|\bigvee_{z \in E} \text{zust}_{p(n),z}| \in \mathcal{O}(1)$$

Die Länge der Übergangsbedingungen ist polynomiell beschränkt, und wegen  $t = 0, \dots, p(n)$  und  $i = -p(n), \dots, p(n)$  gilt:

- $\left| \bigwedge_{t,z,i,a} [(zust_{t,z} \wedge pos_{t,i} \wedge band_{t,i,a}) \rightarrow \bigvee_{\delta(z,a) \ni (z',a',y)} (zust_{t+1,z'} \wedge pos_{t+1,i+y} \wedge band_{t+1,i,a'})] \right| \in \mathcal{O}(p^2)$
- $\left| \bigwedge_{t,i,a} ((\neg pos_{t,i} \wedge band_{t,i,a}) \rightarrow band_{t+1,i,a}) \right| \in \mathcal{O}(p^2)$

Und schließlich stellen wir noch fest, dass auch die Anfangsbedingung polynomiell lang ist:

$$\begin{aligned} & |zust_{0,z_0} \wedge pos_{0,1} \wedge \bigwedge_{j=1}^n band_{0,j,x_j} \\ & \wedge \bigwedge_{j=-p(n)}^0 band_{0,j,\square} \wedge \bigwedge_{j=n+1}^{p(n)} band_{0,j,\square}| \in \mathcal{O}(p) \end{aligned}$$



*Uns interessiert:* Was macht ein Problem schwierig?

- Wir beschränken uns auf Formeln in konjunktiver Normalform:  
Ist **KNF-SAT** auch NP-vollständig?
- Wir beschränken uns auf Formeln in disjunktiver Normalform:  
Ist **DNF-SAT** auch NP-vollständig?
- Wir beschränken die Anzahl der Variablen pro Klausel auf 3:  
Ist **3KNF-SAT** auch NP-vollständig?
- Wir beschränken die Anzahl der Variablen pro Klausel weiter:  
Ist **2KNF-SAT** auch NP-vollständig?
- Wir erlauben in jeder Klausel nur eine positive Variable:  
Ist **HORN-SAT** auch NP-vollständig?

Ist **KNF-SAT** auch NP-vollständig?

- Jede Boolesche Formel ist äquivalent in konjunktive Normalform umformbar.
- Aber: Das Verfahren hat exponentiellen Aufwand.

Wir zeigen:  $\text{SAT} \leq_p 3\text{KNF-SAT}$

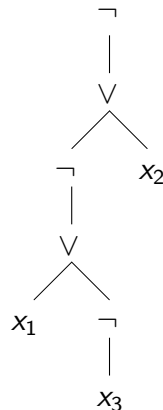
- Wir geben ein polynomielles Verfahren an, das beliebige Boolesche Formeln  $F$  umwandelt in  $F'$ . Dabei ist  $F'$  in konjunktiver Normalform mit höchstens 3 Literalen pro Klausel und es gilt:

$$F \text{ ist erfüllbar} \iff F' \text{ ist erfüllbar}$$

- Es ist nur Erfüllbarkeitsäquivalenz zwischen  $F$  und  $F'$  verlangt, nicht Äquivalenz im strengen Sinne.

Wir stellen uns eine Formel als Baumstruktur vor.

Beispiel:  $F = \neg(\neg(x_1 \vee \neg x_3) \vee x_2)$

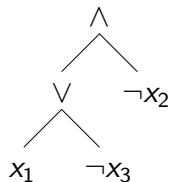


*1. Schritt:* Wir wenden deMorgan's Regeln an und bringen alle Negationszeichen zu den Blättern.

Dabei ändert sich evtl.  $\vee$  zu  $\wedge$  und umgekehrt. Dieser Schritt erfordert nur einen Durchlauf über die Formel.

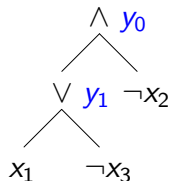
# KNF-SAT ist NP-vollständig

Nach dem ersten Schritt kommen im Innern des Baumes nur  $\vee$  und  $\wedge$  vor und die Blätter sind nur mit negierten oder unnegierten Variablen beschriftet.

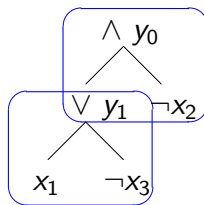


*2. Schritt:* Wir ordnen jedem inneren Knoten eine neue Variable aus  $\{y_0, y_1, \dots\}$  zu.

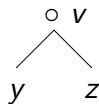
Der Baumwurzel wird  $y_0$  zugewiesen.



3. *Schritt*: Wir fassen jede Verzweigung zu einer Dreier-Gruppe zusammen:



Jede Verzweigung der Bauart



mit  $\circ \in \{\wedge, \vee\}$  ordnen wir eine Teilformel der Form

$$(v \leftrightarrow (y \circ z))$$

zu. Alle diese Formeln werden mit  $\wedge$  verknüpft, zusätzlich kommt die Teilformel  $y_0$  hinzu. Dadurch erhalten wir die Formel  $F_1$ :

$$F_1 = (y_0) \wedge (y_0 \leftrightarrow (y_1 \wedge \neg x_2)) \wedge (y_1 \leftrightarrow (x_1 \vee \neg x_3))$$

4. *Schritt*: Jede der Teilformeln wird separat in konjunktive Normalform umgeformt:

- In jeder Teilformel kommen nur 3 Literale vor.
- Der exponentielle Aufwand für das Umformen in konjunktive Normalform spielt keine Rolle mehr, da die Teilformeln nur konstante Größe haben.

Umwandeln der Teilformeln:

$$(a \leftrightarrow (b \vee c)) \mapsto (a \vee \neg b) \wedge (\neg a \vee b \vee c) \wedge (a \vee \neg c)$$

$$(a \leftrightarrow (b \wedge c)) \mapsto (\neg a \vee b) \wedge (\neg a \vee c) \wedge (a \vee \neg b \vee \neg c)$$

Alle Umformungsschritte können mit polynomielltem Aufwand erfolgen.

Daher ist  $\text{SAT} \leq_p 3\text{KNF-SAT}$  gezeigt!

3KNF-SAT ist NP-vollständig, also auch KNF-SAT, da die Einschränkung auf 3 Literale pro Klausel ja gerade ein Spezialfall von KNF-SAT ist.

*Übung:* 2KNF-SAT ist in P!

Sei  $F$  eine 2KNF-Formel über den Variablen  $x_1, \dots, x_n$ . Dann definieren wir einen Graphen  $G = (V, E)$  mit

- $V = \{x_1, \neg x_1, x_2, \neg x_2, \dots, x_n, \neg x_n\}$  und
- $E = \{(\neg x_i, x_j) \mid (x_i \vee x_j) \in F\} \cup \{(\neg x_j, x_i) \mid (x_i \vee x_j) \in F\}$

*Zeigen Sie:*  $F$  ist genau dann nicht erfüllbar, wenn der Graph einen Kreis enthält, auf dem sowohl  $x_i$  als auch  $\neg x_i$  liegt.

*Hinweis:* Die Klausel  $(x_i \vee x_j)$  kann nur dann erfüllt sein, wenn  $x_i$  oder  $x_j$  mit 1 belegt sind, oder anders ausgedrückt:  $\neg x_i \Rightarrow x_j$  oder  $\neg x_j \Rightarrow x_i$  muss gelten, beides ist äquivalent zu  $x_i \vee x_j$ .

$F$  ist nicht erfüllbar  $\iff$  Graph  $G$  enthält einen Kreis, auf dem sowohl  $x_i$  als auch  $\neg x_i$  liegen.

$\Leftarrow$  Wenn es einen solchen Kreis gibt, dann muss es irgendwo eine Implikation geben, die nicht erfüllt ist:

$$\begin{array}{ccccccc} 0 & & 0 \rightarrow 1 & & 1 & & 1 \rightarrow 0 \text{ (rot)} \\ x & \rightarrow & \dots\dots\dots & \rightarrow & \neg x & \rightarrow & \dots\dots\dots \rightarrow x \end{array}$$

$\Rightarrow$  Wenn kein solcher Kreis existiert, dann liefert folgender Algorithmus eine erfüllende Belegung:

- ① wähle ein noch nicht zugewiesenes Literal  $\alpha$ , wobei kein Weg von  $\alpha$  zu  $\neg\alpha$  existiert. Hinweis:  $\alpha$  ist ein Literal und kann somit auch eine negierte Variable  $\neg x$  sein.
- ② (a) Weise  $\alpha$  den Wert 1 zu, Weise allen von  $\alpha$  aus erreichbaren Knoten ebenfalls 1 zu. (b) Weise den entsprechenden negierten Literalen 0 zu.
- ③ Wiederhole die Schritte 1 und 2 solange, bis alle Literale mit Werten belegt sind.



Wir stellen zunächst fest: Wenn es in  $G$  einen Weg von  $\alpha$  nach  $\beta$  gibt, dann gibt es auch einen Weg von  $\neg\beta$  nach  $\neg\alpha$ .

- Wenn es eine Kante  $x \rightarrow y \equiv \neg x \vee y$  gibt, dann gibt es auch eine Kante  $\neg y \rightarrow \neg x \equiv \neg(\neg y) \vee \neg x \equiv \neg x \vee y$ .
- Wenn es einen Weg  $\alpha \rightarrow p_1 \rightarrow \dots \rightarrow p_k \rightarrow \beta$  gibt, dann gibt es auch einen Weg  $\neg\beta \rightarrow \neg p_k \rightarrow \dots \rightarrow \neg p_1 \rightarrow \neg\alpha$ .

noch zu zeigen:

- Ein Literal  $\alpha$  aus Schritt 1 kann immer gefunden werden:

Falls für ein Literal  $\beta$  ein Weg nach  $\neg\beta$  existiert, dann existiert von  $\alpha := \neg\beta$  kein Weg nach  $\neg(\neg\beta) = \neg\alpha$  aufgrund der Voraussetzung, dass  $\beta$  und  $\neg\beta$  auf keinem Kreis liegen.

noch zu zeigen:

- Es gibt keine Konflikte mit bereits belegten Literalen und die gefundene Belegung ist eine erfüllende Belegung.

Ein Problem könnte es nur geben, falls in einem Schritt Literale mit 0 belegt werden

$$\overset{0}{\alpha} \rightarrow \dots \rightarrow \overset{0}{\beta},$$

und in einem späteren Schritt Literale mit 1 belegt werden,

$$\overset{1}{\gamma} \rightarrow \dots \rightarrow \overset{1}{\delta} \xrightarrow{\text{⚡}} \overset{0}{\alpha} \rightarrow \dots \rightarrow \overset{0}{\beta}$$

sodass Implikationen nicht erfüllt sind. Das kann aber nicht sein, da  $\alpha := 0$  nur in Schritt (2b) gesetzt würde, aber  $\neg\alpha \rightarrow \neg\delta \rightarrow \dots \rightarrow \neg\gamma$  existiert und daher  $\neg\alpha$  und seine Nachfolger  $\neg\delta, \dots, \neg\gamma$  in Schritt (2a) auf 1 gesetzt würde.

Eine Formel  $F$  in konjunktiver Normalform ist eine Hornformel, wenn jede Klausel in  $F$  höchstens ein positives Literal enthält.

*Beispiel:*

$$F = (a \vee \neg b) \wedge (\neg c \vee \neg a \vee d) \wedge (\neg a \vee \neg b) \wedge d \wedge \neg e$$

Hornformeln können anschaulich als Konjunktionen von Implikationen geschrieben werden.

$$F \equiv (b \rightarrow a) \wedge (c \wedge a \rightarrow d) \wedge (a \wedge b \rightarrow 0) \wedge (1 \rightarrow d) \wedge (e \rightarrow 0)$$

Machen Sie sich klar, dass die obigen Formeln äquivalent sind.

*Übung:* HORN-SAT ist in P!

Sei  $F$  eine Hornformel über den atomaren Formeln  $x_1, \dots, x_n$ .

- ① Für alle Teilformeln  $1 \rightarrow x_i$  aus  $F$  markiere Literal  $x_i$ .
- ② Wiederhole:
  - Markiere  $y$ , falls es eine Teilformel  $(x_{i_1} \wedge \dots \wedge x_{i_k}) \rightarrow y$  gibt, bei der  $x_{i_1}, \dots, x_{i_k}$  bereits markiert sind.
  - Falls es eine Teilformel  $(x_{i_1} \wedge \dots \wedge x_{i_k}) \rightarrow 0$  gibt, bei der  $x_{i_1}, \dots, x_{i_k}$  bereits markiert sind, gib *unerfüllbar* aus und stoppe.
- ③ Gib *erfüllbar* aus und stoppe.

Zeigen Sie:

- Der Algorithmus hält nach spätestens  $n$  Markierungsschritten.
- Der Algorithmus ist korrekt.

Der Algorithmus hält nach spätestens  $n$  Markierungsschritten:

- In jedem Schritt wird eine weitere der  $n$  Variablen markiert, oder es wird unerfüllbar ausgegeben und gestoppt.

Der Algorithmus ist korrekt: Alle Disjunktionen bzw. Implikationen müssen den Wert 1 haben, damit  $F$  erfüllt ist.

- Die in Schritt 1 markierten Literale müssen mit 1 belegt werden.
- Daher ist auch die Markierung bzw. der Abbruch in Schritt 2 korrekt.
- Schritt 3 ist korrekt (nächste Folie)

Schritt 3 ist korrekt: Sei  $G$  ein beliebiges Disjunktionsglied in  $F$ .

- Falls  $G$  eine atomare Formel  $1 \rightarrow x_i$  ist, so ist  $G$  nach Schritt 1 erfüllt.
- Falls  $G$  eine atomare Formel  $x_i \rightarrow 0$  ist, dann kann  $x_i$  mit 0 belegt werden, sonst gibt Schritt 2 unerfüllbar aus.
- Falls  $G$  die Form  $(x_{i_1} \wedge \dots \wedge x_{i_k}) \rightarrow y$  hat, dann sind
  - entweder alle Literale in  $G$  mit 1 belegt, insbesondere  $y$ ,
  - oder eins der Literale  $x_{i_j}$  ist mit 0 belegt.

In beiden Fällen ist  $G$  nach Schritt 2 erfüllt.

- Falls  $G$  die Form  $(x_{i_1} \wedge \dots \wedge x_{i_k}) \rightarrow 0$  hat, so muss ein  $x_{i_j}$  mit 0 belegt sein (sonst gibt Schritt 2 unerfüllbar aus), und  $G$  ist erfüllt.

DNF-SAT ist in P!

Sei  $F = (x_{1,1} \wedge \dots \wedge x_{1,k_1}) \vee \dots \vee (x_{m,1} \wedge \dots \wedge x_{m,k_m})$  eine Formel in disjunktiver Normalform. Dann gilt:

- $F$  ist genau dann erfüllbar, wenn wenigstens eins der  $m$  Konjunktionsglieder  $(x_{j,1} \wedge \dots \wedge x_{j,k_j})$  erfüllbar ist.
- Ein Konjunktionsglied  $(x_{j,1} \wedge \dots \wedge x_{j,k_j})$  ist genau dann unerfüllbar, wenn es eine Variable  $x$  und  $\neg x$  enthält.

*Satz:* CLIQUE ist NP-vollständig.

*Beweis:* 3KNF-SAT  $\leq_p$  CLIQUE

- Sei  $F$  eine Formel in konjunktiver Normalform mit  $m$  Klauseln zu je genau 3 Literalen. (Wir können der Einfachheit halber genau 3 Literale annehmen, indem wir Literale in einer Klausel verdoppeln.)
- Wir konstruieren den Graphen  $G$  mit  $3m$  Knoten und:

$G$  hat Clique der Größe  $m \iff F$  ist erfüllbar

- Sei  $F = (a_{11} \vee a_{12} \vee a_{13}) \wedge \cdots \wedge (a_{m1} \vee a_{m2} \vee a_{m3})$
- Sei  $G = (V, E)$  mit
  - $V = \{(i, j) \mid 1 \leq i \leq m, 1 \leq j \leq 3\}$  und
  - $E = \{\{(i, j), (k, l)\} \mid i \neq k \text{ und } a_{ij} \neq \neg a_{kl}\}.$



Also:

- Die Knoten von  $G$  sind die in  $F$  vorkommenden Literale.
- Es gibt jeweils einen Knoten für jedes Vorkommen eines Literals in einer Klausel.
- Zwei Knoten sind miteinander verbunden, wenn ihre Literale
  - in verschiedenen Klauseln sind und
  - sich nicht direkt widersprechen ( $x$  und  $\neg x$ ).
- Die Größe von  $G$  ist polynomiell beschränkt: Es gibt  $3m$  Knoten, und damit höchstens  $9m^2$  viele Kanten.

*Übung:*

Zeigen Sie, dass  $F$  genau dann erfüllbar ist, wenn  $G$  eine Clique der Größe  $m$  hat.

die Formel  $F$  ist erfüllbar  $\Rightarrow G$  hat eine Clique der Größe  $m$

- Durch eine erfüllende Belegung ist in jeder der  $m$  Klauseln mindestens ein Literal mit eins belegt.
- Die Konstruktion des Graphen  $G$  stellt sicher, dass die entsprechenden Knoten durch Kanten verbunden sind.
- Diese Knoten bilden eine Clique der Größe  $m$ .

$G$  hat eine Clique der Größe  $m \Rightarrow$  die Formel  $F$  ist erfüllbar

- Die Knoten, die durch eine Klausel definiert werden, sind nicht durch Kanten verbunden. Daher müssen die Knoten, die eine Clique bilden, aus verschiedenen Klauseln stammen.
- Da die Clique die Größe  $m$  hat, muss aus jeder Klausel genau ein entsprechender Knoten enthalten sein.
- Die diesen Knoten entsprechenden Literale widersprechen sich nicht aufgrund der Konstruktion von  $G$ , daher können die Literale gleichzeitig auf eins gesetzt werden.

# GERICHTETER HAMILTON-KREIS ist NP-vollständig

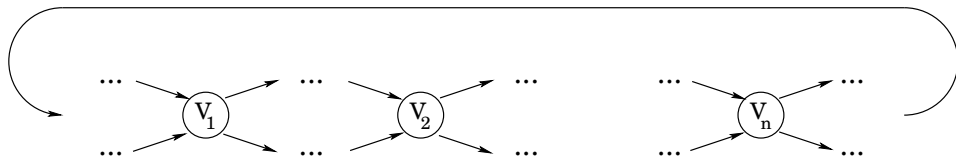
*Satz:* GERICHTETER HAMILTON-KREIS ist NP-vollständig.

*Beweis:*  $3\text{KNF-SAT} \leq_p \text{GERICHTETER HAMILTON-KREIS}$

- Sei  $F$  eine Formel in konjunktiver Normalform mit  $m$  Klauseln zu je genau 3 Literalen. Also:

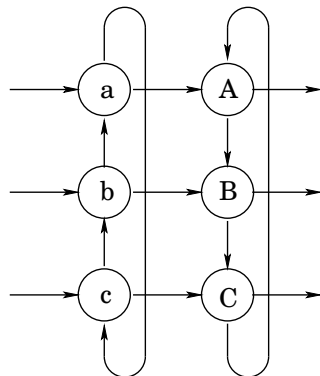
$$F = (a_{11} \vee a_{12} \vee a_{13}) \wedge \cdots \wedge (a_{m1} \vee a_{m2} \vee a_{m3})$$

- Seien  $x_1, x_2, \dots, x_n$  die Variablen von  $F$ .
- Der Graph  $G$  hat zunächst einmal die Knoten  $v_1, \dots, v_n$ , die die Variablen von  $F$  repräsentieren.

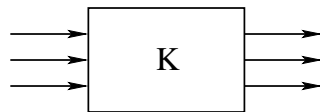


# GERICHTETER HAMILTON-KREIS ist NP-vollständig

Von jedem Knoten  $v_i$  gehen zwei Kanten aus. Diese führen zu Klauselgraphen  $K$ , von denen  $m$  Kopien  $K_1, \dots, K_m$  bereitstehen:



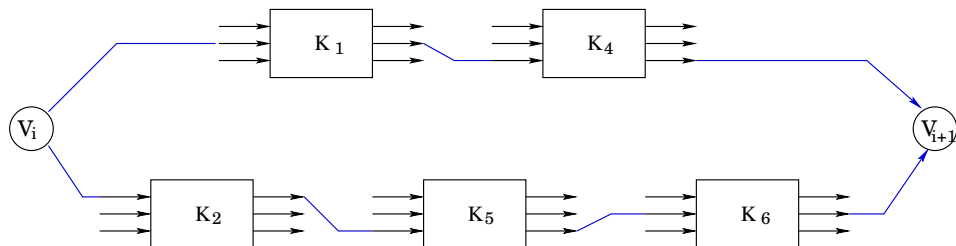
Im Folgenden stellen wir diese Klauselgraphen durch nebenstehendes Symbol dar:



Der obere vom Knoten  $v_i$  ausgehende Weg orientiert sich an den Vorkommen von  $x_i$  in den Klauseln, der untere an denen von  $\neg x_i$ .

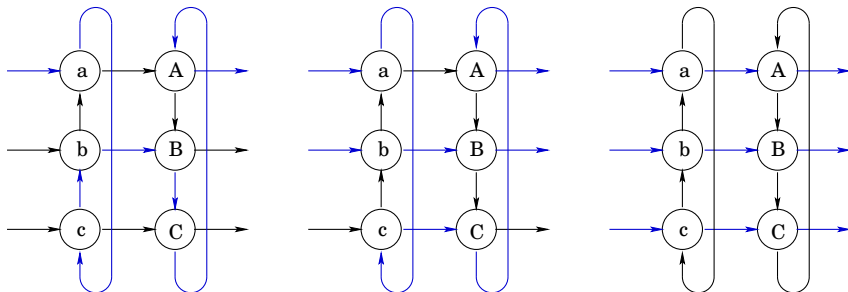
Beispiel:

- $x_i$  komme in Klausel 1, Position 2 und in Klausel 4, Position 3 vor.
- $\neg x_i$  komme in Klausel 2, Position 1, in Klausel 5, Position 3 und in Klausel 6, Position 2 vor.
- Dann erhalten wir folgende Kanten im Graphen:



Wenn die Formel  $F$  eine erfüllende Belegung hat:

- Wenn  $x_i$  die Belegung 1 hat, so folge von  $v_i$  aus dem oberen Pfad, sonst dem unteren.
- Dann durchläuft der Pfad die entsprechenden Klauselgraphen  $K_j$ , in denen  $x_i$  bzw.  $\neg x_i$  vorkommt.
- Je nachdem, wie viele und welche Literale in Klausel  $j$  den Wert 1 haben, wird der Klauselgraph  $K_j$  in einer von sieben Arten durchlaufen. Daraus ergibt sich der Hamilton-Kreis.



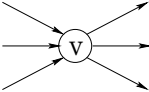
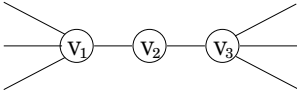
Wenn der Graph einen Hamilton-Kreis besitzt:

- Wenn der Kreis bei Knoten  $a$  einen Klauselgraphen betritt, dann wird dieser Klauselgraph bei Knoten  $A$  verlassen. (gilt ebenso für Knotenpaar  $b, B$  und  $c, C$ )
- Ein bei  $a$  eintretender Hamilton-Kreis kann nur folgende Wege nehmen:
  - $a \rightarrow A$  oder
  - $a \rightarrow c \rightarrow C \rightarrow A$  oder
  - $a \rightarrow c \rightarrow b \rightarrow B \rightarrow C \rightarrow A$
- Der Hamilton-Kreis kann also nur wie vorgesehen durch die Klauselgraphen laufen.
- Eine erfüllende Belegung ergibt sich anhand dessen, ob ein Knoten  $v_i$  oben oder unten verlassen wird.

# HAMILTON-KREIS ist NP-vollständig

*Satz:* GERICHTETER HAMILTON-KREIS  $\leq_p$  HAMILTON-KREIS

*Beweis:*

- Ersetze  durch  .
- Dann gilt: Der gerichtete Graph hat genau dann einen gerichteten Hamilton-Kreis, wenn der ungerichtete Graph einen Hamilton-Kreis hat.
- Die Reduktionsfunktion ist in polynomieller Zeit berechenbar.

*Euler-Kreis:* Eine Rundreise durch den Graphen, bei der jede Kante genau einmal benutzt wird.

*Frage:* Ist das Euler-Kreis-Problem NP-vollständig?



Das Euler-Kreis-Problem kann sehr effizient gelöst werden:

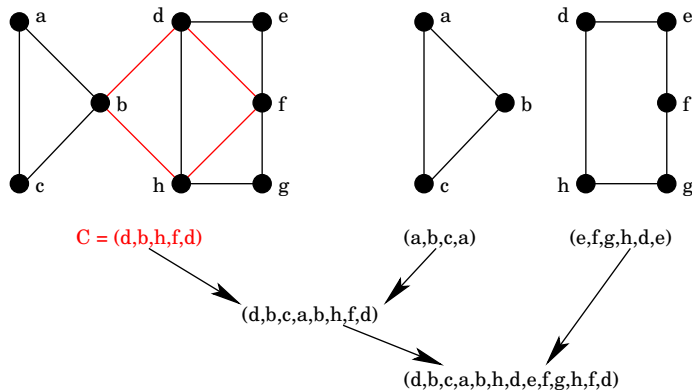
Ein zusammenhängender Graph  $G = (V, E)$  mit  $|V| \geq 3$  besitzt genau dann einen Euler-Kreis, wenn der Knotengrad eines jeden Knotens  $v \in V$  gerade ist.

Wenn ein Knoten bei einer Euler-Tour über eine Kante besucht wird, dann muss der Knoten über eine andere Kante wieder verlassen werden. Es ist also notwendig, dass der Knotengrad gerade ist.

Das die Bedingung auch hinreichend ist, kann mittels vollständiger Induktion gezeigt werden. (nächste Folie)

Bestimme iterativ einen Euler-Kreis:

- Finde einfachen Kreis  $C$  in  $G$  und entferne alle Kanten des Kreises  $C$  aus  $G$ .
- Zerfällt der Graph in mehrere Zusammenhangskomponenten, dann bestimme in jeder ZSHK einen Euler-Kreis.
- Setze die Kreise zu einem Euler-Kreis zusammen.



## 3-Knotenfärbung ist NP-vollständig

3-Knotenfärbung ist in NP:

- Bestimme für Graph  $G = (V, E)$  nicht-deterministisch eine Zuordnung  $f : V \rightarrow \{1, 2, 3\}$  von Farben zu Knoten.
- Prüfe für jede Kante  $\{u, v\} \in E$ , ob  $f(u) \neq f(v)$  gilt.

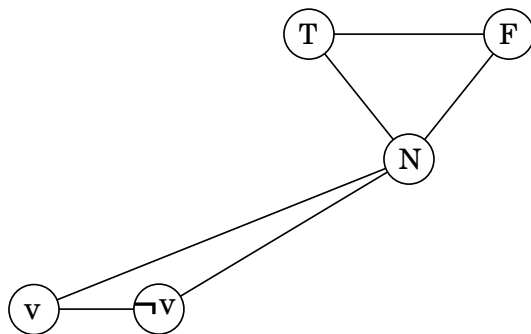
$3\text{-KNF-SAT} \leq_p 3\text{-Knotenfärbung}$

- Gegeben: Boolesche Formel  $F = C_1 \wedge \dots \wedge C_m$  in KNF über Variable  $x_1, \dots, x_n$ ; jede Klausel  $C_i$  enthalte nur drei Literale.
- Aufgabe: Erstelle Graph  $G_F$ , der genau dann 3-färbbar ist, wenn  $F$  erfüllbar ist.
- Idee: Eine Belegung der Variablen muss durch eine 3-Färbung des Graphen erzeugt werden.

## 3-Knotenfärbung ist NP-vollständig

Erzeuge einen Kreis der Länge drei mit Knoten T, F und N.

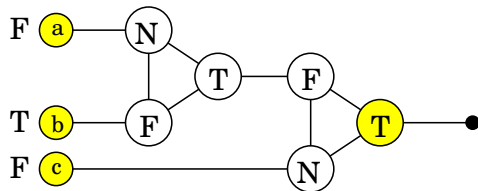
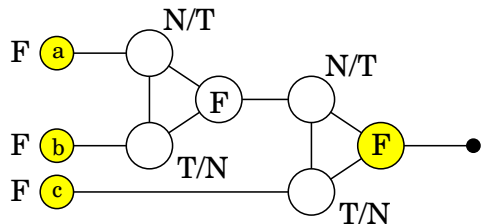
Erzeuge für jede Variable  $x_i$  zwei Knoten  $v_i$  und  $\neg v_i$ . Ordne die Knoten  $v_i$ ,  $\neg v_i$  und N in einem Kreis der Länge drei an.



Wird  $v_i$  mit T gefärbt, dann muss  $\neg v_i$  mit F gefärbt werden und umgekehrt. Eine 3-Färbung entspricht also einer Belegung der Variablen.

### 3-Knotenfärbung ist NP-vollständig

Erzeuge für jede Klausel  $C_i = (a \vee b \vee c)$  einen OR-Gadget-Graph:



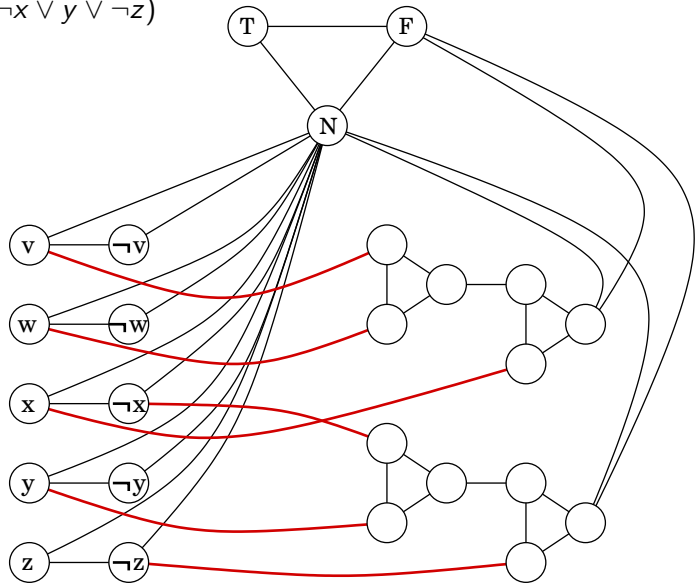
Eigenschaften:

- Falls  $a, b, c$  alle mit F gefärbt sind, muss der Ausgang auch mit F gefärbt werden.
- Falls ein Knoten aus  $a, b, c$  mit T gefärbt ist, kann der Ausgang auch mit T gefärbt werden.

Der Ausgang eines OR-Gadget-Graphen wird mit den zentralen Knoten N und F jeweils durch eine Kante verbunden.

## 3-Knotenfärbung ist NP-vollständig

Beispiel:  $F = (v \vee w \vee x) \wedge (\neg x \vee y \vee \neg z)$



## 3-Knotenfärbung ist NP-vollständig

Falls  $F$  erfüllbar ist, dann ist  $G_F$  mit drei Farben färbbar:

- Falls  $x_i$  mit true belegt ist, dann färbe  $v_i$  mit T und  $\neg v_i$  mit F.
- In jeder Klausel  $C_i = (a \vee b \vee c)$  ist mindestens ein Literal  $a, b, c$  mit T gefärbt.
- Daher kann der entsprechende OR-Gadget-Graph von  $C_i$  mit drei Farben gefärbt werden, sodass der Ausgang mit T gefärbt ist.

Falls  $G_F$  mit drei Farben färbbar ist, dann ist  $F$  erfüllbar:

- Wenn  $v_i$  mit T gefärbt ist, dann belege  $x_i$  mit true, sonst mit false. So erhalten wir eine zulässige Belegung der Variablen.
- Für jede beliebige Klausel  $C_i = (a \vee b \vee c)$  gilt: Nicht alle Literale  $a, b, c$  können mit F gefärbt sein, denn dann müsste der Ausgang mit F gefärbt werden, aber der Ausgang ist mit den Knoten N und F verbunden!

## 4-Knotenfärbung ist NP-vollständig

4-Knotenfärbung ist in NP:

- Bestimme für Graph  $G = (V, E)$  nicht-deterministisch eine Zuordnung  $f : V \rightarrow \{1, 2, 3, 4\}$  von Farben zu Knoten.
- Prüfe für jede Kante  $\{u, v\} \in E$ , ob  $f(u) \neq f(v)$  gilt.

3-Knotenfärbung  $\leq_p$  4-Knotenfärbung

- Gegeben: Graph  $G = (V, E)$
- Aufgabe: Erstelle Graph  $G' = (V', E')$ , der genau dann 4-färbbar ist, wenn  $G$  3-färbbar ist.
- Idee: Füge einen neuen Knoten  $x$  hinzu und verbinde  $x$  mit jedem Knoten aus  $V$  durch eine Kante, also  $V' = V \cup \{x\}$ ,  $E' = E \cup \{\{x, v\} \mid v \in V\}$ .

Aufgrund dieser Idee gilt allgemein:  $k$ -Färbung ist NP-vollständig für  $k \geq 3$ .



*Ablauf bei Reduktionen:* Zeige  $ALT \leq_p NEU$  mittels  $f$ .

- Erstelle zunächst eine Funktion  $f$ , die aus einer Eingabe  $x$  von  $ALT$  eine Eingabe  $f(x)$  von  $NEU$  macht, wobei gelten muss:

$$x \in ALT \iff f(x) \in NEU$$

- Wäre  $NEU$  effizient lösbar, dann wäre auch  $ALT$  effizient lösbar:

$$x \stackrel{?}{\in} ALT \rightsquigarrow \begin{cases} f(x) \in NEU & \Rightarrow x \in ALT \\ f(x) \notin NEU & \Rightarrow x \notin ALT \end{cases}$$

Um  $x \stackrel{?}{\in} ALT$  zu beantworten, nutzen wir  $f$  und den (fiktiven) effizienten Algorithmus für  $NEU$ .

**VERTEX COVER PROBLEM:** Finde zu einem Graphen  $G = (V, E)$  eine möglichst kleine Knotenmenge  $V' \subseteq V$ , sodass jede Kante  $e \in E$  inzident zu einem Knoten in  $V'$  ist.

$VCP := \{(G, k) \mid G \text{ hat Vertex-Cover } C \text{ mit } |C| \leq k\}$

*Zeigen Sie:*  $CLIQUE \leq_p VCP$

**TRAVELING SALESPERSON PROBLEM:** Bestimme zu einem Graphen  $G = (V, E, c)$  eine Rundreise, die genau einmal durch alle Knoten führt, die wieder am Startknoten endet und deren Länge möglichst kurz ist. Dabei ist  $c : E \rightarrow \mathbb{N}$  eine Kostenfunktion, die zu jeder Kante in  $E$  deren „Länge“ definiert.

$TSP := \{(G, k) \mid G \text{ hat Rundreise } P \text{ mit } \sum_{e \in P} c(e) \leq k\}$

*Zeigen Sie:*  $HCP \leq_p TSP$

## $\text{CLIQUE} \leq_p \text{VCP}$

- Berechne zur Eingabe  $G = (V, E)$  und  $k \in \mathbb{N}$  des Cliques-Problems den Graph  $\overline{G} := (V, \overline{E})$  und  $k' := |V| - k$ , wobei  $\overline{E}$  genau die Kanten enthält, die in  $G$  fehlen, also:  $\overline{E} := \{\{u, v\} \mid u, v \in V, u \neq v, \{u, v\} \notin E\}$
- Dann gilt: In  $G$  existiert eine Clique der Größe mindestens  $k$  genau dann, wenn in  $\overline{G}$  ein Vertex-Cover der Größe höchstens  $|V| - k$  existiert.
- Begründung: Aus einer Clique der Größe  $k$  in  $G$  wird ein Independent Set der Größe  $k$  in  $\overline{G}$ , also müssen in  $\overline{G}$  alle Kanten inzident zu den restlichen  $|V| - k$  Knoten sein.

## HCP $\leq_p$ TSP

- Berechne zur Eingabe  $G = (V, E)$  des Hamilton-Kreis- Problems den vollständigen Graphen  $G' = (V, E', c)$  und  $k := |V|$ , wobei  $E' := \{\{u, v\} \mid u, v \in V, u \neq v\}$  und  $c : E' \rightarrow \mathbb{N}$  mit

$$c(e) := \begin{cases} 1, & \text{falls } e \in E \\ 2, & \text{sonst} \end{cases}$$

- Dann gilt: In  $G$  existiert genau dann ein Hamilton-Kreis, wenn in  $G'$  eine Rundreise der Länge  $|V|$  existiert.
- Begründung: Wenn ein Hamilton-Kreis in  $G$  existiert, dann auch in  $G'$ , und jede dieser Kanten hat das Gewicht eins.

Wenn in  $G'$  ein Hamilton-Kreis der Länge  $|V|$  existiert, dann läuft dieser Kreis nur über Kanten, die das Gewicht eins haben. Diese Kanten existieren auch in  $G$ .

Bestimmend für die Rechenzeit ist

- die *Anzahl* der zu bearbeitenden Elemente
  - Sortieren von Elementen
  - Skalar-Produkt zweier Vektoren
  - Städte-Rundreise
- oder die *Größe* der zu bearbeitenden Elemente
  - Multiplikation zweier Zahlen
  - Primzahlzerlegung einer Zahl
  - Rucksack-Problem
- oder beides.

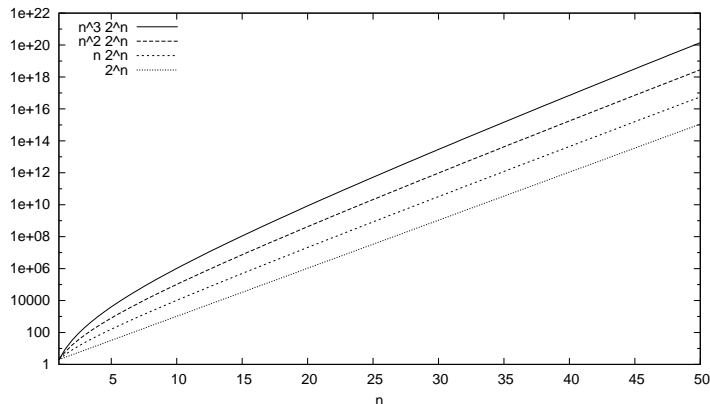
## *Einleitung*

- Bewertung von Algorithmen
- Berechenbarkeitstheorie
- Komplexitätstheorie
- *Exakte Algorithmen für schwere Probleme*<sup>3</sup>

---

<sup>3</sup>basiert auf Vorlesungen von Dorothea Wagner (Karlsruher Institut für Technologie) und Rolf Niedermeier (Universität Tübingen)

Bei der  $\mathcal{O}^*$ -Notation werden polynomielle Faktoren vernachlässigt. So gilt  $n^2 \cdot 2^n \in \mathcal{O}^*(2^n)$  und auch  $n^5 + n^3 \cdot 2^n \in \mathcal{O}^*(2^n)$ . Wird manchmal auch als  $\tilde{\mathcal{O}}$  bezeichnet.



Idee: Das exponentielle Wachstum ist so stark, dass polynomielle Faktoren kaum ins Gewicht fallen.

## *Exakte Algorithmen für schwere Probleme*

- 3-SAT
- Vertex-Cover
- Independent Set



*Algorithmus A1:* Sei  $x$  eine Variable in der Formel  $\Phi$ , und sei  $\Phi \mid x$  die Formel, die aus  $\Phi$  entsteht, wenn  $x$  mit 1 belegt wird.

```
function 3SAT( $\Phi$ )  
  if 3SAT( $\Phi \mid x$ ) then return true  
  return 3SAT( $\Phi \mid \bar{x}$ )
```

*Laufzeit:*  $T(n) = 2 \cdot T(n-1) + \text{poly}(n) \in \mathcal{O}^*(2^n)$

*Idee zur Laufzeitbestimmung:* Wir gehen davon aus, dass sich eine exponentielle Laufzeit  $T(n) \in \mathcal{O}^*(b^n)$  ergibt und wollen die Basis  $b$  bestimmen. Setze  $T(n) := b^n$  und schätze ab:

$$\begin{aligned} b^n &\approx 2 \cdot b^{n-1} + n^k & | & : b^{n-1} \\ \iff b &\approx 2 + \frac{n^k}{b^{n-1}} & | & \lim_{n \rightarrow \infty} \frac{n^k}{b^{n-1}} = 0 \\ \iff b &\approx 2 \end{aligned}$$

*Algorithmus A2:* Sei  $(x \vee y \vee z)$  eine Klausel in der Formel  $\Phi$ , und sei  $\Phi \mid x$  die Formel, die aus  $\Phi$  entsteht, wenn  $x$  mit 1 belegt wird.

```
function 3SAT( $\Phi$ )  
  if 3SAT( $\Phi \mid x$ ) then return true  
  if 3SAT( $\Phi \mid \bar{x}y$ ) then return true  
  return 3SAT( $\Phi \mid \bar{x}\bar{y}z$ )
```

*Laufzeit:*  $T(n) = T(n-1) + T(n-2) + T(n-3) + \text{poly}(n) \in \mathcal{O}^*(1, 84^n)$

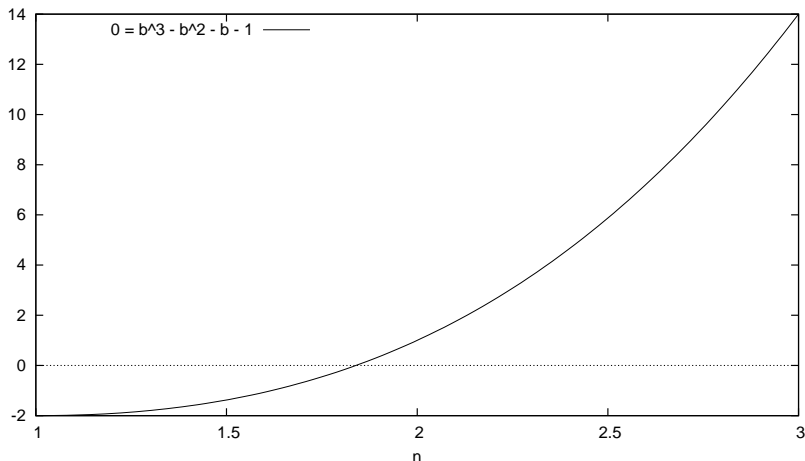
*Idee zur Laufzeitbestimmung:* Setze  $T(n) := b^n$  und schätze ab:

$$\begin{aligned} b^n &\approx b^{n-1} + b^{n-2} + b^{n-3} + n^k & | & : b^{n-3} \\ \iff b^3 &\approx b^2 + b + 1 + \cancel{\frac{n^k}{b^{n-3}}} & | & \lim_{n \rightarrow \infty} \frac{n^k}{b^{n-3}} = 0 \\ \iff b^3 &\approx b^2 + b + 1 \\ \iff 0 &\approx b^3 - b^2 - b - 1 & \rightsquigarrow & \text{Newton-Verfahren} \end{aligned}$$

Das Newton-Verfahren liefert für das charakteristische Polynom

$$b^3 - b^2 - b - 1$$

eine Nullstelle bei  $b = 1,8392\dots$



*Algorithmus A3:* Wenn eine Variable nur positiv oder nur negativ in der Formel vorkommt, dann können wir den Wert der Variablen entsprechend setzen und die Teilformeln streichen. Seien daher die Klauseln  $(x \vee y \vee z)$  und  $(\bar{x} \vee u \vee v)$  in der Formel  $\Phi$  enthalten.

```
function 3SAT( $\Phi$ )  
  if 3SAT( $\Phi \mid xu$ ) then return true  
  if 3SAT( $\Phi \mid x\bar{u}v$ ) then return true  
  if 3SAT( $\Phi \mid \bar{x}y$ ) then return true  
  return 3SAT( $\Phi \mid \bar{x}\bar{y}z$ )
```

*Laufzeit:*  $T(n) = 2T(n-2) + 2T(n-3) + \text{poly}(n) \in \mathcal{O}^*(1, 77^n)$

Ganz besonders tricky geht es in Zeit  $\mathcal{O}^*(1, 34^n)$ , siehe:

Robin A. Moser, Dominik Scheder. A Full Derandomization of Schöningh's  $k$ -SAT Algorithm. arXiv:1008.4067, 2010.

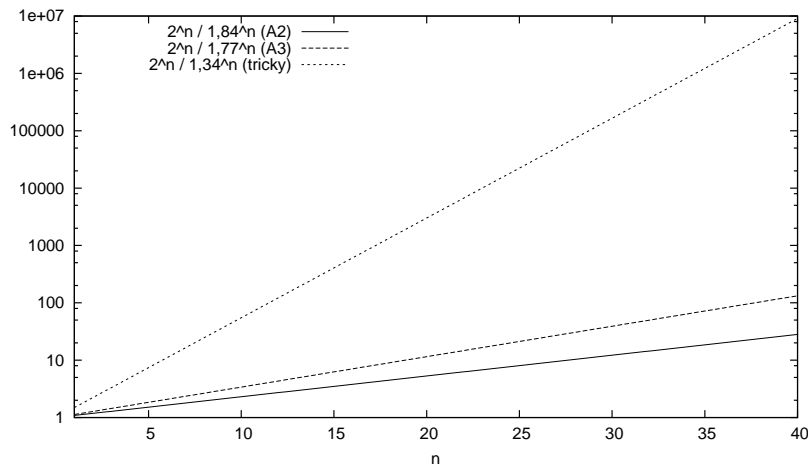
# Exponentialzeit-Algorithmen für 3-SAT

Laufzeitvorteil gegenüber A1 entspricht dem Quotienten

•  $2^n / 1,84^n$  bei A2

•  $2^n / 1,77^n$  bei A3

•  $2^n / 1,34^n$  bei tricky



## *Exakte Algorithmen für schwere Probleme*

- 3-SAT
- *Vertex-Cover*
- Independent Set

Gegeben sei ein ungerichteter Graph  $G = (V, E)$  mit  $n$  Knoten.

*Brute-Force-Ansatz:* Suche ein Vertex Cover, indem alle  $\binom{n}{k}$  vielen Teilmengen der Größe  $k$  betrachtet werden.

Laufzeit:  $\mathcal{O}(\binom{n}{k} \cdot |G|) \in \mathcal{O}(n^k \cdot |G|)$

$$\begin{aligned}\binom{n}{k} &= \frac{n!}{k! \cdot (n-k)!} = \frac{n \cdot (n-1) \cdot \dots \cdot (n-k+1)}{k!} \\ &\leq \frac{n \cdot n \cdot \dots \cdot n}{k!} \leq n^k\end{aligned}$$

Die Laufzeit ist polynomiell, falls  $k$  konstant ist, also nicht zur Eingabe des Problems gehört.

## Greedy-Heuristik:

$C := \emptyset$

**while** es gibt noch Kanten in  $G$  **do**

    sei  $v$  ein Knoten mit größter Anzahl Nachbarn

    nimm  $v$  in  $C$  auf

    entferne  $v$  und alle inzidenten Kanten aus  $G$

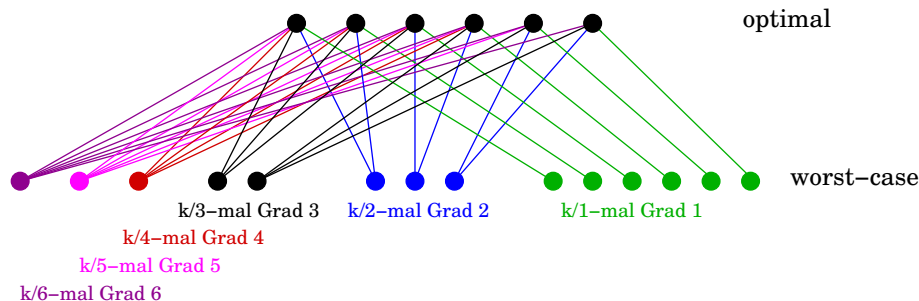
Laufzeit:  $\mathcal{O}(n^2)$

*Bemerkung:* Es gibt Graphen  $G$ , für die die Greedy-Heuristik ein Vertex-Cover der Größe  $\ln(k) \cdot k$  liefert, wobei  $k$  die Größe des kleinsten Vertex-Cover ist.

Der Fehler kann also  $\ln(k)$  groß werden und ist nicht durch eine Konstante beschränkt.



Worst-Case-Graph mit  $k = 6$  für die Greedy-Heuristik:



- obiger bipartiter Graph hat ein Vertex-Cover der Größe  $k = 6$
- im schlechtesten Fall werden die unteren Knoten ins VC aufgenommen, das die Größe  $6 + 3 + 2 + 1 + 1 + 1 = 14$  hat.
- Fehler im Beispiel:  $\frac{14}{6} \approx 2,33$
- harmonische Reihe:  $\lfloor \frac{k}{1} \rfloor + \lfloor \frac{k}{2} \rfloor + \lfloor \frac{k}{3} \rfloor + \dots + \lfloor \frac{k}{k} \rfloor = k \cdot \sum_{i=1}^k \lfloor \frac{1}{k} \rfloor \approx k \cdot \ln(k)$

## Approximationsalgorithmus:

$C := \emptyset$

**while** es gibt noch Kanten in  $G$  **do**

    nimm irgendeine Kante  $\{u, v\}$  von  $G$

    nimm  $u$  und  $v$  beide in  $C$  auf

    entferne  $u$  und  $v$  und alle dazu inzidenten Kanten aus  $G$

*Bemerkung:* Der Algorithmus liefert für alle Graphen ein VC, dass höchstens doppelt so viele Knoten enthält wie ein minimales VC:

- Sei  $F$  die Menge der ausgewählten Kanten, und sei  $C = \{u, v \mid \{u, v\} \in F\}$  das berechnete Vertex-Cover.
- Jedes Vertex-Cover  $C'$  muss  $u$  oder  $v$  enthalten, da sonst die Kante  $\{u, v\}$  nicht abgedeckt würde. Also gilt:

$$|C'| \geq \frac{1}{2}|C| \iff |C| \leq 2|C'|$$

*Algorithmus* binärer Entscheidungsbaum mit beschränkter Tiefe  $k$ :

```
function VC( $G, k$ )  
  if  $k = 0$  und es gibt noch Kanten in  $G$  then return false  
  if es gibt keine Kanten in  $G$  then return true  
  nimm irgendeine Kante  $\{u, v\}$  von  $G$   
   $x := \text{VC}(G - \{u\}, k - 1)$   
   $y := \text{VC}(G - \{v\}, k - 1)$   
  return  $x \vee y$ 
```

Laufzeit:  $\mathcal{O}(2^k \cdot |G|)$

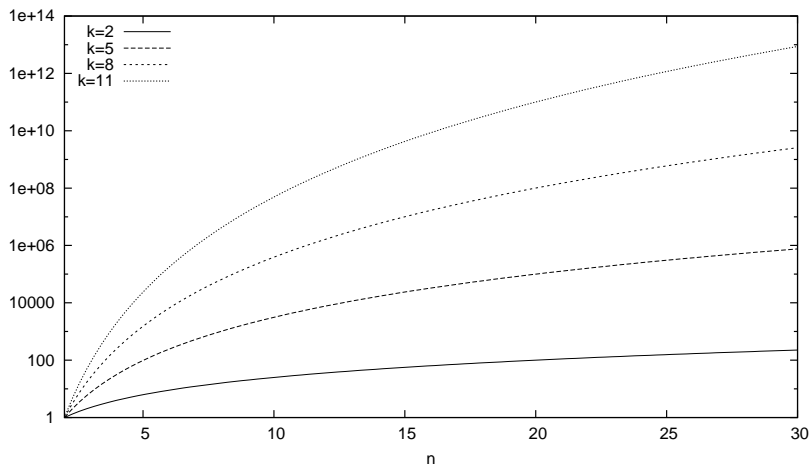
*Bemerkungen:* Falls  $k$  konstant ist

- ergibt sich eine polynomielle Laufzeit und
- die asymptotische Laufzeit hängt nicht von  $k$  ab.

Laufzeitvorteil entspricht dem Quotienten  $n^k/2^k$

• Brute-Force:  $\mathcal{O}(n^k \cdot |G|)$

• Entscheidungsbaum:  $\mathcal{O}(2^k \cdot |G|)$



# Fixed Parameter Tractable (FPT)

Ein *parametrisiertes Problem* ist ein Paar  $(\Pi, k)$ , wobei  $\Pi$  ein Entscheidungsproblem ist,  $\mathcal{I}$  die Menge der möglichen Eingaben für  $\Pi$  ist, und  $k : \mathcal{I} \rightarrow \mathbb{N}$  eine Parametrisierung ist, die sich in polynomieller Zeit berechnen lässt. Der Wert der Parametrisierung sollte bei praktischen Anwendungen klein sein.

Ein Algorithmus  $A$  ist ein *fpt-Algorithmus*, wenn die Laufzeit für jede Eingabe  $I \in \mathcal{I}$  in  $\mathcal{O}(f(k(I)) \cdot |I|^c)$  liegt, wobei  $f$  eine Funktion und  $c \in \mathbb{N}$  konstant ist.

Ein parametrisiertes Problem  $(\Pi, k)$  gehört zur Klasse *FPT* und wird *fixed parameter tractable* genannt, wenn es einen fpt-Algorithmus bezüglich  $k$  gibt, der  $\Pi$  entscheidet.

Vertex-Cover ist in FPT bezüglich des Standardparameters  $k$ .

Die *Idee bei parametrisierten Algorithmen*: Beschränke die kombinatorische Explosion auf den Parameter  $k$ .

Kleine Parameter  $k$  finden sich in vielen Anwendungsbereichen:

- Netzwerke: Positioniere eine kleine Anzahl  $k$  von Geräten wie Router in einem großen Netzwerk.
- VLSI-Entwurf: Layout in der Chip-Produktion ist beschränkt auf  $k < 30$  Verdrahtungsschichten.
- Algorithmische Biologie: Untersuchungen zu DNS-Sequenzen der Länge  $n$  für  $k$  wenige Spezies.

Ein Problem  $\Pi$  kann bezüglich eines Parameters  $k_1$  in FPT sein, und bezüglich eines anderen Parameters  $k_2$  nicht.

*Frage:* Lässt sich für Vertex-Cover die Laufzeit noch weiter reduzieren?

Die Verkleinerung von Suchbaumgrößen ist wichtig, wie wir bereits bei dem 3-SAT-Problem gesehen haben.

*Kernbildung:* Reduziere die Eingabe durch Anwendung verschiedener Regeln auf einen schweren, aber kleinen Problemkern.

- Reduziere die Eingabe  $(I, k)$  in Zeit  $\mathcal{O}(p(|I|))$  auf eine äquivalente Eingabe  $I'$ , so dass die Größe von  $I'$  nur von  $k$  und nicht von  $|I|$  abhängt, und  $p$  ein Polynom ist.
- Löse die äquivalente Eingabe  $I'$  mit erschöpfender Suche. Die Laufzeit für diesen Schritt hängt dann nur noch von  $k$ , aber nicht von  $|I|$  ab.

Beobachtungen für ein Vertex-Cover  $V' \subseteq V$  für einen Graphen  $G = (V, E)$  mit  $|V'| \leq k$ :

- Für einen Knoten  $v \in V$  liegt  $v$  selbst oder seine gesamte Nachbarschaft  $N(v)$  in  $V'$ .
- $V'$  enthält jeden Knoten  $v \in V$  mit Knotengrad  $\deg(v) > k$ . Denn würde  $v$  nicht in  $V'$  aufgenommen, dann müssten mehr als  $k$  viele Nachbarn im Vertex-Cover liegen. ⚡

Jedes Vertex-Cover für  $G$  hat mehr als  $k$  viele Knoten, falls

- $\Delta(G) \leq k$  ist, also jeder Knoten im Graphen  $G$  einen Knotengrad höchstens  $k$  hat, **und**
- $|E| > k^2$  ist, also mehr als  $k^2$  viele Kanten im Graphen enthalten sind,

denn jeder der  $k$  Knoten kann höchstens  $k$  viele Kanten abdecken.



Sei  $G = (V, E)$  ein Graph und  $k \in \mathbb{N}$ . Der folgende Algorithmus basiert auf den obigen Beobachtungen.

```
function VERTEXCOVER( $G, k$ )  
   $H := \{v \in V \mid \deg(v) > k\}$   $\mathcal{O}(n)$   
  if  $|H| > k$  then return false  $\mathcal{O}(1)$   
   $k' := k - |H|$   $\mathcal{O}(1)$   
   $G' := G - H$   $\mathcal{O}(n \cdot k)$   
  if  $|E(G')| > k \cdot k'$  then return false  $\mathcal{O}(n)$   
  entferne alle isolierten Knoten aus  $G'$   $\mathcal{O}(n)$   
  berechne ein Vertex-Cover der Größe  $k'$  in  $G'$   $\mathcal{O}(2^k \cdot k^2)$ 
```

*Laufzeit:*  $\mathcal{O}(n \cdot k + 2^k \cdot k^2)$  mit  $n = |V|$  und  $k' \leq k$ .

*Frage:* Wie wird der Graph gespeichert, und wie werden die einzelnen Operationen ausgeführt, um die angegebenen Laufzeiten zu erreichen?

Wir speichern den Graphen mittels *Adjazenzlisten*: Zu jedem Knoten wird eine Liste der Nachbarknoten gespeichert.

- Jede Liste hat Länge  $\mathcal{O}(n)$ , da jeder Knoten höchstens mit  $n - 1$  anderen Knoten durch eine Kante verbunden sein kann.
- Die Länge einer Liste wird in einer Variablen gespeichert und kann daher in Zeit  $\mathcal{O}(1)$  ermittelt werden: Beim Einfügen in oder Entfernen aus der Liste aktualisiere diese Variable.

Bleibt noch zu klären, wie  $G' := G - H$  berechnet wird:

- Lösche für jeden Knoten  $v \in H$  seine Nachbarschaftsliste.  
⇒ Übrig bleibende Listen haben Länge höchstens  $k$ .
- Durchlaufe jede verbliebene Liste und lösche Knoten  $v \in H$ .  
⇒ Höchstens  $n$  Listen der Länge höchstens  $k$  müssen durchlaufen werden und Elemente daraus gelöscht werden:  $\mathcal{O}(n \cdot k)$

Die Anzahl der zu lösenden Teilprobleme und damit die Laufzeit hängt von der Struktur des Baumes ab:

- Binärbaum:  $T(k) \leq T(k - t_1) + T(k - t_2) + c$
- allgemein:  $T(k) \leq T(k - t_1) + \dots + T(k - t_s) + c$

Der Vektor  $(t_1, \dots, t_s)$  heißt **Verzweigungsvektor**. Dabei geben  $t_1, \dots, t_s$  an, um wie viel kleiner die Teilprobleme im Vergleich zum ursprünglichen Problem sind.

Beispiel: Ein Binärbaum, bei dem in beiden Zweigen das zu lösende Teilproblem um eins kleiner ist als im ursprünglichen Problem, wenn also  $t_1 = 1$  und  $t_2 = 1$  gilt, hat den Verzweigungsvektor  $(1, 1)$ . (z.B. Algorithmus A1 für 3-SAT)

Die Zahl der Teilprobleme ist exponentiell in  $k$ , also  $T(k) \in \mathcal{O}(b^k)$ , wobei die Basis  $b$  vom Verzweigungsvektor abhängt:

Vektor	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(2,3)	(3,3)	(3,3,6)	(3,4,6)
Basis $b$	2	1,618	1,466	1,381	1,325	1,325	1,26	1,342	1,305

## *Gedanken-Experiment für Vertex-Cover:*

Angenommen, es gibt nach der Kernbildung immer einen Knoten  $v$  mit  $\deg(v) \geq 4$ .

- Wähle in jedem Schritt entweder  $v$  oder alle seine Nachbarn und verzweige im binären Entscheidungsbaum entsprechend.
  - $\Rightarrow$  Verzweigungsvektor  $(1, 4)$
- Laufzeit:  $\mathcal{O}(n \cdot k + 1,381^k \cdot k^2)$

*Problem:* Es gibt nicht in jedem Schritt einen Knoten mit Grad mindestens vier.

*Idee:* Verzweige in jedem Schritt je nach eintretendem Fall mit Verzweigungsvektor  $(1, 5)$ ,  $(2, 3)$ ,  $(3, 3)$ ,  $(3, 4, 6)$  oder  $(3, 3, 6)$  oder höchstens einmal pro Pfad  $(1, 4)$ .

Gesamtlaufzeit:  $\mathcal{O}(n \cdot k + 1,342^k \cdot k^2)$

Verzweige anhand der Bedingung mit der kleinsten Nummer:

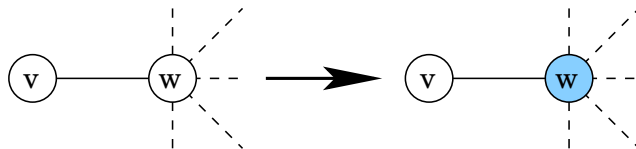
- ① falls ein Knoten  $v$  mit  $\deg(v) = 1$  existiert  $\leadsto$  Regel 1
- ② falls ein Knoten  $v$  mit  $\deg(v) \geq 5$  existiert  $\leadsto$  Regel 2
- ③ falls ein Knoten  $v$  mit  $\deg(v) = 2$  existiert  $\leadsto$  Regel 3
- ④ falls ein Knoten  $v$  mit  $\deg(v) = 3$  existiert  $\leadsto$  Regel 4
- ⑤ falls alle Knoten Grad 4 haben  $\leadsto$  Regel 5

Offensichtlich ist immer eine der obigen Bedingungen erfüllt.

Auf den folgenden Folien beschreiben wir die Regeln im Detail.

**Regel 1:** Es existiert ein Knoten  $v$  mit  $\deg(v) = 1$ .

Sei  $w$  der Nachbar von  $v$ .

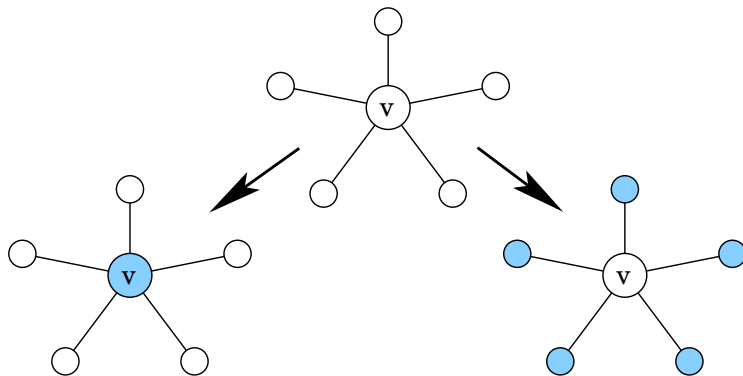


⇒ Es kann nie besser sein,  $v$  statt  $w$  zu nehmen.

⇒ Keine Verzweigung nötig: Nimm  $w$  ins Vertex-Cover auf!

Sicher: alle Knoten haben Grad 2, 3, 4, 5, ...

**Regel 2:** Es existiert ein Knoten  $v$  mit  $\deg(v) \geq 5$ .



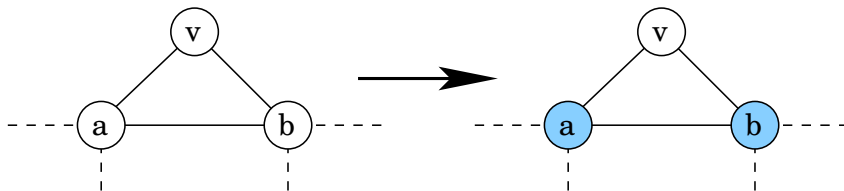
⇒ Wähle entweder  $v$  oder alle Nachbarn von  $v$ .

⇒ Verzweigungsvektor:  $(1, 5)$

Sicher: alle Knoten haben Grad 2, 3 oder 4

**Regel 3:** Es existiert ein Knoten  $v$  mit  $\deg(v) = 2$ .

**Fall 1:** Die Nachbarn  $a$  und  $b$  von  $v$  sind adjazent.



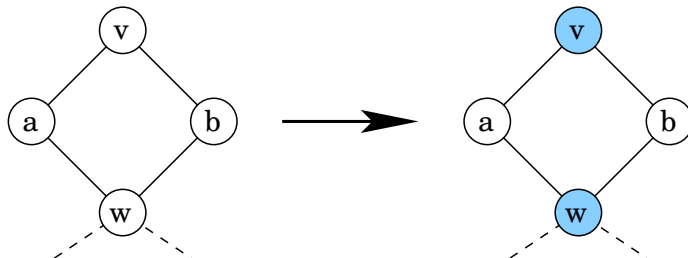
- ⇒ Zwei der Knoten  $v, a, b$  müssen gewählt werden
- ⇒ Es ist nie besser  $v$  zu wählen.
- ⇒ Keine Verzweigung nötig: Nimm  $a$  und  $b$  ins Vertex-Cover auf!



Sicher: alle Knoten haben Grad 2, 3 oder 4

**Regel 3:** Es existiert ein Knoten  $v$  mit  $\deg(v) = 2$ .

**Fall 2:** Die Nachbarn  $a$  und  $b$  von  $v$  haben Grad zwei und einen gemeinsamen Nachbarn  $w$ .

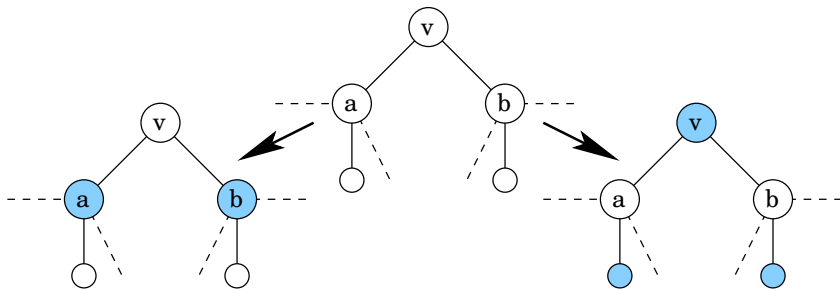


- ⇒ Zwei der Knoten  $v$ ,  $a$ ,  $b$ ,  $w$  müssen gewählt werden.
- ⇒ Es ist nie besser  $a$  und  $b$  zu wählen.
- ⇒ Keine Verzweigung: Nimm  $v$  und  $w$  ins Vertex-Cover auf!

Sicher: alle Knoten haben Grad 2, 3 oder 4

**Regel 3:** Es existiert ein Knoten  $v$  mit  $\deg(v) = 2$ .

**Fall 3:** sonst ( $a$  und  $b$  sind nicht adjazent,  $a$  oder  $b$  haben Grad größer als zwei oder haben keinen gemeinsamen Nachbarn)

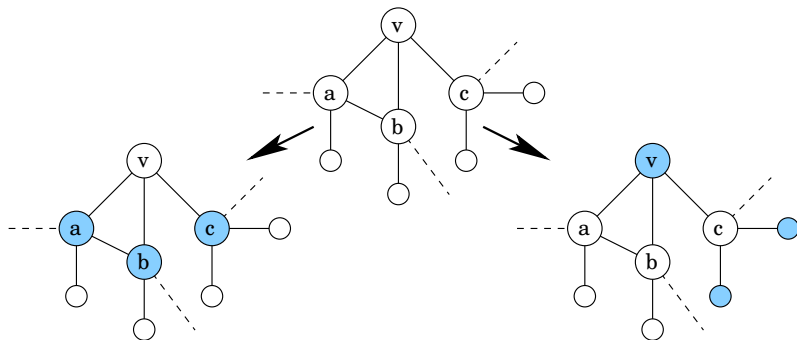


- ⇒ Falls  $a$  oder  $b$  gewählt wurde: sinnlos  $v$  zu wählen
- ⇒ Wähle entweder  $a$  und  $b$ , oder die Nachbarn von  $a$  und  $b$ .
- ⇒ Verzweigungsvektor:  $(2, 3)$

Sicher: alle Knoten haben Grad 3 oder 4

**Regel 4:** Es existiert ein Knoten  $v$  mit  $\deg(v) = 3$ .

**Fall 1:**  $v$  ist Teil eines Dreiecks aus  $v$ ,  $a$  und  $b$ .



⇒ Falls  $c$  gewählt wurde, macht es keinen Sinn,  $v$  zu wählen.

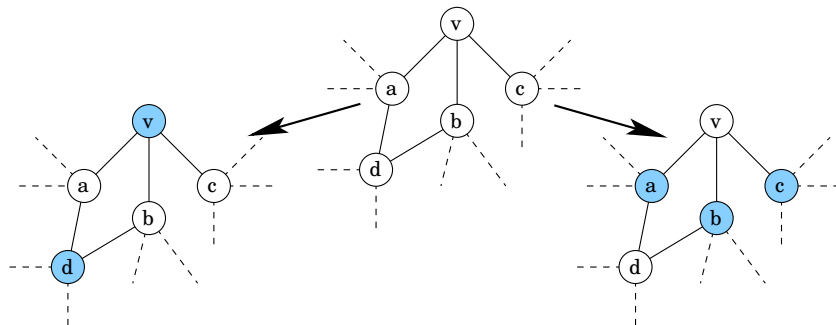
⇒ Füge die Nachbarn von  $v$  oder die Nachbarn von  $c$  hinzu.

⇒ Verzweigungsvektor:  $(3, 3)$

Sicher: alle Knoten haben Grad 3 oder 4

**Regel 4:** Es existiert ein Knoten  $v$  mit  $\deg(v) = 3$ .

**Fall 2:**  $v$  ist Teil eines Vierecks aus  $v, a, d$  und  $b$ .



⇒ In jedem Vertex-Cover sind zumindest  $v$  und  $d$  oder  $a$  und  $b$  enthalten. Wenn  $a$  und  $b$  enthalten sind: sinnlos  $v$  zu wählen.

⇒ Füge  $a, b$  und  $c$  oder  $v$  und  $d$  hinzu.

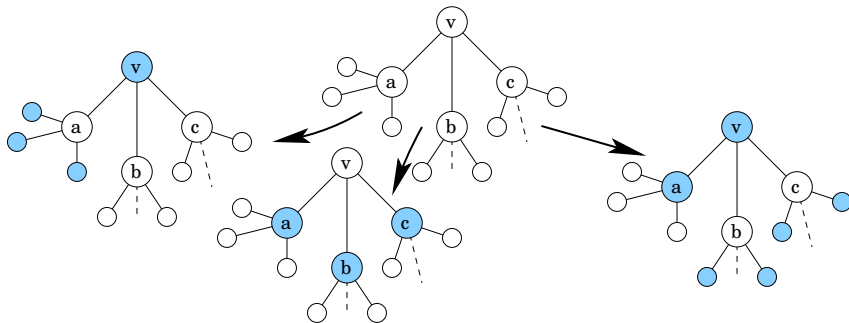
⇒ Verzweigungsvektor:  $(2, 3)$

Sicher: alle Knoten haben Grad 3 oder 4

**Regel 4:** Es existiert ein Knoten  $v$  mit  $\deg(v) = 3$ .

**Fall 3:**  $v$  ist in keinem Dreieck oder Viereck enthalten

**Fall a:**  $\deg(a) = 4$  (analog für  $b$  oder  $c$ )



⇒ Ein Vertex-Cover enthält  $a$  oder alle Nachbarn von  $a$ .

⇒ Ist  $a$  und ( $b$  oder  $c$ ) enthalten: sinnlos  $v$  zu wählen

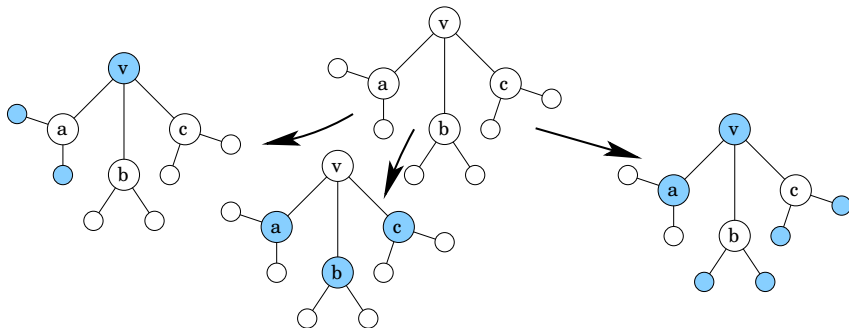
⇒ Verzweigungsvektor: (3, 4, 6)

Sicher: alle Knoten haben Grad 3 oder 4

**Regel 4:** Es existiert ein Knoten  $v$  mit  $\deg(v) = 3$ .

**Fall 3:**  $v$  ist in keinem Dreieck oder Viereck enthalten

**Fall b:**  $\deg(a) = \deg(b) = \deg(c) = 3$  (sonst)

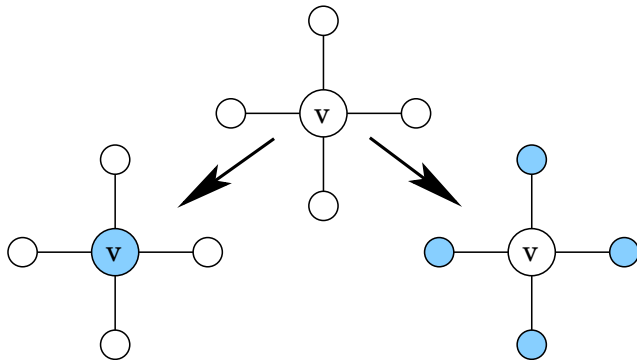


⇒ Ein Vertex-Cover enthält  $a$  oder alle Nachbarn von  $a$ .

⇒ Ist  $a$  und  $(b$  oder  $c)$  enthalten: sinnlos  $v$  zu wählen

⇒ Verzweigungsvektor:  $(3, 3, 6)$

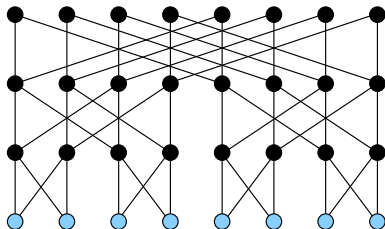
**Regel 5:** Alle Knoten haben Grad vier.



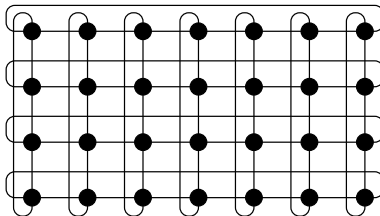
- Wähle entweder  $v$  oder alle seine Nachbarn.
- ⇒ Verzweigungsvektor  $(1, 4)$
- Danach enthält jeder Subgraph mindestens einen Knoten mit kleinerem Grad!
- ⇒ Die Regel wird pro Pfad nur einmal ausgeführt.

Es gibt reguläre Graphen vom Grad 4:

Wrapped Butterfly Network:

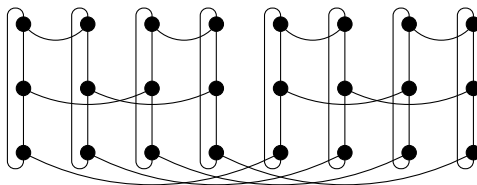


Torus:



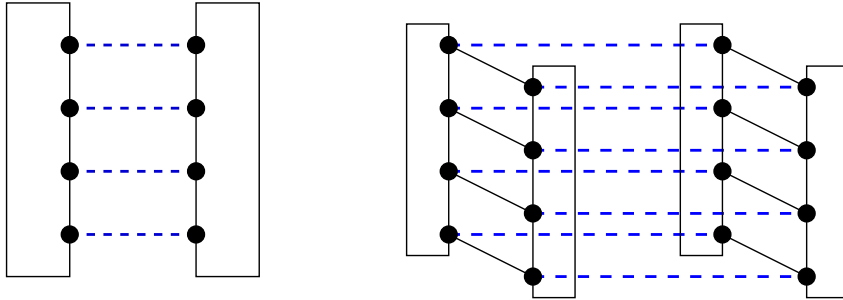
Knoten der oberen und unteren Reihe des Butterfly sind identisch.

Regulärer Graph vom Grad 3: Cube Connected Cycles





Aus zwei Kopien eines  $k$ -regulären Graphen kann ein  $k + 1$ -regulärer Graph erstellt werden:



# Integer Programming for Vertex Cover

minimize

$$\sum_{v \in V} x_v$$

subject to

$$\begin{aligned} x_u + x_v &\geq 1 & \forall \{u, v\} \in E \\ x_u &\in \{0, 1\} & \forall u \in V \end{aligned}$$

## *Exakte Algorithmen für schwere Probleme*

- 3-SAT
- Vertex-Cover
- *Independent Set*

## Tiefenbeschränkte Suchbäume

- nicht immer zum Lösen von schweren Problemen geeignet
- bei Maximierungsproblemen oft ungeeignet

Bei Maximum Independent Set ist eine möglichst große Teilmenge gesucht:

- der Suchbaum müsste die Tiefe  $n$  haben
- das würde aber zu einer großen Laufzeit führen

Sei  $v$  ein beliebiger Knoten. Dann sei  $\hat{N}(v)$  die Menge der Nachbarn von  $v$  inklusive  $v$  selbst.

*Algorithmus B1:* Falls ein Knoten  $v$  ins I.S. aufgenommen wird, dann sind alle Nachbarn von  $v$  nach Definition nicht im I.S.

```
function MAXINDEPENDENTSET( $G = (V, E)$ )  
  if  $E = \emptyset$  then  
    return  $|V|$   
  let  $v$  be some arbitrary node in  $G$   
   $a := 1 + \text{MAXINDEPENDENTSET}(G - \hat{N}(v))$   
   $b := \text{MAXINDEPENDENTSET}(G - \{v\})$   
  return  $\max\{a, b\}$ 
```

Laufzeit:  $T(n) = 2 \cdot T(n-1) + p(n) \in \mathcal{O}^*(2^n)$

Obige Laufzeit ergibt sich, falls  $v$  keinen Nachbarn hat, also  $\hat{N}(v) = \{v\}$  ist. Dann sind beide rekursiven Aufrufe gleich.

Betrachten wir diesen Worst-Case genauer:

- Wenn  $\hat{N}(v) = \{v\}$  ist, also  $v$  keine Nachbarn hat, dann gehört  $v$  zu jedem Maximum Independent Set und wir können auf den zweiten rekursiven Aufruf verzichten.
- Ansonsten hat  $v$  mindestens einen Nachbarn, und  $G - \hat{N}(v)$  hat höchstens  $n - 2$  Knoten.

Wir können daher genauer abschätzen, mit unseren guten, alten Freunden, den Fibonacci-Zahlen:

$$T(n) = \max \left\{ \begin{array}{l} T(n-1) \\ T(n-1) + T(n-2) \end{array} \right\} + p(n) \in \mathcal{O}^*(1, 6181^n)$$

Neuer Worst-Case:  $v$  hat genau einen Nachbarn  $w$ .

- Entweder ist  $v$  oder  $w$  im Independent Set, aber nicht beide.
- Falls ein Independent Set  $w$  enthält, dann gibt es ein ebenso großes Independent Set, das statt  $w$  den Knoten  $v$  enthält.
- Wir können also davon ausgehen, dass es ein Maximum Independent Set gibt, das den Knoten  $v$  enthält.
- Daher muss der Graph  $G - \{v\}$  gar nicht untersucht werden, es reicht aus, den Graphen  $G - \hat{N}(v)$  zu untersuchen, der höchstens  $n - 2$  Knoten hat.

Falls  $v$  mehr als einen Nachbarn hat, dann hat  $G - \hat{N}(v)$  höchstens  $n - 3$  Knoten, und wir schätzen ab:

$$T(n) = \max \left\{ \begin{array}{l} T(n-1) \\ T(n-2) \\ T(n-1) + T(n-3) \end{array} \right\} + p(n) \in \mathcal{O}^*(1,4658^n)$$

Neuer Worst-Case: Jeder Knoten hat Grad mindestens 2.

- Falls  $G$  einen Knoten  $v$  vom Grad mindestens drei hat, dann hat  $G - \hat{N}(v)$  höchstens  $n - 4$  Knoten.
- Sonst hat jeder Knoten den Grad zwei. Sei  $u, v, w$  ein Weg in  $G$ . Dann gilt für jedes maximale Independent Set:
  - Entweder  $v$  ist enthalten, aber  $u$  und  $w$  nicht, oder
  - $u$  ist enthalten, aber seine beiden Nachbarn nicht, oder
  - $w$  ist enthalten, aber seine beiden Nachbarn nicht.

In allen drei Fällen erhalten wir rekursive Aufrufe mit höchstens  $n - 3$  Knoten.

Laufzeit:

$$T(n) = \max \left\{ \begin{array}{l} T(n-1) \\ T(n-2) \\ T(n-1) + T(n-4) \\ 3 \cdot T(n-3) \end{array} \right\} + p(n) \in \mathcal{O}^*(1, 4423^n)$$



Obige Laufzeit wird bestimmt durch den Worst-Case, wo jeder Knoten Grad 2 hat. Schauen wir uns diesen Spezialfall genauer an:

- Wenn jeder Knoten den Grad 2 hat, dann besteht jede Komponente des Graphen aus einem Kreis.
- Ein Kreis der Länge  $k$  hat maximal ein Independent Set der Größe  $\lfloor k/2 \rfloor$ .
- Es ist also gar kein rekursiver Aufruf nötig: Dieser Spezialfall wird direkt in polynomieller Zeit gelöst.

Daher ergibt sich als Laufzeit:

$$T(n) = \max \left\{ \begin{array}{l} T(n-1) \\ T(n-2) \\ T(n-1) + T(n-4) \end{array} \right\} + p(n) \in \mathcal{O}^*(1, 3803^n)$$

```
function MAXINDEPENDENTSET( $G = (V, E)$ )  
  if  $E = \emptyset$  then  
    return  $|V|$   
  else if  $G$  hat Knoten  $v$  mit Grad 0 oder 1 then  
    return  $1 + \text{MAXINDEPENDENTSET}(G - \hat{N}(v))$   
  else if  $G$  hat Knoten  $v$  mit Grad größer als 2 then  
     $a := 1 + \text{MAXINDEPENDENTSET}(G - \hat{N}(v))$   
     $b := \text{MAXINDEPENDENTSET}(G - \{v\})$   
    return  $\max\{a, b\}$   
  else  
     $total := 0$   
    for all Komponente  $K$  von  $G$  do  
       $total := total + \lfloor |K|/2 \rfloor$   
  return  $total$ 
```

Es gibt weitere Verbesserungen, die aber zunehmend komplex sind:

Fomin, Grandoni, Kratsch. Measure and Conquer: A simple  $\mathcal{O}(2^{0,288n})$  independent set algorithm. Proceedings of SODA, 18 – 25, 2006.

Anmerkung:  $2^{0,288n} = 1.2209465^n$

## *Integer Programming for Independent Set*

maximize

$$\sum_{v \in V} x_v$$

subject to

$$\begin{aligned} x_u + x_v &\leq 1 & \forall \{u, v\} \in E \\ x_u &\in \{0, 1\} & \forall u \in V \end{aligned}$$

- Einleitung
- *Entwurfsmethoden*
- Sortieren
- Auswahlproblem
- Graphalgorithmen
- Spezielle Graphklassen
- Vorrangwarteschlangen
- Suchbäume
- Amortisierte Laufzeitanalyse
- Algorithmen für moderne Hardware
- Algorithmen für geometrische Probleme
- Randomisierte Algorithmen

## *Entwurfsmethoden*

- *divide and conquer*
- dynamic programming
- greedy
- local search

## *Entwurfsprinzip:*

- *Divide* the problem into subproblems.
- *Conquer* the subproblems by solving them recursively.
- *Combine* subproblem solutions.

## *Beispiele:*

- Binäre Suche
- Potenzieren einer Zahl
- Matrix-Multiplikation
- Quicksort

*Problem:* Berechne  $x^n$  für ein  $n \in \mathbb{N}$ .

- *Einfacher Algorithmus:*

```
erg := 1
for i := 1 to n do
    erg := erg * x
```

→ Laufzeit:  $\Theta(n)$  Multiplikationen

- *Divide & Conquer:*

$$x^n = \begin{cases} x^{\frac{n}{2}} \cdot x^{\frac{n}{2}} & \text{falls } n \text{ gerade} \\ x^{\frac{n-1}{2}} \cdot x^{\frac{n-1}{2}} \cdot x & \text{sonst} \end{cases}$$

→ Laufzeit:  $T(n) = T(n/2) + c \in \mathcal{O}(???)$

## Induktive Einsetzungsmethode

- Rate eine Lösung und bestätige diese durch vollst. Induktion.
- *Beispiel:* Für  $T(n) = 2 \cdot T(n/2) + b$  "raten" wir  $T(n) \leq c \cdot n - a$  als Lösung und rechnen nach:

$$\begin{aligned}T(n) = 2 \cdot T(n/2) + b &\leq 2 \cdot \left(c \cdot \frac{n}{2} - a\right) + b \\&\leq c \cdot n - 2a + b \\&= c \cdot n - a + (b - a) \\&\leq c \cdot n - a \text{ für } b \leq a\end{aligned}$$

- Problem ist falsches Raten. Nehmen wir bei obigem Beispiel  $T(n) \in \mathcal{O}(n)$  an, also  $T(n) \leq c \cdot n$ , dann erhalten wir:

$$\begin{aligned}T(n) = 2 \cdot T(n/2) + b &\leq 2 \cdot \left(c \cdot \frac{n}{2}\right) + b \\&\leq c \cdot n + b \not\leq\end{aligned}$$



## Iterative Methode

- Setze Rekursionsgleichung fort bis zu einer geschlossenen Form.
- *Beispiel:*  $T(n) = 2 \cdot T(n/2) + b$

$$\begin{aligned}T(n) &= 2 \cdot T(n/2) + b \\&= 2 \cdot (2 \cdot T(n/4) + b) + b = 4 \cdot T(n/4) + 3b \\&= 4 \cdot (2 \cdot T(n/8) + b) + 3b = 8 \cdot T(n/8) + 7b \\&\vdots \\&= 2^k \cdot T(n/2^k) + (2^k - 1) \cdot b\end{aligned}$$

Für  $n = 2^k \iff \log_2(n) = k$  und  $T(1) = \text{const}$  erhalten wir:

$$T(n) = n \cdot T(1) + (n - 1) \cdot b \in \mathcal{O}(n)$$

## Variablensubstitution

- Ersetze  $n$  durch einen Ausdruck, so dass die Rekursionsgleichung eine bekannte Form bekommt.
- *Beispiel:* Für  $T(n) = 2 \cdot T(\sqrt{n}) + \log_2(n)$  mit der Substitution  $m = \log_2(n) \iff 2^m = n$  erhalten wir

$$\begin{aligned} T(2^m) &= 2 \cdot T(\sqrt{2^m}) + m \\ &= 2 \cdot T(2^{\frac{m}{2}}) + m \end{aligned}$$

Löse Gleichung mit einer der ersten beiden Methoden.

*Eingabe:* zwei  $n \times n$ -Matrizen  $A$  und  $B$

*Ausgabe:*  $C = A \cdot B$

Es gilt:

$$\begin{pmatrix} c_{11} & \cdots & c_{1n} \\ c_{21} & \cdots & c_{2n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & \cdots & a_{2n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & \cdots & b_{1n} \\ b_{21} & \cdots & b_{2n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{pmatrix}$$

mit

$$c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}$$

## *Einfacher Algorithmus:*

```
for i := 1 to n do
  for j := 1 to n do
    c[i][j] := 0
    for k := 1 to n do
      c[i][j] := c[i][j] + a[i][k] * b[k][j]
```

→ Laufzeit:  $\Theta(n^3)$  Additionen/Multiplikationen

Die Laufzeit ergibt sich aus der Tabellengröße mal Aufwand pro Eintrag: Die Tabelle hat  $n^2$  Einträge, für jeden Eintrag wird eine Summe über  $n$  viele Produkte gebildet.

Aufteilen der  $n \times n$ -Matrizen in jeweils vier  $\frac{n}{2} \times \frac{n}{2}$ -Matrizen:

$$\left( \begin{array}{c|c} r & s \\ \hline t & u \end{array} \right) = \left( \begin{array}{c|c} a & b \\ \hline c & d \end{array} \right) \cdot \left( \begin{array}{c|c} e & f \\ \hline g & h \end{array} \right)$$
$$C = A \cdot B$$

mit

$$r = ae + bg$$

$$s = af + bh$$

$$t = ce + dg$$

$$u = cf + dh$$

$\Rightarrow$  8 Multiplikationen von  $\frac{n}{2} \times \frac{n}{2}$ -Matrizen  
4 Additionen von  $\frac{n}{2} \times \frac{n}{2}$ -Matrizen

$\rightarrow$  Laufzeit:  $T(n) = 8 \cdot T(n/2) + 4 \cdot (n/2)^2 = \dots \in \Theta(n^3)$

$$\left( \begin{array}{c|c} r & s \\ \hline t & u \end{array} \right) = \left( \begin{array}{c|c} a & b \\ \hline c & d \end{array} \right) \cdot \left( \begin{array}{c|c} e & f \\ \hline g & h \end{array} \right)$$

mit

$$P_1 = a \cdot (f - h)$$

$$P_2 = (a + b) \cdot h$$

$$P_3 = (c + d) \cdot e$$

$$P_4 = d \cdot (g - e)$$

$$P_5 = (a + d) \cdot (e + h)$$

$$P_6 = (b - d) \cdot (g + h)$$

$$P_7 = (a - c) \cdot (e + f)$$

und

$$r = P_5 + P_4 - P_2 + P_6$$

$$s = P_1 + P_2$$

$$t = P_3 + P_4$$

$$u = P_5 + P_1 - P_3 - P_7$$

→ Laufzeit:  $T(n) = 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 \in \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$

Vergleich der Laufzeiten unter der Annahme, dass ein Rechner  $20 \cdot 10^9$  charakteristische Operationen pro Sekunde ausführen kann:

$n$	$n^3$	time	$n^{2.807}$	time	$n^{2.397}$	time
1e3	1e9	<1 s	2.64e8	<1 s	1.55e7	<1 s
10e3	1e12	50 s	1.69e11	9 s	3.87e9	<1 s
100e3	1e15	14 h	1.08e14	90 m	9.66e11	49 s
1.000e3	1e18	2 j	6.95e16	40 t	2.41e14	201 m
10.000e3	1e21	1.585 j	4.46e19	71 j	6.01e16	35 t

s: Sekunden, m: Minuten, h: Stunden, t: Tage, j: Jahre

Beachte: Obige Laufzeitabschätzungen zur Matrixmultiplikation zählen nur die Anzahl der Multiplikationen und Additionen. Wir berücksichtigen dabei nicht die Größe der Zahlen!

- Addition zweier Zahlen der Länge  $\ell$  nach Schulmethode:  $\mathcal{O}(\ell)$
- Multiplikation zweier Zahlen der Länge  $\ell$  nach
  - Schulmethode:  $\mathcal{O}(\ell^2)$
  - Karazuba:  $\mathcal{O}(\ell^{\log_2(3)}) = \mathcal{O}(\ell^{1.5849})$
  - Schönhage/Strassen:  $\mathcal{O}(\ell \cdot \log(\ell) \cdot \log(\log(\ell)))$

Die Laufzeit der einfachen Matrix-Multiplikation unter Berücksichtigung der Größe der Zahlen ergäbe nach

- Schulmethode:  $\mathcal{O}(n^3 \cdot \ell^2)$
- Karazuba:  $\mathcal{O}(n^3 \cdot \ell^{\log_2(3)})$
- Schönhage/Strassen:  $\mathcal{O}(n^3 \cdot \ell \cdot \log(\ell) \cdot \log(\log(\ell)))$



Die Zifferntupel seien  $X = (x_{2n-1} \dots x_0)_b$  und  $Y = (y_{2n-1} \dots y_0)_b$ .

Jedes Zifferntupel aufspalten in zwei Tupel der Länge  $n$ :

$$\begin{aligned} X_h &= (x_{2n-1} \dots x_n)_b \quad \text{und} \quad X_l = (x_{n-1} \dots x_0)_b \\ Y_h &= (y_{2n-1} \dots y_n)_b \quad \text{und} \quad Y_l = (y_{n-1} \dots y_0)_b. \end{aligned}$$

Damit ist  $X = X_h b^n + X_l$  und  $Y = Y_h b^n + Y_l$  und wir erhalten

$$XY = X_h Y_h b^{2n} + (X_h Y_l + X_l Y_h) b^n + X_l Y_l.$$

Den Term  $X_h Y_l + X_l Y_h$  in andere Form bringen:

$$\begin{aligned} X_h Y_l + X_l Y_h &= (X_h Y_h + X_h Y_l + X_l Y_h + X_l Y_l) - (X_h Y_h + X_l Y_l) \\ &= (X_h + X_l)(Y_h + Y_l) - (X_h Y_h + X_l Y_l). \end{aligned}$$

Dann sind im Produkt nur noch drei Produkte enthalten:

$$XY = X_h Y_h b^{2n} + ((X_h + X_l)(Y_h + Y_l) - (X_h Y_h + X_l Y_l)) b^n + X_l Y_l$$

*Master-Theorem:*

$$T(n) = a \cdot T(n/b) + \Theta(n^k) \quad \text{mit} \quad T(1) \in \mathcal{O}(1)$$

Unterscheide drei Fälle:

- für  $a < b^k$  gilt:  $T(n) \in \Theta(n^k)$
- für  $a = b^k$  gilt:  $T(n) \in \Theta(n^k \cdot \log(n))$
- für  $a > b^k$  gilt:  $T(n) \in \Theta(n^{\log_b a})$

*Herleitung:* → nächste Folien

Wir nutzen die iterative Methode und erhalten:

$$\begin{aligned}T(n) &= a \cdot T(n/b) + cn^k \\&= a \cdot [a \cdot T(n/b^2) + c(n/b)^k] + cn^k \\&= a^2 \cdot T(n/b^2) + cn^k \cdot (1 + a/b^k) \\&= a^2 \cdot [a \cdot T(n/b^3) + c(n/b^2)^k] + cn^k \cdot (1 + a/b^k) \\&= a^3 \cdot T(n/b^3) + cn^k \cdot (1 + a/b^k + (a/b^k)^2) \\&\vdots \\&= a^\ell \cdot T(n/b^\ell) + cn^k \cdot \sum_{i=0}^{\ell-1} (a/b^k)^i\end{aligned}$$

mit  $n/b^\ell = 1 \iff n = b^\ell$  und somit  $\ell = \log_b(n)$  gilt:

$$T(n) = a^{\log_b(n)} \cdot T(1) + cn^k \cdot \sum_{i=0}^{\log_b(n)-1} (a/b^k)^i$$

Wir hatten festgestellt:

$$T(n) = a^{\log_b(n)} \cdot T(1) + cn^k \cdot \sum_{i=0}^{\log_b(n)-1} (a/b^k)^i$$

*Fall 1:  $a = b^k$*

$$\begin{aligned} T(n) &= a^{\log_b(n)} + cn^k \cdot \sum_{i=0}^{\log_b(n)-1} 1^i \\ &= (b^k)^{\log_b(n)} + cn^k \cdot \log_b(n) \\ &= (b^{\log_b(n)})^k + cn^k \cdot \log_b(n) \\ &= n^k + cn^k \cdot \log_b(n) \\ &\in \Theta(n^k \cdot \log(n)) \end{aligned}$$

Wir hatten festgestellt:

$$T(n) = a^{\log_b(n)} \cdot T(1) + cn^k \cdot \sum_{i=0}^{\log_b(n)-1} (a/b^k)^i$$

*Fall 2:  $a < b^k$*

$$T(n) = a^{\log_b(n)} + cn^k \cdot \frac{1 - (a/b^k)^{\log_b(n)}}{1 - a/b^k}$$

Es gilt:

- $a^{\log_b(n)} = (b^{\log_b(a)})^{\log_b(n)} = (b^{\log_b(n)})^{\log_b(a)} = n^{\log_b(a)}$  und wegen  $a < b^k$  gilt:  
 $n^{\log_b(a)} < n^{\log_b(b^k)} = n^k$
- $\lim_{n \rightarrow \infty} \frac{1 - (a/b^k)^{\log_b(n)}}{1 - a/b^k} = \frac{1}{1 - a/b^k} = \textit{konstant}$

und daher gilt:  $T(n) \in \Theta(n^k)$

Wir hatten festgestellt:

$$T(n) = a^{\log_b(n)} \cdot T(1) + cn^k \cdot \sum_{i=0}^{\log_b(n)-1} (a/b^k)^i$$

*Fall 3:  $a > b^k$*

$$T(n) = a^{\log_b(n)} + cn^k \cdot \frac{(a/b^k)^{\log_b(n)} - 1}{a/b^k - 1}$$

Da  $a/b^k - 1$  konstant ist, können wir eine neue Konstante  $\hat{c}$  einführen und es gilt:  
 $T(n) = n^{\log_b(a)} + \hat{c}n^k \cdot ((a/b^k)^{\log_b(n)} - 1)$

Außerdem gilt:

$$\begin{aligned} (a/b^k)^{\log_b(n)} &= a^{\log_b(n)} / (b^k)^{\log_b(n)} \\ &= n^{\log_b(a)} / (b^{\log_b(n)})^k \\ &= n^{\log_b(a)} / n^k \end{aligned}$$

Damit erhalten wir:

$$\begin{aligned}T(n) &= n^{\log_b(a)} + \hat{c}n^k \cdot ((a/b^k)^{\log_b(n)} - 1) \\&= n^{\log_b(a)} + \hat{c}n^k \cdot ((n^{\log_b(a)})/n^k) - 1) \\&= n^{\log_b(a)} + \hat{c} \cdot n^{\log_b(a)} - \hat{c}n^k\end{aligned}$$

Wegen  $a > b^k$  gilt  $n^{\log_b(a)} > n^{\log_b(b^k)} = n^k$ , und wir erhalten schließlich:

$$T(n) \in \Theta(n^{\log_b(a)})$$

*Beispiele:*

- *Binäre Suche:*  $a = 1, b = 2, k = 0$

$$a = b^k \rightarrow T(n) = \Theta(n^k \cdot \log(n)) = \Theta(\log(n))$$

- *Matrix-Multiplikation:*  $a = 8, b = 2, k = 2$

$$a > b^k \rightarrow T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$$

- *Merge-Sort:*  $a = 2, b = 2, k = 1$

$$a = b^k \rightarrow T(n) = \Theta(n^k \cdot \log(n)) = \Theta(n \cdot \log(n))$$

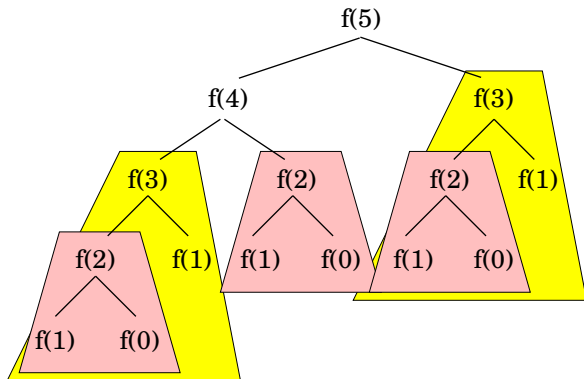


## *Entwurfsmethoden*

- divide and conquer
- *dynamic programming*
- greedy
- local search

*Motivation:* berechne Fibonacci-Zahlen rekursiv, top down

```
long fibo(int n) {  
    long f1, f2;  
  
    if (n <= 1)  
        return n;  
  
    f1 = fibo(n-1);  
    f2 = fibo(n-2);  
    return f1 + f2;  
}
```



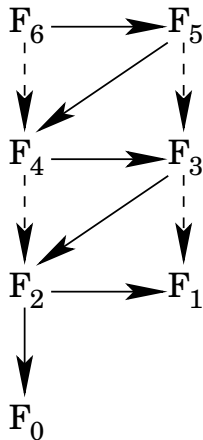
*Problem:* Zwischenlösungen werden mehrfach berechnet!

*Lösung:* Speichere bereits berechnete Zwischenlösungen in einer Tabelle. Diese Technik nennt man *Memorieren*. Es ist immer noch ein Top-Down-Ansatz.

<pre>long fibo(int n) {     long f1, f2;      if (fib[n-1] &gt;= 0)         f1 = fib[n-1];     else {         f1 = fibo(n-1);         fib[n-1] = f1;     } }</pre>		<pre>    if (fib[n-2] &gt;= 0)         f2 = fib[n-2];     else {         f2 = fibo(n-2);         fib[n-2] = f2;     }      return f1 + f2; }</pre>
--	--	--

*Frage:* Wie kann das Programm vereinfacht werden?

Dadurch ergibt sich folgende Struktur der rekursiven Aufrufe:



Gestrichelte Linien stellen den Zugriff auf bereits gespeicherte, memorierte Werte dar.

*Bottom-Up-Ansatz:* Aus Rekursion wird Iteration, indem aus kleinen Teillösungen größere Lösungen berechnet werden.

```
long fibo(int n) {  
    int i;  
  
    fib[0] = 0;  
    fib[1] = 1;  
    for (i = 2; i <= n; i++)  
        fib[i] = fib[i-1] + fib[i-2];  
  
    return fib[n];  
}
```

Untypisches Beispiel für dynamische Programmierung: es fehlt die Optimierung!

# Matrix-Kettenmultiplikation

**Gegeben:**  $n$  Matrizen  $M_1, \dots, M_n$ , wobei  $M_i$  eine  $p_{i-1} \times p_i$  Matrix ist, also  $p_{i-1}$  Zeilen und  $p_i$  Spalten hat.

**Gesucht:** Eine Klammerung, so dass  $M_1 \cdot \dots \cdot M_n$  mit minimaler Anzahl an skalaren Multiplikationen erfolgt.

**Anmerkung:** Multiplikation  $p \times q$  Matrix mit  $q \times r$  Matrix erfordert  $p \cdot q \cdot r$  Multiplikationen und liefert  $p \times r$  Matrix.

**Beispiel:**  $M_1, M_2, M_3$  haben die Dimensionen  $(50 \times 10)$ ,  $(10 \times 20)$  und  $(20 \times 5)$ . Dann sind bei

- $(M_1 \cdot M_2) \cdot M_3 \rightarrow 50 \cdot 10 \cdot 20 + 50 \cdot 20 \cdot 5 = 15.000$
- $M_1 \cdot (M_2 \cdot M_3) \rightarrow 10 \cdot 20 \cdot 5 + 50 \cdot 10 \cdot 5 = 3.500$

Multiplikationen erforderlich!

*Naiv:* Ausprobieren aller möglichen Klammerungen.

$$K_n = \sum_{j=1}^{n-1} K_j \cdot K_{n-j} \quad \text{mit} \quad K_1 = 1$$

Erklärung:

- Es gibt  $n - 1$  mögliche äußere Klammerungen.
- Bei Aufteilung in  $M_1 \cdots M_j$  und  $M_{j+1} \cdots M_n$  verbleiben  $K_j$  bzw.  $K_{n-j}$  innere Klammerungen.

Lösung der Rekursionsgleichung: Catalansche Zahlen

$$K_{n+1} = C(n) = \binom{2n}{n} \frac{1}{n+1} \in \Omega\left(\frac{4^n}{n^{1.5}}\right)$$

Bezeichne  $m(i, j)$  die minimale Anzahl skalarer Multiplikationen bei optimaler Klammerung des Abschnitts  $M_i \cdots M_j$ .

Sei  $(M_i \cdots M_k) \cdot (M_{k+1} \cdots M_j)$  die optimale Klammerung des Abschnitts  $M_i \cdots M_j$ . Dann gilt:

$$m(i, j) = m(i, k) + m(k + 1, j) + p_{i-1} \cdot p_k \cdot p_j$$

→ **Optimalitätsprinzip**: Eine optimale Lösung kann aus optimalen Teillösungen zusammengesetzt werden.

Da wir die optimale Klammerung nicht kennen, müssen wir alle möglichen Werte für  $k$  ausprobieren und dann den minimalen Wert wählen. Damit ergibt sich folgende Rekursionsformel:

$$m(i, j) = \begin{cases} 0, & i = j \\ \min_{i \leq k < j} (m(i, k) + m(k + 1, j) + p_{i-1} \cdot p_k \cdot p_j), & i < j \end{cases}$$



## Übung:

- 1 Formulieren Sie einen rekursiven Algorithmus zur Lösung der Matrix-Kettenmultiplikation. Welche Laufzeit hat Ihr Algorithmus?
- 2 Wie sieht eine Lösung mittels dynamischer Programmierung aus und welche Laufzeit hat diese Lösung?
- 3 Wie kann der Algorithmus erweitert werden, so dass auch die optimale Klammerung bestimmt wird?

rekursiver Algorithmus zur Matrix-Kettenmultiplikation:

```
function MKM( $i, j$ )  
  if ( $i = j$ ) then  
    return 0  
   $r := \infty$   
  for  $k := i$  to  $j - 1$  do  
     $q := \text{MKM}(i, k) + \text{MKM}(k + 1, j) + p_{i-1} \cdot p_k \cdot p_j$   
    if ( $q < r$ ) then  
       $r := q$   
  return  $r$ 
```

Laufzeit des Algorithmus: exponentiell

Lösung mittels dynamischer Programmierung:

**for**  $i := 1$  to  $n$  **do**

$m[i][i] := 0$

**for**  $\ell := 1$  to  $n - 1$  **do**

**for**  $(i, j)$  where  $j - i = \ell$  **do**

$m[i][j] = \infty$

**for**  $k := i$  to  $j - 1$  **do**

$q := m[i][k] + m[k + 1][j] + p_{i-1} \cdot p_k \cdot p_j$

**if**  $(q < m[i][j])$  **then**

$m[i][j] := q$

Laufzeit dieser Lösung:  $\mathcal{O}(n^3)$

bestimme auch die optimale Klammerung:

```
for  $i := 1$  to  $n$  do
```

```
     $m[i][i] := 0$ 
```

```
for  $\ell := 1$  to  $n - 1$  do
```

```
    for  $i = 1$  to  $n - \ell$  do
```

```
         $j = i + \ell$ 
```

```
         $m[i][j] = \infty$ 
```

```
        for  $k := i$  to  $j - 1$  do
```

```
             $q := m[i][k] + m[k + 1][j] + p_{i-1} \cdot p_k \cdot p_j$ 
```

```
            if ( $q < m[i][j]$ ) then
```

```
                 $m[i][j] := q$ 
```

```
                 $s[i][j] := k$ 
```

Wie kann die Klammerung ausgegeben werden?

## 0/1-Rucksack-Problem

Packe einen Teil von  $n$  Objekten mit den Größen  $g_1, \dots, g_n$  und den Werten  $w_1, \dots, w_n$  so in einen Rucksack der Größe  $G$ , dass der Gesamtwert maximal ist.

*Formal:* Finde einen 0/1-Vektor  $(a_1, \dots, a_n) \in \{0, 1\}^n$  mit

$$\sum_{i=1}^n a_i g_i \leq G \quad \text{und} \quad \sum_{i=1}^n a_i w_i \longrightarrow \max.$$

Formulierung als Integer Programm!

*Rekursiver Algorithmus:* Sei  $knap(h, i)$  der maximale Wert, der mit den Objekten  $i, \dots, n$  und Rucksackgröße  $h$  erreicht werden kann. Dann gilt für  $i < n$  und  $h \geq g_i$ :

$$knap(h, i) = \max\{knap(h, i + 1), w_i + knap(h - g_i, i + 1)\}$$

Sonst gilt:

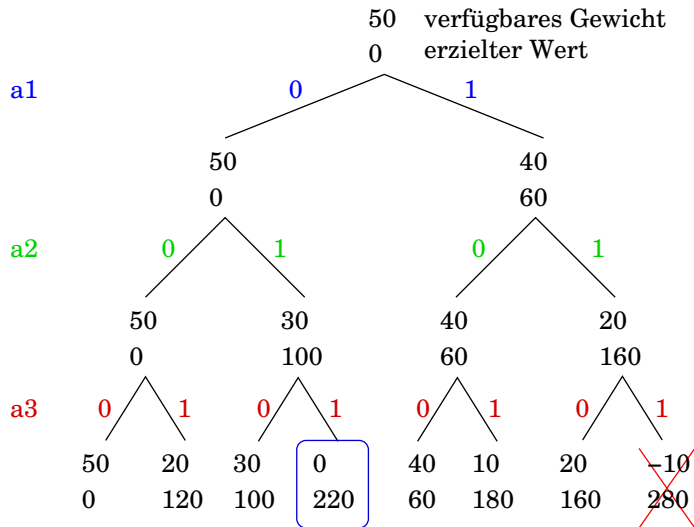
$$knap(h, i) = \begin{cases} knap(h, i + 1), & i < n, h < g_i \\ 0, & i = n, h < g_n \\ w_n, & i = n, h \geq g_n \end{cases}$$

*Übung:*

- ① Implementieren Sie obigen Algorithmus. Laufzeit?
- ② Lösung mittels dynamischer Programmierung? Laufzeit?

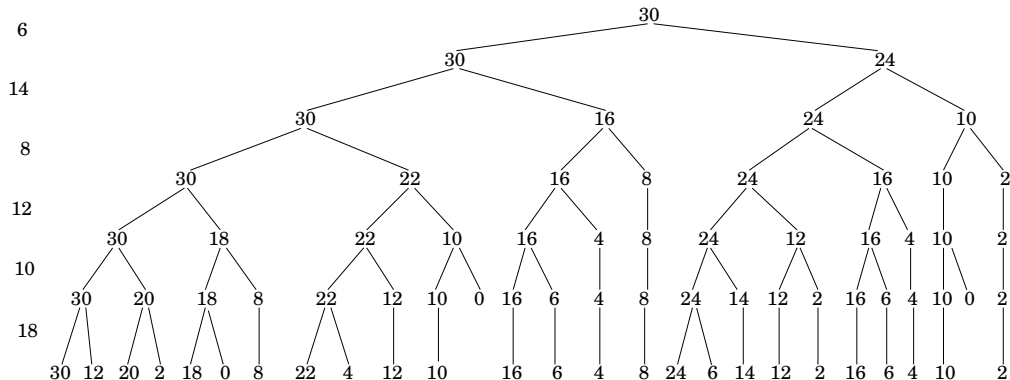
# 0/1-Rucksack-Problem

Beispiel:  $g = (10, 20, 30)$ ,  $w = (60, 100, 120)$ ,  $G = 50$



## Probleme:

- 1 Laufzeit des rekursiven Algorithmus:  $\mathcal{O}(2^n)$
- 2 Memorieren bringt nichts, da nur wenige Teillösungen wiederverwendet werden können. Beispiel: rekursive Aufrufe zu  $g = (6, 14, 8, 12, 10, 18)$  mit  $G = 30$ .





*Versuch:* Bottom-Up-Strategie

```
FOR  $i := n$  DOWN TO 1 DO  
  FOR  $h := 0$  TO  $G$  DO  
    berechne  $knap(h, i)$  nach obiger Formel
```

→ Laufzeit  $\in \mathcal{O}(n \cdot G)$

*Beispiel:*  $g = (1, 2, 3)$ ,  $w = (6, 10, 12)$ ,  $G = 5$

$i \backslash h$	0	1	2	3	4	5
3	0	0	0	12	12	12
2	0	0	10	12	12	22
1	0	6	10	16	18	22

## Übung:

- Welcher Algorithmus ist besser? Der rekursive mit Laufzeit  $\mathcal{O}(2^n)$  oder der mittels dynamischer Programmierung und Laufzeit  $\mathcal{O}(n \cdot G)$ ?
- Ist die Laufzeit von  $\mathcal{O}(n \cdot G)$  nicht ein Widerspruch zur Tatsache, dass das 0/1-Rucksack-Problem NP-vollständig ist?

Ist der rekursive Algorithmus mit Laufzeit  $\mathcal{O}(2^n)$  oder der mittels dynamischer Programmierung und Laufzeit  $\mathcal{O}(n \cdot G)$  besser?

Das ist abhängig davon, wie groß  $n$  und  $G$  sind:

- Falls  $G \in \mathcal{O}(n^c)$  für ein konstantes  $c$  gilt, dann ist die Laufzeit der dynamischen Programmierung mit  $\mathcal{O}(n^{c+1})$  asymptotisch viel besser als die Laufzeit  $\mathcal{O}(2^n)$  des rekursiven Algorithmus.
- Falls die Gewichte groß sind, also insbesondere  $G \in \mathcal{O}(2^n)$  gilt, dann ist die Laufzeit beider Algorithmen schlecht.

Die Laufzeit  $\mathcal{O}(n \cdot G)$  ist also kein Widerspruch zur Tatsache, dass das 0/1-Rucksack-Problem NP-vollständig ist. Die Laufzeiten bei Turingmaschinen beziehen sich immer auf die Größe der Eingabe, und dabei spielen die Zahlenwerte eine entscheidende Rolle.

$$|G| = \log_2(G) \rightarrow \mathcal{O}(n \cdot G) = \mathcal{O}(n \cdot 2^{\log_2(G)}) = \mathcal{O}(n \cdot 2^{|G|})$$

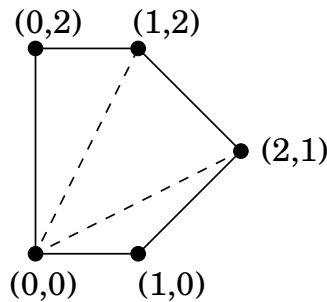
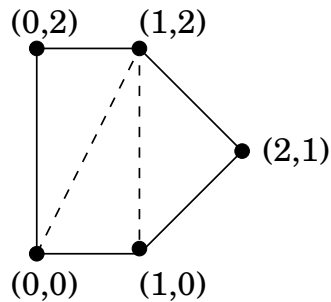
# Polygon Triangulierung

**Gegeben:** Ein konvexes Polygon aus Punkten  $(p_1, \dots, p_n)$ .

**Gesucht:** Eine Teilmenge  $B$  der Menge aller Saiten (auch Sehnen genannt) mit den Eigenschaften:

- Keine zwei Saiten aus  $B$  kreuzen sich und
- die Saiten aus  $B$  teilen das Polygon in Dreiecke.

*Beispiel:*



Sei  $w(i, j, k)$  der Umfang des Dreiecks  $\triangle p_i p_j p_k$ . Die Kosten einer Triangulierung ist die Summe über den Umfang aller Dreiecke.

Im Beispiel:

$$\begin{array}{r} (1 + 2 + \sqrt{5}) \\ + (1 + 2 + \sqrt{5}) \\ + (2 + \sqrt{2} + \sqrt{2}) \\ \hline = 8 + 2\sqrt{2} + 2\sqrt{5} \\ \approx 15.30 \end{array}$$

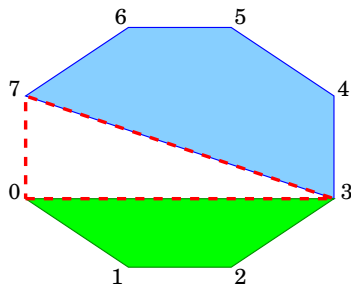
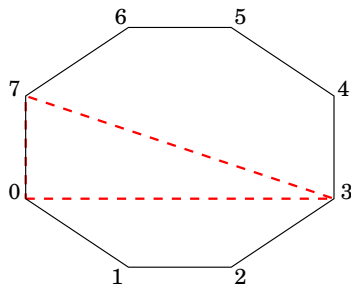
$$\begin{array}{r} (1 + 2 + \sqrt{5}) \\ + (\sqrt{2} + \sqrt{5} + \sqrt{5}) \\ + (1 + \sqrt{2} + \sqrt{5}) \\ \hline = 4 + 2\sqrt{2} + 4\sqrt{5} \\ \approx 15.77 \end{array}$$

*Übung:*

- Formulieren Sie einen rekursiven Algorithmus für das Problem. Welche Laufzeit hat Ihr Algorithmus?
- Lösung mittels dynamischer Programmierung? Laufzeit?

Die Kante von 0 nach  $n$  gehört auf jeden Fall zu einem Dreieck. Da wir die optimale Triangulierung und damit den anderen Knoten des Dreiecks nicht kennen, testen wir alle möglichen Dreiecke.

Bezeichne  $c(i, k)$  die Kosten der minimalen Triangulierung des Teilpolygons mit den Knoten  $(i, i+1, i+2, \dots, k, i)$ .



Um die Kosten  $c(0, 7)$  zu berechnen, testen wir unter anderem das Dreieck aus den Knoten 0, 3 und 7 sowie die sich daraus ergebenden Teilpolygone  $(0, 1, 2, 3, 0)$  und  $(3, 4, 5, 6, 7, 3)$ .

allgemein:

$$c(i, k) = \begin{cases} 0 & \text{falls } i + 2 > k \\ \min_{i < j < k} \left\{ c(i, j) + c(j, k) + w(i, j, k) \right\} & \text{sonst} \end{cases}$$

rekursiver Algorithmus:

```
function PT( $i, k$ )  
  if ( $i + 2 > k$ ) then  
    return 0  
   $r := \infty$   
  for  $j := i + 1$  to  $k - 1$  do  
     $q := \text{PT}(i, j) + \text{PT}(j, k) + w(i, j, k)$   
    if ( $q < r$ ) then  
       $r := q$   
  return  $r$ 
```

Laufzeit: wie bei Matrix-Kettenmultiplikation

dynamische Programmierung:

```
for  $i := 1$  to  $n$  do  
     $c[i][i] := 0$   
for  $\ell := 2$  to  $n$  do  
    for  $i = 1$  to  $n - \ell$  do  
         $k = i + \ell$   
         $c[i][k] = \infty$   
        for  $j := i + 1$  to  $k - 1$  do  
             $q := c[i][j] + c[j][k] + w(i, j, k)$   
            if  $(q < c[i][k])$  then  
                 $c[i][k] := q$   
                 $s[i][k] := j$ 
```

Laufzeit: wie bei Matrix-Kettenmultiplikation



Die Levenshtein-Distanz zweier Wörter  $a$  und  $b$  wird auch als Edit-Distanz, Editierdistanz oder Editierabstand bezeichnet.

Sie bezeichnet ein Maß für den Unterschied zwischen zwei Zeichenketten bezüglich der minimalen Anzahl der Operationen Einfügen, Löschen und Ersetzen, um die eine Zeichenkette in die andere zu überführen.

*Beispiel:* Die Edit-Distanz von Tier und Tor ist zwei.

$\text{Tier} \rightarrow \text{Toer} (\text{ersetze i durch o}) \rightarrow \text{Tor} (\text{Lösche e})$

In der Praxis: Bestimmen der Ähnlichkeit von Zeichenketten zur Rechtschreibprüfung oder bei einer Duplikaterkennung.

*Rekursiver Algorithmus:* Sei  $D_{i,j}$  der minimale Editierabstand zwischen den Teilwörtern bis Position  $i$  bzw.  $j$ . Dann gilt initial:

$$D_{0,0} = 0, \quad D_{0,j} = j, \quad D_{i,0} = i$$

Außerdem gilt:

$$D_{i,j} = \min \left\{ \begin{array}{ll} D_{i,j-1} + 1 & \text{einfügen} \\ D_{i-1,j} + 1 & \text{löschen} \\ D_{i-1,j-1} + c & \text{sonst} \end{array} \right\}$$

Dabei ist  $c = 0$ , falls  $a_i = b_j$  gilt, sonst ist  $c = 1$  (ersetze  $a_i$  durch  $b_j$ ).

*Übung:*

- 1 Gilt das Optimalitätsprinzip?
- 2 Implementieren Sie obigen rekursiven Algorithmus und eine Lösung mittels dynamischer Programmierung.
- 3 Vergleichen Sie die beiden Laufzeiten für einige Beispiele.

Ein Editierpfad von  $x$  nach  $y$  ist eine Folge

$$x = x_0, x_1, x_2, \dots, x_\ell = y,$$

so dass  $x_{i+1} = \omega(x_i)$  für eine geeignete Operation  $\omega$  gilt.

*Optimalitätsprinzip nach Bellman:* Die optimale Lösung eines Teilproblems (der Größe  $n$ ) setzt sich aus optimalen Lösungen kleinerer Teilprobleme zusammen.

Wenn der Editierpfad  $P$  von  $x_0$  nach  $x_\ell$

$$x_0, x_1, x_2, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_j, x_{j+1}, \dots, x_\ell$$

optimal ist, dann ist auch der Editierpfad  $P'$  von  $x_i$  nach  $x_j$  optimal, denn sonst könnte  $P'$  ersetzt werden durch einen besseren Pfad und damit würde auch  $P$  verbessert werden.  $\nmid$

```
int distance_rek(int i, int j) {  
    if (i == 0) return j;  
    if (j == 0) return i;  
  
    int min = distance_rek(i-1, j-1);  
    if (s[i-1] != t[j-1])  
        min += 1;  
  
    int tmp = distance_rek(i-1, j) + 1;  
    if (tmp < min)  
        min = tmp;  
  
    tmp = distance_rek(i, j-1) + 1;  
    if (tmp < min)  
        min = tmp;  
  
    return min;  
}
```

```
int distance_dyn() {  
    for (int i = 0; i <= n; i++) dist[i][0] = i;  
    for (int j = 0; j <= m; j++) dist[0][j] = j;  
  
    for (int i = 1; i <= n; i++) {  
        for (int j = 1; j <= m; j++) {  
            int min = dist[i-1][j-1];  
  
            if (s[i-1] != t[j-1])  
                min += 1;  
            if (dist[i-1][j] + 1 < min)  
                min = dist[i-1][j] + 1;  
            if (dist[i][j-1] + 1 < min)  
                min = dist[i][j-1] + 1;  
            dist[i][j] = min;  
        }  
    }  
    return dist[n][m];  
}
```

## Erklärung:

- Die Variablen  $s$  und  $t$  verweisen auf die Zeichenketten, die die beiden Wörter enthalten. Dabei speichert  $n$  die Länge von  $s$  und  $m$  die Länge von  $t$ .
- Wenn  $d[i][j]$  berechnet wird, muss auf  $s[i-1]$  und  $t[j-1]$  zugegriffen werden, da Zeichenketten wie  $s = \text{"Tier\0"}$  und  $t = \text{"Tor\0"}$  jeweils bei Index 0 in C und C++ beginnen, und daher der  $x$ -te Buchstabe an Position  $x - 1$  steht.

Für kontextfreie Grammatiken, die in Chomsky-Normalform vorliegen, kann das Wortproblem effizient gelöst werden.

*Wortproblem:*

Ist ein gegebenes Wort  $x$  in der Sprache  $L(G)$ ?

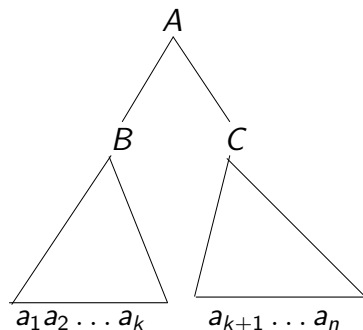
Dabei ist  $G$  eine Grammatik in Chomsky-Normalform und  $L(G)$  die durch die Grammatik  $G$  erzeugte Sprache.

*Chomsky-Normalform:* Kontextfreie Grammatik  $G$  mit  $\epsilon \notin L(G)$ , wobei alle Regeln eine der beiden Formen haben:

- $A \rightarrow BC$
- $A \rightarrow a$

Dabei stehen  $A, B, C$  für Variablen und  $a$  für ein terminales Symbol.

- Ein Wort  $x = a$  der Länge 1 kann nur aus einer Regel der Form  $A \rightarrow a$  abgeleitet werden.
- Ein Wort  $x = a_1 a_2 \dots a_n$  mit  $n \geq 2$  kann nur abgeleitet werden, wenn zunächst eine Regel  $A \rightarrow BC$  angewendet wird.



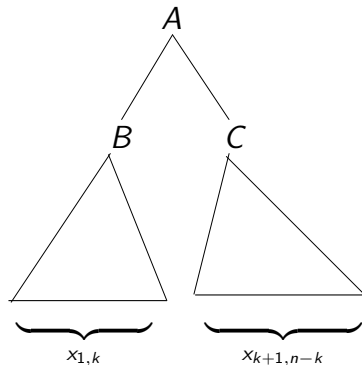
- Von  $B$  aus wird ein Anfangsstück von  $x$  abgeleitet und von  $C$  aus wird das Endstück abgeleitet.
- Dann kann das Wortproblem auf zwei entsprechende Entscheidungen für Wörter der Länge  $k$  und  $n - k$  zurückgeführt werden.

Da wir  $k$  nicht kennen, müssen alle Werte von 1 bis  $n - 1$  in Betracht gezogen werden.



Bezeichne  $x_{i,\ell}$  das Teilwort von  $x$ , das an Position  $i$  beginnt und die Länge  $\ell$  hat.

- In  $T[i, \ell]$  notieren wir, aus welchen Variablen das Wort  $x_{i,\ell}$  abgeleitet werden kann.
- $x = a_1 a_2 \dots a_n$  ist aus  $L(G)$ , falls am Ende  $S \in T[1, n]$  gilt.



$$T[i, \ell] = \bigcup_{k=1, \dots, \ell-1} \{A \mid A \rightarrow BC, B \in T[i, k], C \in T[i+k, \ell-k]\}$$

Eingabe: Ein Wort  $x = a_1 a_2 \dots a_n$  und eine kontextfreie Grammatik  $G = (V, \Sigma, P, S)$  in Chomsky-Normalform.

**for**  $i := 1$  to  $n$  **do**

▷ Teilwörter der Länge 1

$T[i, 1] := \{A \in V \mid A \rightarrow a_i \in P\}$

**for**  $\ell := 2$  to  $n$  **do**

▷ Teilwörter der Länge  $\ell$

**for**  $i := 1$  to  $n + 1 - \ell$  **do**

$T[i, \ell] := \emptyset$

**for**  $k := 1$  to  $\ell - 1$  **do**

$T[i, \ell] := T[i, \ell] \cup \{A \in V \mid A \rightarrow BC \in P \text{ und} \\ B \in T[i, k] \text{ und } C \in T[i + k, \ell - k]\}$

## Übung:

- Geben Sie für die Sprache  $L = \{a^n b^n c^m \mid n, m \geq 1\}$  eine kontextfreie Grammatik  $G$  in Chomsky-Normalform an.
- Zeigen Sie mittels des CYK-Algorithmus, dass das Wort  $x = aaabbbcc$  in der Sprache  $L(G)$  enthalten ist.
- Wie können Mengen gespeichert werden, so dass die Mengen-Operationen des CYK-Algorithmus effizient ausgeführt werden?

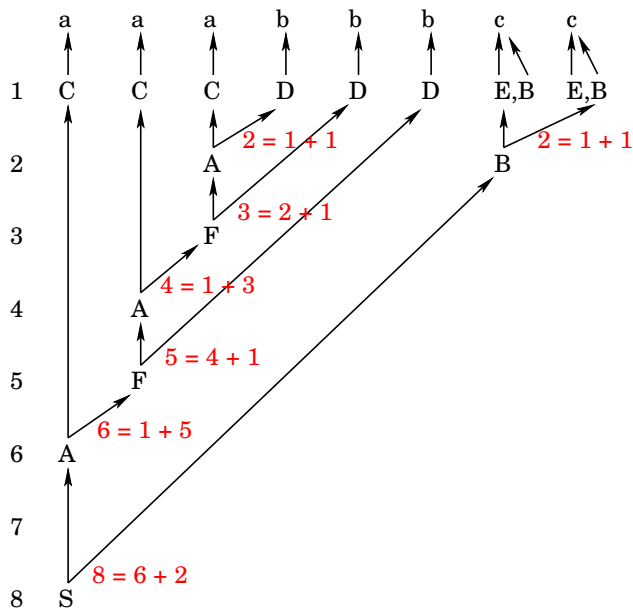
Kontextfreie Grammatik für  $L = \{a^n b^n c^m \mid n, m \geq 1\}$ :

$$\begin{array}{ll} S \rightarrow AB & B \rightarrow cB \\ A \rightarrow aAb & B \rightarrow c \\ A \rightarrow ab \end{array}$$

Umwandeln in Chomsky-Normalform:

$$\begin{array}{ll} S \rightarrow AB & C \rightarrow a \\ A \rightarrow CD & D \rightarrow b \\ A \rightarrow CF & E \rightarrow c \\ B \rightarrow c & F \rightarrow AD \\ B \rightarrow EB \end{array}$$

# Lösung zur Übung



## Systematisches Umwandeln in Chomsky-Normalform:

- Ersetze alle Terminale  $a$  durch nicht-terminale  $X_a$  und füge die Regel  $X_a \rightarrow a$  zur Regelmenge hinzu.

$$\begin{array}{llll} S \rightarrow AB & \Rightarrow & S \rightarrow AB & \checkmark \\ A \rightarrow aAb & \Rightarrow & A \rightarrow X_aAX_b & \\ A \rightarrow ab & \Rightarrow & A \rightarrow X_aX_b & \\ B \rightarrow c & \Rightarrow & B \rightarrow X_c & \\ B \rightarrow cB & \Rightarrow & B \rightarrow X_cB & \\ & & X_a \rightarrow a & \\ & & X_b \rightarrow b & \\ & & X_c \rightarrow c & \end{array}$$

- Ersetze die rechten Seiten, die mehr als 2 nicht-terminale Symbole aufweisen: streiche  $A \rightarrow X_aAX_b$  und füge  $A \rightarrow YX_b$  sowie  $Y \rightarrow X_aA$  hinzu
- Entferne Kettenregeln: streiche  $B \rightarrow X_c$ , füge  $B \rightarrow c$  hinzu

Die Mengen werden mittels eines 3-dimensionalen Arrays realisiert.

```
for  $\ell := 2$  to  $n$  do  
  for  $i := 1$  to  $n$  do  
    for all  $A \in V$  do  
       $M[i, \ell, A] := false$   
  
  for  $i := 1$  to  $n$  do  
    for all  $A \in V$  do  
       $M[i, 1, A] := true$   
  
  for  $\ell := 2$  to  $n$  do  
    for  $i := 1$  to  $n + 1 - \ell$  do  
      for  $k := 1$  to  $\ell - 1$  do  
        for all  $A \rightarrow BC \in P$  do  
          if  $M[i, k, B]$  and  $M[i + k, \ell - k, C]$  then  
             $M[i, \ell, A] := true$ 
```

# Traveling Salesperson Problem

Bestimme zu einem vollständigen Graphen  $G = (V, E, c)$  einen Hamiltonkreis, dessen Gesamtlänge möglichst klein ist. Dabei ist  $c : E \rightarrow \mathbb{N}$  eine Kostenfunktion, die zu jeder Kante in  $E$  deren „Länge“ definiert.

Da der Knoten, in der die Rundreise beginnt, festgelegt ist, hat die erschöpfende Suche bei  $n$  Knoten eine Komplexität von  $(n - 1)!$ .

(1, 2, 3, 4, 1) (1, 3, 2, 4, 1) (1, 4, 3, 2, 1)  
(1, 2, 4, 3, 1) (1, 3, 4, 2, 1) (1, 4, 2, 3, 1)

Geht es effizienter?



aus Uwe Schöning: Algorithmik. Springer Verlag, 2001.

- Wenn eine optimale Rundreise bei Knoten 1 beginnt und dann den Knoten  $k$  besucht, dann muss der Weg von  $k$  aus durch die Städte  $\{2, \dots, n\} - \{k\}$  zurück nach Knoten 1 auch optimal sein.
- Sei  $g(i, S)$  die Länge des kürzesten Wegs, der bei Knoten  $i$  beginnt, dann durch jeden Knoten aus  $S$  genau einmal führt, und dann bei Knoten 1 endet.
- Berechne  $g(1, \{2, \dots, n\})$  mittels dynamischer Programmierung und folgender Formel:

$$g(i, S) = \begin{cases} c(i, 1) & \text{falls } S = \emptyset \\ \min_{j \in S} \left( c(i, j) + g(j, S - \{j\}) \right) & \text{sonst} \end{cases}$$

- Laufzeit:  $\mathcal{O}(n^2 \cdot 2^n) \leftarrow$  Tabellengröße mal Aufwand pro Eintrag

## *Entwurfsmethoden*

- divide and conquer
- dynamic programming
- *greedy*
- local search

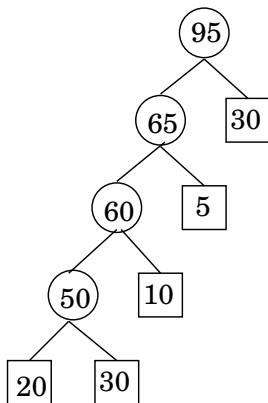
Aus Ellis Horowitz und Sartaj Sahni: Algorithmen – Entwurf und Analyse. Springer Verlag.

- Two sorted files containing  $q_1$  and  $q_2$  records respectively could be merged together to obtain one sorted file in time  $\mathcal{O}(q_1 + q_2)$ .
- When more than two sorted files are to be merged together the merge can be accomplished by repeatedly merging sorted files in pairs.
- Different pairings require different amounts of computing time.
- *Greedy*: At each step merge the two smallest sized files together.

# Optimal Merge Patterns

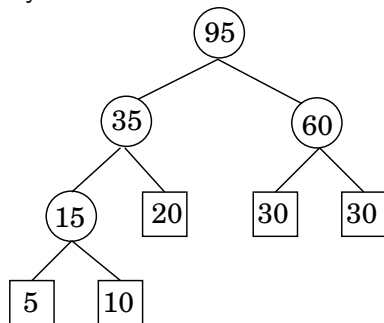
*Example:*  $(F_1, F_2, F_3, F_4, F_5) = (20, 30, 10, 5, 30)$

given order:



cost:  $50 + 60 + 65 + 95 = 270$

greedy order:



cost:  $15 + 35 + 60 + 95 = 205$

→ Correctness?

Korrektheit für  $n$  Dateien:

- Sei  $d_i$  die Entfernung von der Wurzel zum externen Knoten  $F_i$ , d.h. die Sätze von  $F_i$  werden  $d_i$ -mal verschoben.
- Sei  $q_i$  die Anzahl der Sätze von  $F_i$ , also die Länge von  $F_i$ .
- Die gewichtete externe Pfadlänge ist definiert als:  $\sum_{i=1}^n d_i \cdot q_i$

Induktionsanfang:

- Für  $n = 1$  hat der Baum keine inneren Knoten. ✓
- Für  $n = 2$  hat der Baum nur einen inneren Knoten. ✓

Induktionsschluss  $n \rightsquigarrow n + 1$ :

- o.B.d.A. sei  $q_1 \leq q_2 \leq \dots \leq q_n \leq q_{n+1}$
- Im ersten Schritt von Greedy wird  $F_1$  und  $F_2$  gemischt und es entsteht  $F_{1/2}$  mit Länge  $q_1 + q_2$ .

- Sei  $T$  ein optimaler Mischbaum für  $F_1, F_2, \dots, F_{n+1}$  und sei  $v$  ein innerer Knoten mit maximaler Entfernung zur Wurzel.
- Falls  $F_1$  und  $F_2$  nicht Nachfolger von  $v$  in  $T$  sind, dann ersetze die Nachfolger  $F_i$  und  $F_j$  von  $v$  durch  $F_1$  und  $F_2$ , wodurch die externe Pfadlänge für  $T$  nicht vergrößert wird:

$$\begin{aligned} & q_i \cdot d_{\max} + q_j \cdot d_{\max} + \dots + q_1 \cdot d_1 + q_2 \cdot d_2 \\ & \geq q_1 \cdot d_{\max} + q_2 \cdot d_{\max} + \dots + q_i \cdot d_1 + q_j \cdot d_2 \end{aligned}$$

Denn:

$$\begin{aligned} & (q_i - q_1) \cdot d_{\max} + (q_j - q_2) \cdot d_{\max} \\ & \geq (q_i - q_1) \cdot d_1 + (q_j - q_2) \cdot d_2 \end{aligned}$$

- Dann kann der innere Knoten  $v$  durch einen externen Knoten  $F_{1/2}$  mit Länge  $q_1 + q_2$  ersetzt werden. Der so entstandene Baum ist ein optimaler Mischbaum für  $F_{1/2}, F_3, F_4, \dots, F_{n+1}$ .
- Nach Induktionsvoraussetzung liefert Greedy dafür einen optimalen Mischbaum.

# Fractional Knapsack Problem

- We are given  $n$  objects and a knapsack.
- Object  $i$  has weight  $w_i$ , and profit  $p_i$ .
- The knapsack has a capacity of  $M$ .
- If a fraction  $x_i$  of object  $i$  is placed into the knapsack then a profit of  $p_i x_i$  is earned.
- The objective is to obtain a filling of the knapsack that maximises the total profit earned.
- *Greedy*: Include next the object which has the maximum profit per unit of capacity used.

# Fractional Knapsack Problem

*Example:*

$$M = 20, (p_1, p_2, p_3) = (25, 24, 15), (w_1, w_2, w_3) = (18, 15, 10)$$

*Greedy:* Include next the object with

- ① largest profit.  $\rightarrow$  not optimal
- ② lowest capacity.  $\rightarrow$  not optimal
- ③ maximum profit per unit of capacity used.  $\rightarrow$  Correctness?

Algo	$x_1$	$x_2$	$x_3$	$\sum w_i x_i$	$\sum p_i x_i$
1	1	$\frac{2}{15}$	0	20	28,2
2	0	$\frac{2}{3}$	1	20	31
3	0	1	$\frac{1}{2}$	20	31,5



Korrektheit für  $n$  Objekte: Es gelte  $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \dots \geq \frac{p_n}{w_n}$ .

- Wir werden zeigen, dass eine optimale Lösung die folgende Form hat:

$$(x_1, \dots, x_n) := (\underbrace{1, 1, \dots, 1}_k, b, \underbrace{0, 0, \dots, 0}_{n-k-1})$$

Dabei hat  $b$  den Wert  $(M - \sum_{i=1}^k w_i)/w_{k+1}$  und die Lösung erreicht den Wert  $\sum_{i=1}^k p_i + b \cdot p_{k+1}$ .

- Reduzieren wir eine der ersten  $k$  Einsen, dann können wir nur mit einem Objekt  $j > k$  den Rucksack auffüllen, also  $x_j$  erhöhen.
- Die neue Lösung wird um  $(1 - \alpha) \cdot p_i$  reduziert und um  $\beta \cdot p_j$  erhöht, wobei  $\beta \cdot w_j = (1 - \alpha) \cdot w_i$ , also  $\beta = (1 - \alpha) \cdot w_i/w_j$  gilt.

Fortsetzung:

- Wir hatten auf der letzten Seite festgestellt: Die neue Lösung wird um  $(1 - \alpha) \cdot p_i$  reduziert und um  $\beta \cdot p_j$  erhöht, wobei  $\beta \cdot w_j = (1 - \alpha) \cdot w_i$ , also  $\beta = (1 - \alpha) \cdot w_i / w_j$  gilt.
  - Der Wert der neuen Lösung unterscheidet sich also vom Wert der alten Lösung um  $(1 - \alpha) \cdot w_i p_j / w_j - (1 - \alpha) \cdot p_i$ .
  - Dieser Betrag ist kleiner gleich Null, da  $p_i / w_i \geq p_j / w_j$  gilt.
- Die neue Lösung ergibt keine Verbesserung.

analog zeigt man: Man erhält keine Verbesserung, wenn der Wert  $x_{k+1} = b$  reduziert und stattdessen ein Wert  $x_j$  mit  $j > k + 1$  erhöht wird.

# Coin Changing

The problem is to make change for  $n$  cents using the least number of coins. Is also called Change Making Problem CMP.

*Example:* Consider  $n = 88c$  and coins  $25c, 10c, 5c, 1c$ .

$i$	$c_i$	$n_i$	$n_i \text{ div } c_i$	$n_{i+1} = n_i \text{ mod } c_i$
1	25	88	3	13
2	10	13	1	3
3	5	3	0	3
4	1	3	3	0

$$\rightarrow 88c = 3 \cdot 25c + 1 \cdot 10c + 0 \cdot 5c + 3 \cdot 1c$$

*Counter-Example:* Consider  $n = 14c$  and coins  $11c, 7c, 1c$ .

- greedy yields  $n = 1 \cdot 11c + 3 \cdot 1c \rightarrow 4$  coins
- optimal is  $n = 2 \cdot 7c \rightarrow 2$  coins

*Counter-Example:* Consider  $n = 34c$  and coins  $25c, 10c, 1c$ .

- greedy yields  $n = 1 \cdot 25c + 9 \cdot 1c \rightarrow 10$  coins
- optimal is  $n = 3 \cdot 10c + 4 \cdot 1c \rightarrow 7$  coins

*Exercise:*

- Show that the greedy algorithm always yields an optimal solution for the coins  $25c, 10c, 5c, 1c$ .
- Suppose that the available coins are in the denominations  $c^0, c^1, c^2, \dots, c^k$  for some integers  $c > 1$  and  $k \geq 1$ . Show that the greedy algorithm always yields an optimal solution.
- Develop a solution of the coin changing problem by using dynamic programming.

optimal substructure:

- Sei  $(a_1, \dots, a_m)$  eine optimale Lösung für den Wert  $n$ , wobei  $\sum_{i=1}^m d(a_i) = n$  gilt, also alle Münzen zusammen den Wert  $n$  ergeben, und die Münzen aus dem gegebenen Wertebereich sind.
- Wenn wir die Münze  $a_1$  wegnehmen, dann ist  $(a_2, \dots, a_m)$  eine optimale Lösung für den Wert  $n - d(a_1)$ .

Andernfalls gäbe es eine Lösung  $(b_1, \dots, b_k)$  mit  $k < m - 1$  für den Wert  $n - d(a_1)$ . Dann wäre aber  $(a_1, b_1, b_2, \dots, b_k)$  eine bessere Lösung für den Wert  $n$ .  $\downarrow$

The greedy algorithm always yields an optimal solution for the coins  $c_1 = 1c$ ,  $c_2 = 5c$ ,  $c_3 = 10c$ ,  $c_4 = 25c$ .

Induktion über den Wert  $n$ :

- Greedy wählt die Münze  $c_k$ , falls  $c_k \leq n < c_{k+1}$  gilt.
- Jeder optimale Algorithmus wählt ebenfalls die Münze  $c_k$ :
  - Sonst müsste der optimale Algorithmus den Wert  $c_k$  durch eine Kombination von Münzen vom Typ  $c_1, \dots, c_{k-1}$  ersetzen.
  - Eine optimale Lösung enthält höchstens
    - vier 1c-Münzen, denn  $5 \cdot 1c = 1 \cdot 5c$ .
    - eine 5c-Münze, denn  $2 \cdot 5c = 1 \cdot 10c$ .
    - zwei 10c-Münzen, denn  $3 \cdot 10c = 1 \cdot 25c + 1 \cdot 5c$ .
    - eine 5c- und eine 10c-Münze, denn  $1 \cdot 5c + 2 \cdot 10c = 1 \cdot 25c$ .

Daher kann die Münze  $c_k$  nicht durch Münzen  $c_1, \dots, c_{k-1}$  ersetzt werden.

- Das Problem reduziert sich also auf das Wechselgeldproblem für den Wert  $n - c_k$ , was aufgrund der I.V. korrekt vom Greedy-Algorithmus gelöst wird.

Greedy yields an optimal solution if the available coins are in the denominations  $c^0, c^1, \dots, c^k$  for some integers  $c > 1$  and  $k \geq 1$ .

Induktion über den Wert  $n$ :

- Greedy wählt die Münze  $c^k$ , falls  $c^k \leq n < c^{k+1}$  gilt.
- Jeder optimale Algorithmus wählt ebenfalls die Münze  $c^k$ :
  - Sonst müsste der optimale Algorithmus den Wert  $c^k$  durch eine Kombination von Münzen vom Typ  $c^0, \dots, c^{k-1}$  ersetzen.
  - Jede optimale Lösung enthält maximal  $c - 1$  viele Münzen eines Typs, denn  $c \cdot c^i = 1 \cdot c^{i+1}$ . Außerdem gilt:

$$\sum_{i=0}^{k-1} c^i = c^0 + c^1 + \dots + c^{k-1} = \frac{c^k - 1}{c - 1}$$

Also gilt:  $(c - 1) \cdot (c^0 + c^1 + \dots + c^{k-1}) = c^k - 1 < c^k$

- Das Problem reduziert sich also auf das Wechselgeldproblem für den Wert  $n - c^k$ , was aufgrund der I.V. korrekt vom Greedy-Algorithmus gelöst wird.

*dynamic programming* Let  $C(p)$  be the minimum number of coins of denominations  $d_1, \dots, d_k$  needed to make change for  $p$  cents.

$$C(p) = \begin{cases} 0 & \text{if } p = 0 \\ \min_{i: d_i \leq p} \{1 + C(p - d_i)\} & \text{if } p > 0 \end{cases}$$

$C[0] := 0$

**for**  $p := 1$  to  $n$  **do**

$min := \infty$

**for**  $i := 1$  to  $k$  **do**

**if**  $p - d[i] \geq 0$  **and**  $1 + C[p - d[i]] < min$  **then**

$min := 1 + C[p - d[i]]$

$C[p] := min$



## Remarks:

- Change Making Problem is NP-complete! It is a variant of the knapsack problem, where each item can be taken multiple times. [Lueker, 1975]
- Coin systems, for which the greedy algorithm always gives an optimal solution, are called canonical.
- Goebbels, Gurski, Rethmann, Yilmaz. Change-Making Problems revisited: A Parameterized Point of View. Journal of Combinatorial Optimization, 34(4): 1218-1236, 2017.

*Gegeben:*

- Alphabet  $\Sigma = \{a_1, a_2, \dots, a_n\}$
- Wahrscheinlichkeit  $p_i$  für das Zeichen  $a_i$

*Gesucht:*

- Optimaler Präfixcode für die W-Verteilung.

*Präfixcode:* Kein Codewort  $c(a_i)$  ist Anfangsstück (Präfix) eines anderen Codeworts  $c(a_j)$ .

*Ziel:* Minimiere die mittlere Codewortlänge.

$$\sum_{i=1}^n p_i \cdot |c(a_i)|$$

*Beispiel:* Sei  $\Sigma = \{a, b, c, d, e, f\}$  und

$$\begin{array}{lll} p(a) = 0,45 & p(b) = 0,13 & p(c) = 0,12 \\ p(d) = 0,16 & p(e) = 0,09 & p(f) = 0,05 \end{array}$$

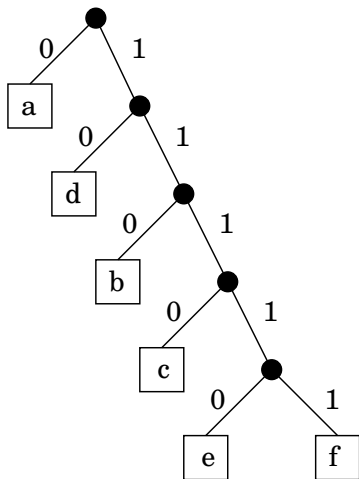
*Präfixcode 1:*  $a \mapsto 0$        $b \mapsto 110$        $c \mapsto 1110$   
 $d \mapsto 10$        $e \mapsto 11110$        $f \mapsto 11111$

0 11110 10 0 0 11111 10 0 1110 110 = a e d a a f d a c b

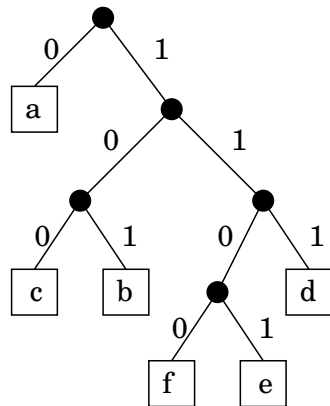
*Präfixcode 2:*  $a \mapsto 0$        $b \mapsto 101$        $c \mapsto 100$   
 $d \mapsto 111$        $e \mapsto 1101$        $f \mapsto 1100$

0 1101 111 0 0 1100 111 0 100 101 = a e d a a f d a c b

Jeder Code lässt sich in Form eines Codebaums darstellen:



Mittlere Codewortlänge: 2,34

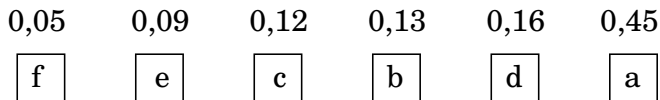


Mittlere Codewortlänge: 2,24

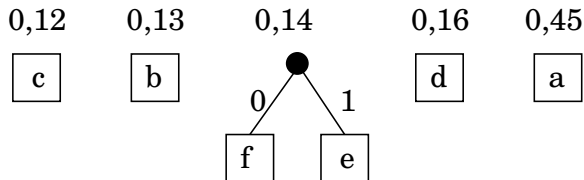
# Präfix-Code nach Huffman

*Greedy:* Fasse jeweils zwei Bäume zusammen, indem man diejenigen mit den kleinsten Werten auswählt und eine gemeinsame Wurzel erstellt.

*Initial:* Jedes Symbol ist ein eigener Baum.

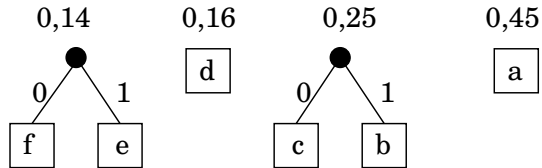


1. Schritt:

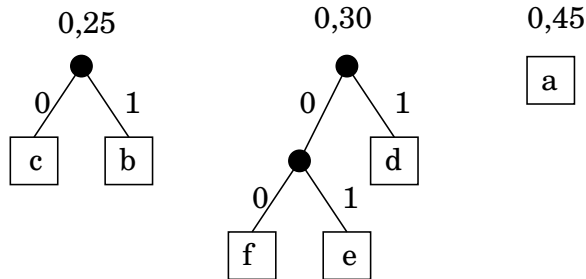


# Präfix-Code nach Huffman

## 2. Schritt:



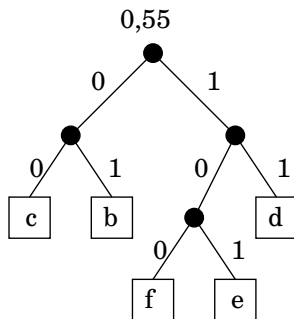
## 3. Schritt:



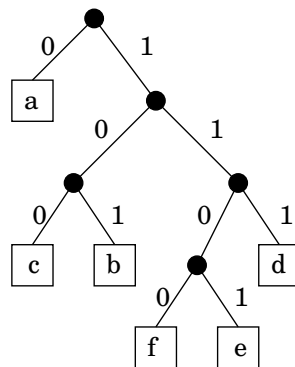
4. Schritt:

0,45

a



5. Schritt:



Korrektheit? → Uwe Schöning: Algorithmen – kurz gefasst.

## *Entwurfsmethoden*

- divide and conquer
- dynamic programming
- greedy
- *local search*



## *Idee:*

- Beginne mit einer beliebigen Lösung.
- Mutiere die Lösung durch geringfügige, lokale Veränderungen.
- Ist die neue Lösung besser, übernehme die neue Lösung.
- Das Verfahren wird auch *hill climbing* oder *lokale Verbesserungsstrategie* genannt.

*Problem:* Eventuell wird nur ein lokales Optimum gefunden.

## *Lösung:*

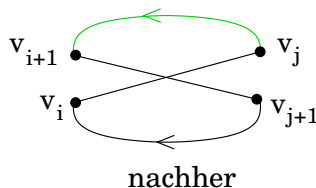
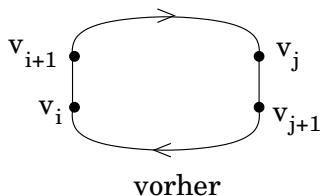
- Wiederholen mit unterschiedlichen Startlösungen.
- Wähle lokale Veränderungen zufällig aus.
- Akzeptiere mit geringer Wahrscheinlichkeit auch schlechtere Lösungen.

# Traveling Salesperson Problem

- Initial: Wähle zufällige Permutation der Knoten  $v_1, v_2, \dots, v_n$ .
- Wähle zwei zufällige Knoten  $v_i$  und  $v_j$ , die auf der bisherigen Rundreise nicht aufeinander folgen, dabei sei  $i < j$ .
- Falls die neue Rundreise

$v_1, v_2, \dots, v_i, v_j, v_{j-1}, \dots, v_{i+2}, v_{i+1}, v_{j+1}, v_{j+2}, \dots, v_n$

kürzer ist, wiederhole das Verfahren mit der geänderten Rundreise.



- Nur bei symmetrischer Entfernungsmatrix sinnvoll.

## Übung

- (1) Schreiben Sie eine Funktion, die eine zufällige Permutation von  $n$  Zahlen berechnet.
- (2) Schreiben Sie eine Funktion, die alle Permutationen von  $n$  Zahlen aufzählt.

# Lösung Übung (1)

```
#include <vector>
#include <algorithm>
using namespace std;

vector<int> getRndPerm(int n) {
    vector<int> p, res;
    int cnt = n;

    for (int i = 0; i < n; i++)
        p.push_back(i);

    for (int i = 0; i < n; i++) {
        int pos = rand() % cnt;
        res.push_back(p[pos]);
        p[pos] = p[cnt - 1];
        cnt -= 1;
    }
    return res;
}
```

## Lösung Übung (2)

```
void perm(list<vector<int>>& res, vector<int>& v, int n) {  
    if (n == 0)  
        res.push_back(v);  
    else for (int i = 0; i < n; i++) {  
        swap(v[i], v[n-1]);  
        perm(res, v, n - 1);  
        swap(v[i], v[n-1]);  
    }  
}
```

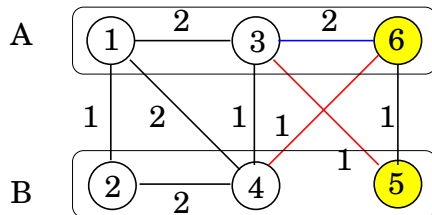
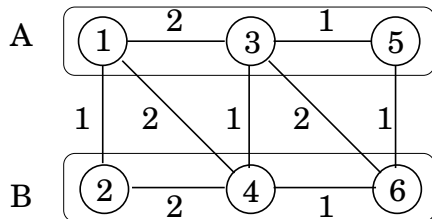
```
list<vector<int>> getAllPerms(int n) {  
    list<vector<int>> res;  
    vector<int> base;  
  
    for (int i = 0; i < n; i++)  
        base.push_back(i);  
    perm(res, base, n);  
    return res;  
}
```

# Uniform Graph Partitioning

- Given a symmetric cost matrix  $[d_{ij}]$  defined on the edges of an undirected graph  $G = (V, E)$  with  $|V| = 2n$ .
- A partition  $V = A \uplus B$  such that  $|A| = |B| = n$  is called a uniform partition.
- Find a uniform partition such that the cost

$$c(A, B) = \sum_{i \in A, j \in B} d_{ij}$$

is minimum over all uniform partitions.



*Observation:* Given a uniform partition  $A, B$ .

- Suppose  $A^*, B^*$  is an optimal uniform partition.
- Let  $X$  be those elements of  $A$  that are not in  $A^*$  and let  $Y$  be similarly defined for  $B$ .
- Then  $|X| = |Y|$  and  $A^* = (A \setminus X) \cup Y$  and  $B^* = (B \setminus Y) \cup X$ .

For two elements  $a \in A, b \in B$ , the operating of forming  $A' := (A \setminus \{a\}) \cup \{b\}$  and  $B' := (B \setminus \{b\}) \cup \{a\}$  is called a swap.

*Vertex Cover:* Given a graph  $G = (V, E)$ , find a subset of nodes  $S \subseteq V$  of minimal cardinality such that for each edge  $\{u, v\} \in E$ , either  $u$  or  $v$  (or both) are in  $S$ .

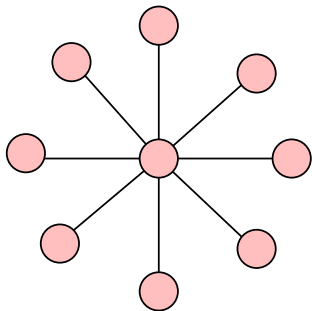
*Neighbor relation:*  $S - S'$  if  $S'$  can be obtained from  $S$  by adding or deleting a single node. Each vertex cover  $S$  has at most  $n$  neighbors.

*Gradient descent:* Start with  $S = V$ . If there is a neighbor  $S'$  that is a vertex cover and has lower cardinality, replace  $S$  with  $S'$ .

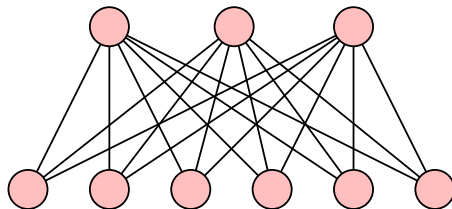
*Remark:* Algorithm terminates after at most  $n$  steps since each update decreases the size of the cover by one.



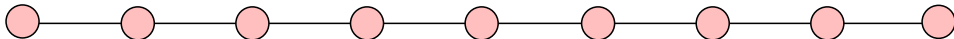
# Vertex Cover



optimum: center node only  
local optimum: all other nodes



optimum: all nodes at top  
local optimum: all nodes at bottom



optimum: even nodes  
local optimum: omit every third node

- Einleitung
- Entwurfsmethoden
- *Sortieren*
- Auswahlproblem
- Graphalgorithmen
- Spezielle Graphklassen
- Vorrangwarteschlangen
- Suchbäume
- Amortisierte Laufzeitanalyse
- Algorithmen für moderne Hardware
- Algorithmen für geometrische Probleme
- Randomisierte Algorithmen

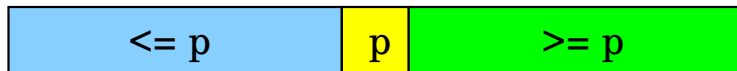
## Sortieren

- *Quick-Sort – unterschiedliche Varianten*
- Heap-Sort
- Untere Schranke
- Counting-/Radix-Sort
- Bucket-Sort

## Einige Fakten:

- Quicksort wurde 1962 von C.A.R. Hoare veröffentlicht.
- Es ist ein Divide-and-Conquer-Algorithmus.
- Quicksort ist eines der schnellsten allgemeinen Sortierverfahren.
- in-situ: Es ist kein zusätzlicher Speicherplatz zur Speicherung von Datensätzen erforderlich, außer einer konstanten Anzahl von Hilfsspeicherplätzen für Tauschoperationen.
- Praxistauglich!

- 1 *Divide*: Wähle aus allen Werten einen beliebigen Wert  $p$  aus und teile die Folge in zwei Teilfolgen  $K$  und  $G$  auf:
  - $K$  enthält Werte die kleiner oder gleich  $p$  sind,
  - $G$  enthält Werte die größer oder gleich  $p$  sind.



- 2 *Conquer*: Sortiere  $K$  und  $G$  rekursiv
- 3 *Combine*: Trivial, entfällt.

Noch offene Punkte:

- Wie wird die Folge in zwei Teilfolgen aufgeteilt?
- Wie soll das Pivot-Element  $p$  gewählt werden?

*Pivot-Element:* Erstes Element der Teilfolge.

*Aufteilen der Folge in zwei Teilfolgen:*

```
partition(int  $\ell$ , int  $r$ )  
   $p := A[\ell]$ ,  $i := \ell + 1$ ,  $j := r$   
  repeat  
    while ( $i < r$ ) and ( $A[i] \leq p$ ) do  $i := i + 1$   
    while ( $j > \ell$ ) and ( $A[j] \geq p$ ) do  $j := j - 1$   
    if  $i < j$  then swapAt( $i, j$ )  
  until  $j \leq i$   
  swapAt( $\ell, j$ )  
  return  $j$ 
```

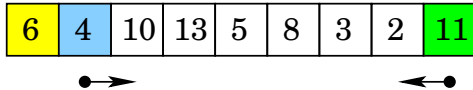
Hinweis: **repeat A until B** ist äquivalent zu **do A while  $\neg B$**

→ Die Laufzeit ist in  $\Theta(n)$  für  $n$  Elemente.

Beispiel:

6	4	10	13	5	8	3	2	11
i								j

Beispiel:





Beispiel:

6	4	10	13	5	8	3	2	11
---	---	----	----	---	---	---	---	----

6	4	2	13	5	8	3	10	11
---	---	---	----	---	---	---	----	----

i

j

Beispiel:

6	4	10	13	5	8	3	2	11
---	---	----	----	---	---	---	---	----

6	4	2	13	5	8	3	10	11
---	---	---	----	---	---	---	----	----



Beispiel:

6	4	10	13	5	8	3	2	11
---	---	----	----	---	---	---	---	----

6	4	2	13	5	8	3	10	11
---	---	---	----	---	---	---	----	----

6	4	2	3	5	8	13	10	11
---	---	---	---	---	---	----	----	----

i

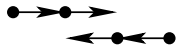
j

Beispiel:

6	4	10	13	5	8	3	2	11
---	---	----	----	---	---	---	---	----

6	4	2	13	5	8	3	10	11
---	---	---	----	---	---	---	----	----

6	4	2	3	5	8	13	10	11
---	---	---	---	---	---	----	----	----



Beispiel:

6	4	10	13	5	8	3	2	11
---	---	----	----	---	---	---	---	----

6	4	2	13	5	8	3	10	11
---	---	---	----	---	---	---	----	----

6	4	2	3	5	8	13	10	11
---	---	---	---	---	---	----	----	----

j   i

Beispiel:

6	4	10	13	5	8	3	2	11
---	---	----	----	---	---	---	---	----

6	4	2	13	5	8	3	10	11
---	---	---	----	---	---	---	----	----

6	4	2	3	5	8	13	10	11
---	---	---	---	---	---	----	----	----

5	4	2	3	6	8	13	10	11
---	---	---	---	---	---	----	----	----

*Pseudo-Code:*

```
quicksort(int  $\ell$ , int  $r$ )  
  if  $\ell < r$   
     $m := \text{partition}(\ell, r)$   
    quicksort( $\ell, m - 1$ )  
    quicksort( $m + 1, r$ )
```

*Initialer Aufruf:* quicksort(0, n-1)

## *Worst-case Analyse:*

Betrachte sortierte Folge: Bei jedem rekursiven Aufruf ist die Teilfolge  $K$  leer und Teilfolge  $G$  wird um ein Element, dem Pivot-Element, kürzer.

$$\begin{aligned}T(n) &= T(n-1) + \Theta(n) \\&= T(n-2) + \Theta(n-1) + \Theta(n) \\&= T(n-3) + \Theta(n-2) + \Theta(n-1) + \Theta(n) \\&\vdots \\&= T(1) + \Theta(2) + \dots + \Theta(n-2) + \Theta(n-1) + \Theta(n) \\&= \Theta\left(\sum_{k=1}^n k\right) = \Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^2)\end{aligned}$$



## *Best-case Analyse:*

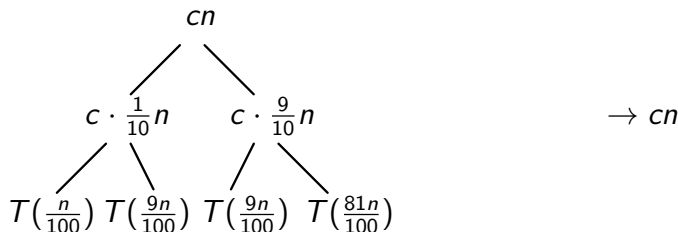
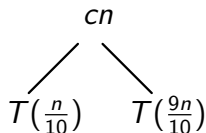
- Die Folge wird bei jeder Aufteilung halbiert.
- $T(n) = 2 \cdot T(n/2) + \Theta(n)$
- Das Master-Theorem liefert für  $a = 2$ ,  $b = 2$  und  $k = 1$ :

$$T(n) = \Theta(n^k \cdot \log(n)) = \Theta(n \cdot \log(n))$$

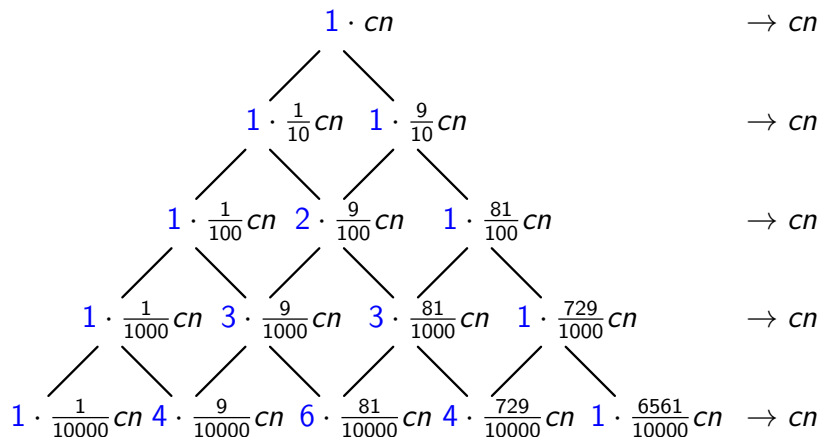
## *Übung:*

- Welche Laufzeit hat Quicksort, wenn die Aufteilung immer in einem festen Verhältnis erfolgt?
- Welche Laufzeit hat Quicksort im mittleren Fall?

Wenn die Aufteilung im Verhältnis  $\frac{1}{10}$  zu  $\frac{9}{10}$  erfolgt, erhalten wir folgenden Rekursionsbaum:



Wir setzen die Aufteilung fort, wobei wir gleich große Teile zusammen fassen:



Auf jeder vollständigen Ebene  $k$  des Rekursionsbaumes ergibt sich eine Laufzeit von  $cn$ , denn:

$$\begin{aligned}\sum_{i=0}^k \binom{k}{i} \frac{9^i}{10^k} cn &= \frac{cn}{10^k} \sum_{i=0}^k \binom{k}{i} 9^i \\ &= \frac{cn}{10^k} \sum_{i=0}^k \binom{k}{i} 9^i \cdot 1^{k-i} \\ &= \frac{cn}{10^k} (9 + 1)^k = cn\end{aligned}$$

Wir erhalten einen Baum, der rechts wesentlich tiefer ist als links. Das Rekursionsende im rechten Zweig ist erreicht bei:

$$\left(\frac{9}{10}\right)^k \cdot n = 1 \iff n = \left(\frac{10}{9}\right)^k \iff \log_{10/9}(n) = k$$

Der Baum hat also nur eine logarithmische Tiefe.

Die Wahrscheinlichkeit, dass die Folge an Position  $p$  aufgeteilt wird, ist  $1/n$ . Das Aufteilen in zwei Teilfolgen erfolgt in Zeit  $\mathcal{O}(n)$ .

$$T(n) = c \cdot n + \frac{1}{n} \cdot \sum_{p=1}^n (T(p-1) + T(n-p))$$

Es gilt:

$$T(0) + \dots + T(n-1) = T(n-1) + \dots + T(0)$$

Damit erhalten wir:

$$T(n) = c \cdot n + \frac{2}{n} \cdot \sum_{p=1}^n T(p-1)$$

Multiplikation mit  $n$  ergibt:

$$n \cdot T(n) = c \cdot n^2 + 2 \cdot \sum_{p=1}^n T(p-1)$$

Wir hatten auf der letzten Folie festgestellt:

$$n \cdot T(n) = c \cdot n^2 + 2 \cdot \sum_{p=1}^n T(p-1)$$

Subtraktion der gleichen Formel für  $n-1$  ergibt:

$$\begin{aligned} n \cdot T(n) - (n-1) \cdot T(n-1) \\ = cn^2 + 2 \cdot \sum_{p=1}^n T(p-1) - c(n-1)^2 - 2 \cdot \sum_{p=1}^{n-1} T(p-1) \end{aligned}$$

Die Terme der beiden Summen heben sich gegenseitig auf, bis auf den Term  $2 \cdot T(n-1)$ . Außerdem gilt  $(n-1)^2 = n^2 - 2n + 1$ , daher erhalten wir:

$$\begin{aligned} n \cdot T(n) - (n-1) \cdot T(n-1) &= 2 \cdot T(n-1) + c \cdot (2n-1) \\ n \cdot T(n) &\leq (n+1) \cdot T(n-1) + 2cn \end{aligned}$$

Wir hatten bereits festgestellt:

$$n \cdot T(n) \leq (n+1) \cdot T(n-1) + 2cn$$

Division durch  $n \cdot (n+1)$  liefert:

$$\frac{T(n)}{n+1} \leq \frac{T(n-1)}{n} + \frac{2c}{n+1}$$

Dies lässt sich fortsetzen:

$$\begin{aligned} \frac{T(n)}{n+1} &\leq \frac{T(n-2)}{n-1} + \frac{2c}{n} + \frac{2c}{n+1} \\ &\leq \frac{T(n-3)}{n-2} + \frac{2c}{n-1} + \frac{2c}{n} + \frac{2c}{n+1} \\ &\leq \frac{T(1)}{2} + \sum_{p=2}^n \frac{2c}{p+1} \end{aligned}$$

Wir formen das Ergebnis der letzten Folie ein wenig um:

$$\begin{aligned}\frac{T(n)}{n+1} &\leq \frac{T(1)}{2} + \sum_{p=2}^n \frac{2c}{p+1} = \frac{T(1)}{2} + 2c \cdot \sum_{p=3}^{n+1} \frac{1}{p} \\ &= \frac{T(1)}{2} + 2c \cdot \sum_{p=2}^n \frac{1}{p} + 2c \frac{1}{n+1} - c\end{aligned}$$

Wir wissen, dass für die harmonische Reihe gilt:

$$\sum_{p=2}^n \frac{1}{p} \leq \ln(n)$$

Also erhalten wir:

$$\frac{T(n)}{n+1} \leq \frac{T(1)}{2} + 2c \cdot \ln(n) + 2c \frac{1}{n+1} - c$$



Wir hatten bereits festgestellt:

$$\frac{T(n)}{n+1} \leq \frac{T(1)}{2} + 2c \cdot \ln(n) + 2c \frac{1}{n+1} - c$$

Multiplikation mit  $n+1$  liefert:

$$T(n) \leq (n+1) \cdot \frac{T(1)}{2} + 2c(n+1) \cdot \ln(n) + 2c - c(n+1)$$

Lassen wir Konstanten und Terme niedriger Ordnung weg, so erhalten wir schließlich:

$$T(n) \in \Theta(n \cdot \log(n))$$

*Fazit:* Laufzeit von Quicksort im

- worst-case:  $T(n) \in \Theta(n^2)$
- average-case:  $T(n) \in \Theta(n \cdot \log(n))$
- best-case:  $T(n) \in \Theta(n \cdot \log(n))$

*Problem:*

- Laufzeit ist in  $\mathcal{O}(n^2)$  bei stark vorsortierten Folgen.

*Lösung:*

- **Zufallsstrategie:** Wähle als Pivot-Element ein zufälliges Element aus  $A[l \dots r]$  und vertausche es mit  $A[l]$ .
- ⇒ Laufzeit ist unabhängig von der zu sortierenden Folge.
- ⇒ Mittlere/erwartete Laufzeit:  $\Theta(n \cdot \log(n))$

- Implementieren Sie Quicksort und vergleichen Sie Ihre Implementierung mit der qsort-Implementierung aus der Standardbibliothek.

Vergleichen Sie dazu die Laufzeiten für zufällige Zahlenfolgen der Länge  $2^{15}, 2^{16}, 2^{17}, \dots, 2^{25}$  als Eingabe.

- Ändert sich das Laufzeitverhalten, wenn die Zufallszahlen nur aus dem Bereich von 0 bis 99.999 gewählt werden.

Man nennt ein Sortierverfahren *glatt* (smooth), wenn es im Mittel  $N$  verschiedene Schlüssel in  $\mathcal{O}(N \cdot \log(N))$  und  $N$  gleiche Schlüssel in  $\mathcal{O}(N)$  Schritten zu sortieren vermag mit einem weichen Übergang zwischen diesen Werten.

## *3-Wege-Split Quicksort:*

Teile die Folge  $a[\ell], \dots, a[r]$  in drei Folgen  $F_\ell, F_m, F_r$  auf.

- ①  $F_\ell$  enthält die Elemente mit Schlüssel  $< k$ .
- ②  $F_m$  enthält die Elemente mit Schlüssel  $= k$ .
- ③  $F_r$  enthält die Elemente mit Schlüssel  $> k$ .

Sortiere  $F_\ell$  und  $F_r$  auf dieselbe Weise.

## 3-Wege-Split Quicksort

### Anmerkungen:

- keine zwei gleichen Schlüssel  $\rightarrow$  keine Ersparnis
- alle Schlüssel identisch  $\rightarrow$  kein rekursiver Aufruf

*Laufzeit:* Im Mittel werden  $\mathcal{O}(N \cdot \log(n) + N)$  Schritte benötigt, wobei  $n$  die Anzahl der verschiedenen Schlüssel unter den  $N$  Schlüsseln der Eingabefolge ist.

*Idee:* Die Pivotelemente werden zuerst am Rand zwischen 1 und 12 sowie zwischen r2 und r gesammelt und vor dem rekursiven Aufruf in die Mitte transportiert.

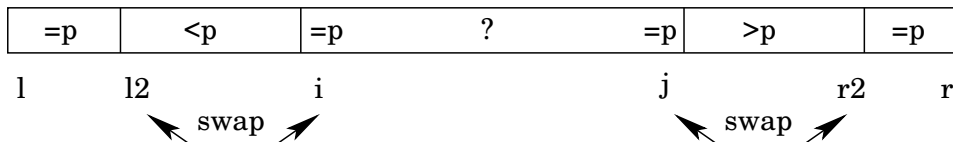
=p	<p	?	>p	=p	
l	l2	i	j	r2	r

### 3-Wege-Split Quicksort

Initial setzen wir  $l2 := 1$  und  $r2 := r$ , da noch keine Elemente gefunden wurden, die gleich dem Pivot-Element sind.

Während der Aufteilung, also der Partitionierung, sind drei Fälle zu unterscheiden. Der vierte Fall wird wie im ursprünglichen Quicksort behandelt:

*Fall 1:*  $a[i] == p$  und  $a[j] == p$

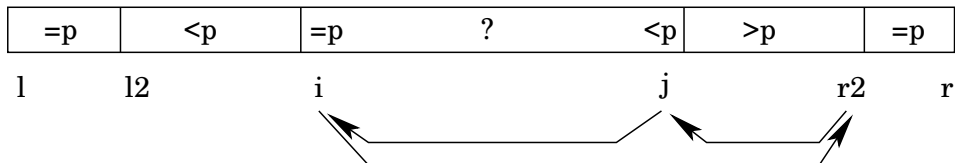


Nach dem Tauschen setze  $l2 := l2 + 1$  und  $r2 := r2 - 1$ , da jeweils am linken und rechten Rand Elemente positioniert wurden, die gleich dem Pivot-Element sind.

In allen vier Fällen setzen wir  $i := i + 1$  und  $j := j - 1$ .

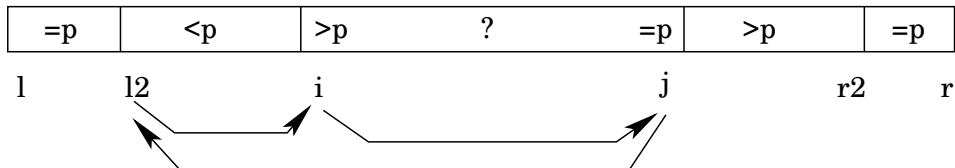
## 3-Wege-Split Quicksort

Fall 2:  $a[i] == p$  und  $a[j] < p$



$r2 := r2 - 1$

Fall 3:  $a[i] > p$  und  $a[j] == p$

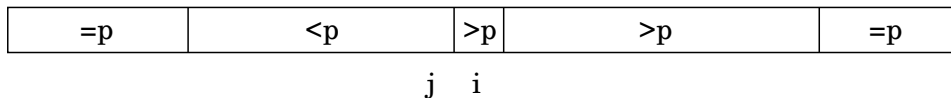


$l2 := l2 + 1$

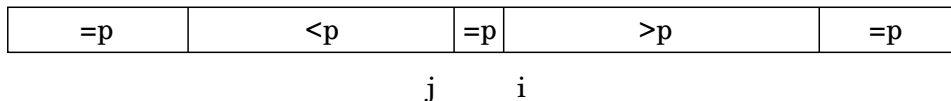
## 3-Wege-Split Quicksort

Unterscheide drei Fälle nach der Aufteilung, falls  $i = j$ :

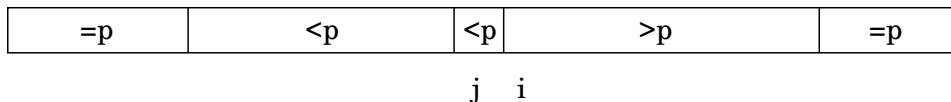
if  $a[i] > p$  then  $j := j - 1$



if  $a[i] = p$  then  $j := j - 1; i := i + 1$



if  $a[i] < p$  then  $i := i + 1$





Anschließend wird vom linken Rand nach  $j$  und vom rechten Rand nach  $i$  kopiert

```
for k := 1 to l2 do  
    swapAt(k, j);  
    j := j - 1;
```

```
for k := r downto r2 do  
    swapAt(k, i);  
    i := i + 1;
```

### *Übung:*

Implementieren Sie den 3-Wege-Split Quicksort und vergleichen Sie die Laufzeit mit der ursprünglichen Version.

*Problem:* Im ungünstigen Fall ist die Rekursionstiefe  $\mathcal{O}(n)$ .

Verringern der Rekursionstiefe auf  $\mathcal{O}(\log(n))$ :

- Löse das kleinere Teilproblem rekursiv und
- löse das größere Teilproblem iterativ direkt.

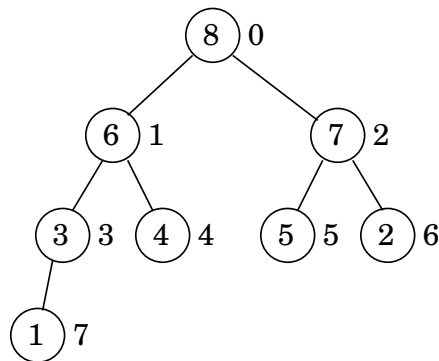
```
quicksort(int  $\ell$ , int  $r$ )  
  while  $\ell < r$   
     $m := \text{partition}(\ell, r)$   
    if  $(m - \ell) < (r - m)$  then  
      quicksort( $\ell, m - 1$ )  
       $\ell := m + 1$   
    else  
      quicksort( $m + 1, r$ )  
       $r := m - 1$ 
```

## *Sortieren*

- Quick-Sort – unterschiedliche Varianten
- *Heap-Sort*
- Untere Schranke
- Counting-/Radix-Sort
- Bucket-Sort

*Heap:* Eine Folge  $F = k_0, \dots, k_n$  von Schlüsseln, so dass  $k_i \geq k_{2i+1}$  und  $k_i \geq k_{2(i+1)}$  gilt, sofern  $2i + 1 \leq n$  bzw.  $2(i + 1) \leq n$ .

*Beispiel:*  $F = 8, 6, 7, 3, 4, 5, 2, 1$



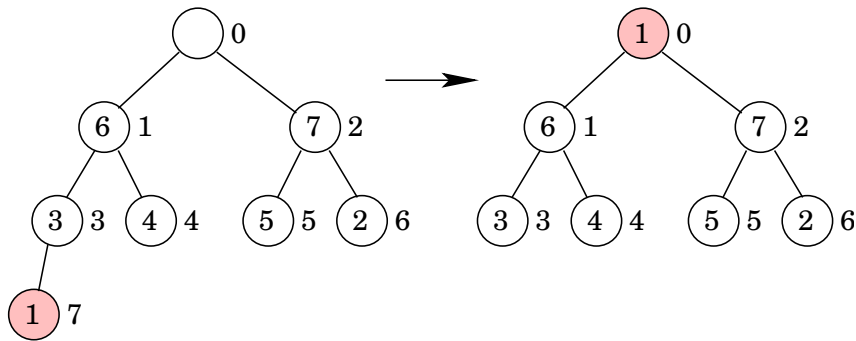
Heap-Eigenschaft ist erfüllt:

- $k_0 = 8 \geq k_1 = 6$  und  $k_0 \geq k_2 = 7$
- $k_1 = 6 \geq k_3 = 3$  und  $k_1 \geq k_4 = 4$
- $k_2 = 7 \geq k_5 = 5$  und  $k_2 \geq k_6 = 2$
- $k_3 = 3 \geq k_7 = 1$

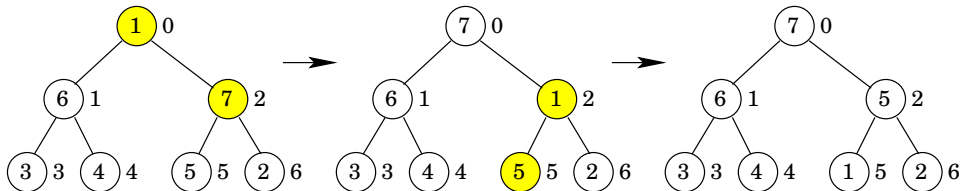
Bestimmung des Maximums ist leicht:  $k_0$  ist das Maximum.

Das nächst kleinere Element wird bestimmt, indem das Maximum aus  $F$  entfernt wird und die Restfolge wieder zu einem Heap transformiert wird:

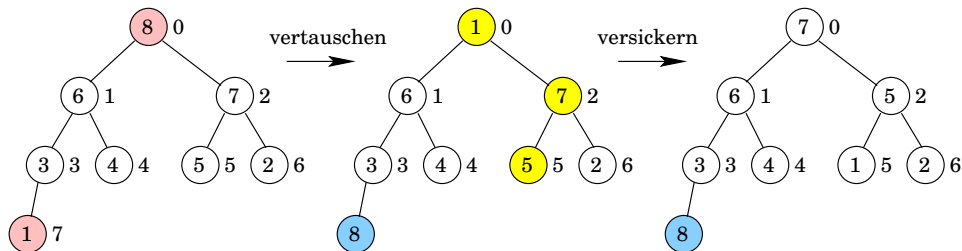
1. Setze den Schlüssel mit dem größten Index an die erste Position.  $\Rightarrow$   
**Heap-Eigenschaft verletzt!**



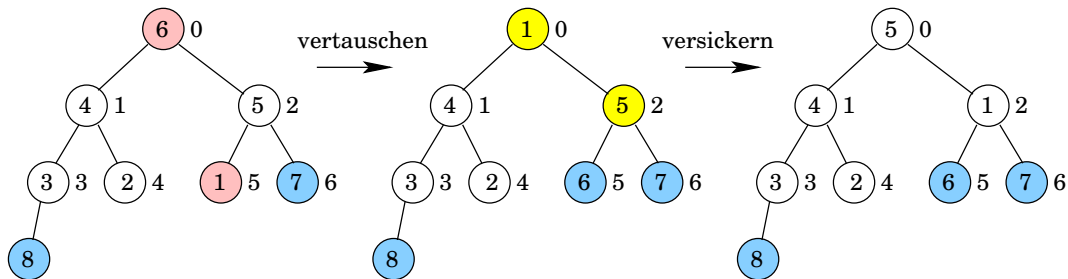
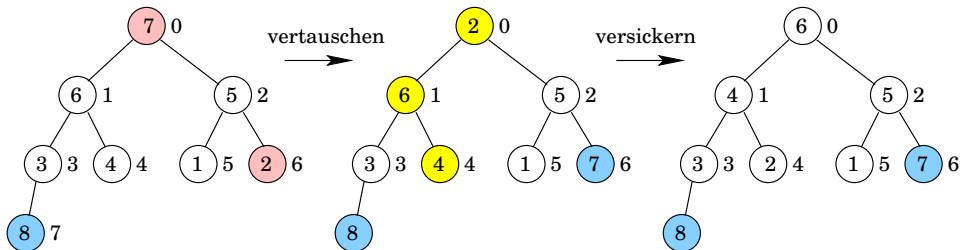
2. Schlüssel versickern lassen, indem er immer mit dem größten seiner Nachfolger getauscht wird, bis entweder beide Nachfolger kleiner sind oder der Schlüssel unten angekommen ist.



Die Datensätze können sortiert werden, indem das jeweils aus dem Heap entfernte Maximum an die Stelle desjenigen Schlüssels geschrieben wird, der nach dem Entfernen des Maximums nach  $k_0$  übertragen wird.



# Heapsort



USW.



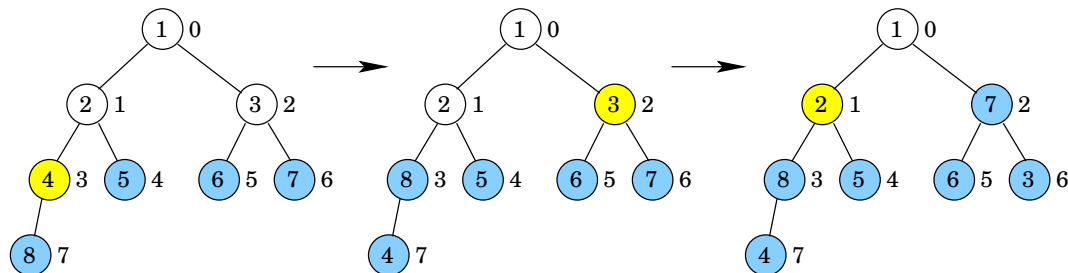
## *Analyse:*

- Es erfolgen  $n - 1$  Vertauschungen außerhalb der Funktion `versickern`.
  - Innerhalb von `versickern` wird ein Schlüssel wiederholt mit einem seiner Nachfolger vertauscht, wobei der Datensatz jeweils eine Stufe tiefer wandert.
  - Ein Heap mit  $n$  Datensätzen hat  $\lceil \log_2(n + 1) \rceil$  viele Ebenen.
- ⇒ Wir erhalten  $\mathcal{O}(n \cdot \log(n))$  als obere Schranke für die Anzahl der Vertauschungen.

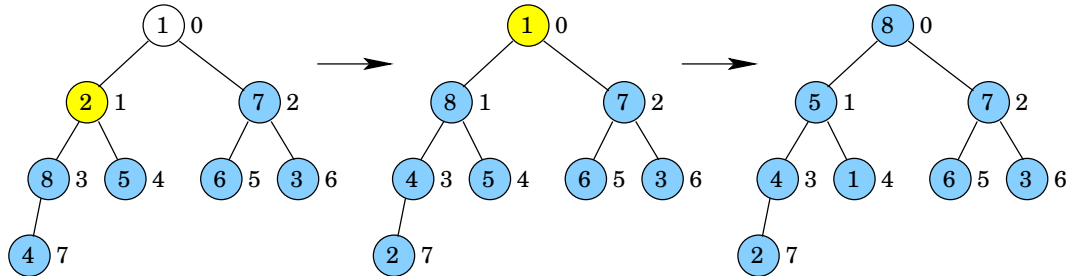
Wie wird die Anfangsfolge in einen Heap transformiert?

*Idee:* Lasse die Schlüssel  $k_{n/2-1}, \dots, k_0$  versickern. Dadurch werden schrittweise immer größere Heaps aufgebaut, bis letztlich ein Heap übrig bleibt.

*Beispiel:*  $F = 1, 2, 3, 4, 5, 6, 7, 8$



# Heapsort



*Analyse:* Aufbauen des initialen Heaps ist in linearer Zeit möglich, weil ...

Betrachten wir einen vollständigen Binärbaum mit  $\ell + 1$  Ebenen, also mit  $\approx 2^{\ell+1}$  vielen Knoten.

- Auf jeder Ebene  $i$ ,  $0 \leq i \leq \ell$ , befinden sich  $2^i$  viele Knoten.
- Wenn ein Element von Ebene  $i$  versickert wird, werden höchstens  $\ell - i$  viele Vertauschungen durchgeführt. Auf Ebene  $\ell$  werden keine Elemente versickert.
- Die Anzahl Vertauschungen insgesamt beträgt also:

$$\begin{aligned} & 2^0 \cdot \ell + 2^1 \cdot (\ell - 1) + 2^2 \cdot (\ell - 2) + \dots + 2^{\ell-1} \cdot 1 \\ &= 2^{\ell-1} \cdot 1 + 2^{\ell-2} \cdot 2 + 2^{\ell-3} \cdot 3 + \dots + 2^0 \cdot \ell \\ &= 2^{\ell} \cdot (1/2^1 + 2/2^2 + 3/2^3 + 4/2^4 + \dots + \ell/2^{\ell}) \end{aligned}$$

- Um  $1/2^1 + 2/2^2 + 3/2^3 + 4/2^4 + \dots + \ell/2^{\ell}$  zu berechnen, sei  $s_m := 1/2^1 + 2/2^2 + 3/2^3 + \dots + m/2^m$

$$\begin{aligned}s_1 &= \frac{1}{2} &= 2 - \frac{3}{2} \\s_2 &= s_1 + \frac{2}{2^2} = \frac{1}{2} + \frac{2}{4} = \frac{2}{4} + \frac{2}{4} = \frac{4}{4} &= 2 - \frac{4}{4} \\s_3 &= s_2 + \frac{3}{2^3} = \frac{4}{4} + \frac{3}{8} = \frac{8}{8} + \frac{3}{8} = \frac{11}{8} &= 2 - \frac{5}{8} \\s_4 &= s_3 + \frac{4}{2^4} = \frac{11}{8} + \frac{4}{16} = \frac{22}{16} + \frac{4}{16} = \frac{26}{16} &= 2 - \frac{6}{16} \\s_5 &= s_4 + \frac{5}{2^5} = \frac{26}{16} + \frac{5}{32} = \frac{52}{32} + \frac{5}{32} = \frac{57}{32} &= 2 - \frac{7}{32} \\s_6 &= s_5 + \frac{6}{2^6} = \frac{57}{32} + \frac{6}{64} = \frac{114}{64} + \frac{6}{64} = \frac{120}{64} &= 2 - \frac{8}{64}\end{aligned}$$

Vermutung  $s_m = 2 - \frac{m+2}{2^m}$  mittels vollständiger Induktion beweisen!

Für die Anzahl der Vertauschungen gilt also:

$$2^\ell \cdot (1/2^1 + 2/2^2 + 3/2^3 + \dots + \ell/2^\ell) < 2^\ell \cdot 2 = 2^{\ell+1} \in \mathcal{O}(n)$$

Damit ergibt sich insgesamt eine Laufzeit von  $\mathcal{O}(n \cdot \log(n))$ .

*Anmerkungen:*

- Eine Vorsortierung der Eingabefolge schadet und nützt der Sortierung nichts.
- Heapsort benötigt nur konstant viel zusätzlichen Speicherplatz (in-situ Sortierv Verfahren).

*Übung:*

Implementieren Sie Quick-Sort, Merge-Sort und Heap-Sort und vergleichen Sie deren Laufzeiten, indem Sie zufällige Zahlenfolgen der Länge  $2^{15}, 2^{16}, \dots, 2^{25}$  als Eingabe erzeugen.

- glibc: modifizierter Quicksort
  - chose pivot by median-of-three decision
  - use insertion sort for small partitions
  - bounded depth of stack
- bionic (android)
  - chose pivot by median-of-three decision
  - use insertion sort for small partitions
  - bounded depth of stack
- musl: smoothsort (variant of heapsort)
- uclibc-ng: Shellsort
- dietlibc: Quicksort with 3-way partitioning

## *Sortieren*

- Quick-Sort – unterschiedliche Varianten
- Heap-Sort
- *Untere Schranke*
- Counting-/Radix-Sort
- Bucket-Sort



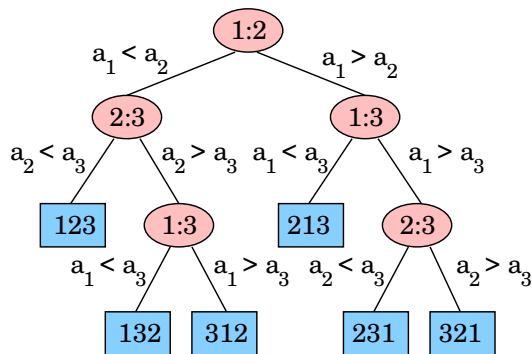
*Allgemeine Sortierverfahren* sind solche Sortierverfahren, die nur Vergleichsoperationen zwischen Schlüsseln verwenden.

*Satz:* Jedes allgemeine Sortierverfahren benötigt zum Sortieren von  $n$  verschiedenen Schlüsseln im schlechtesten Fall und im Mittel wenigstens  $\Omega(n \cdot \log(n))$  Schlüsselvergleiche.

W. Dobosiewicz:

- Sorting by distributive partitioning. Information Processing Letters, 7(1), 1978.
- Sortierverfahren nutzt arithmetische Operationen und die Floor-Funktion und hat im Mittel eine Laufzeit von  $\mathcal{O}(n)$ .

## Entscheidungsbaum:



- Innere Knoten sind mit  $i:j$  beschriftet, wobei  $i, j \in \{1, 2, \dots, n\}$  gilt.
- Linker Teilbaum enthält alle nachfolgenden Vergleiche, falls  $a_i < a_j$ .
- Rechter Teilbaum: alle nachfolgenden Vergleiche, falls  $a_i > a_j$ .

- Jedes Blatt stellt eine Permutation  $(\pi(1), \pi(2), \dots, \pi(n))$  dar und bezeichnet die Sortierung  $a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$ .

Ein Entscheidungsbaum kann jedes allgemeine Sortierverfahren modellieren:

- Es gibt jeweils einen Baum für jede Eingabegröße  $n$ .
- Die Laufzeit des Algorithmus ist gerade die Länge des gewählten Pfades.
- Die Worst-case-Laufzeit entspricht der Höhe des Baums.

*Worst-case Analyse:*

- Es gibt  $n!$  verschiedene Permutationen über  $n$  Zahlen.
- Ein Entscheidungsbaum hat mindestens  $n!$  Blätter.
- Ein Binärbaum der Höhe  $h$  hat maximal  $2^h - 1$  Blätter.

$$\Rightarrow 2^h \geq n!$$

$$\begin{aligned} h &\geq \log(n!) && \text{log ist monoton steigend} \\ &\geq \log((n/e)^n) && \text{Stirling-Formel} \\ &= \log(n^n) - \log(e^n) \\ &= n \cdot \log(n) - n \cdot \log(e) \\ &\in \Omega(n \cdot \log(n)) \end{aligned}$$

*Average-case Analyse:* Beweis durch Widerspruch.

- *Behauptung:* Die mittlere Tiefe eines Entscheidungsbaums mit  $k$  Blättern ist wenigstens  $\log_2(k)$ .
- *Annahme:* Es existiert ein Entscheidungsbaum mit  $k$  Blättern, dessen mittlere Tiefe kleiner als  $\log_2(k)$  ist.

Sei  $T$  ein solcher Baum, der unter all diesen Bäumen der kleinste ist.  $T$  habe  $k$  Blätter. Dann gilt:

- $T$  hat einen linken Teilbaum  $T_1$  mit  $k_1$  Blättern.
- $T$  hat einen rechten Teilbaum  $T_2$  mit  $k_2$  Blättern.
- Es gilt  $k_1 < k$ ,  $k_2 < k$  und  $k_1 + k_2 = k$ .

Da  $T$  der kleinste Baum ist, für den die Annahme gilt, erhalten wir für  $T_1$  und  $T_2$ :

$$\text{mittlere Tiefe}(T_1) \geq \log_2(k_1)$$

$$\text{mittlere Tiefe}(T_2) \geq \log_2(k_2)$$

Jedes Blatt in  $T_1$  bzw.  $T_2$  auf Tiefe  $t$  hat in  $T$  die Tiefe  $t + 1$ . Also gilt insgesamt:

$$\begin{aligned} \text{mT}(T) &= \frac{k_1}{k}(\text{mT}(T_1) + 1) + \frac{k_2}{k}(\text{mT}(T_2) + 1) \\ &\geq \frac{k_1}{k}(\log_2(k_1) + 1) + \frac{k_2}{k}(\log_2(k_2) + 1) \\ &= \frac{1}{k}(k_1 \cdot \log_2(2k_1) + k_2 \cdot \log_2(2k_2)) \end{aligned}$$

Unter der Nebenbedingung  $k_1 + k_2 = k$  hat die Funktion

$$\text{mT}(T) = \frac{1}{k}(k_1 \cdot \log_2(2k_1) + k_2 \cdot \log_2(2k_2))$$

ein Minimum bei  $k_1 = k_2 = k/2$ . Damit gilt

$$\text{mT}(T) \geq \frac{1}{k} \left( \frac{k}{2} \log_2(k) + \frac{k}{2} \log_2(k) \right) = \log_2(k)$$

im Widerspruch zur Annahme.

Jeder Entscheidungsbaum zur Sortierung von  $n$  Zahlen hat also mindestens  $n!$  Blätter, daher gilt:

$$\text{mT}(T) \geq \log_2(n!) \geq \log_2((n/e)^n) \in \Omega(n \cdot \log(n))$$

## *Sortieren*

- Quick-Sort – unterschiedliche Varianten
- Heap-Sort
- Untere Schranke
- *Counting-/Radix-Sort*
- Bucket-Sort

# Counting Sort

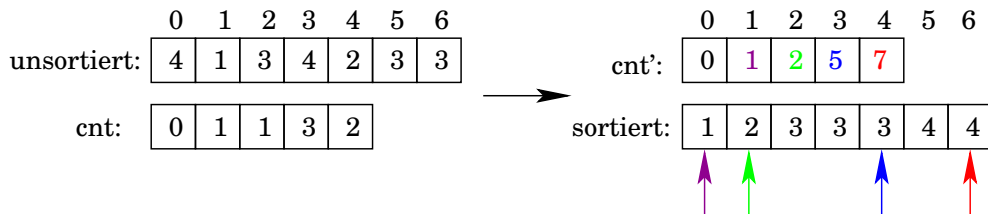
Sortierv Verfahren ohne Schlüsselvergleiche!

*Eingabe:*  $a[0 \dots N]$  mit  $a[i] \in \{0, \dots, k\}$

*Ausgabe:*  $b[0 \dots N]$  sortiert

*Hilfsspeicher:*  $cnt[0 \dots k]$  zum Zählen

*Idee:* Zähle die Häufigkeiten der Zahlen  $0, \dots, k$  in der zu sortierenden Liste und berechne daraus die Position der jeweiligen Zahl in der sortierten Liste.





# Counting Sort

## *Pseudo-Code:*

```
for i := 0 to k-1 do                                /* init */
    cnt[i] := 0
for j := 0 to n-1 do                                /* count */
    cnt[a[j]] := cnt[a[j]] + 1
for i := 1 to k-1 do                                /* collect */
    cnt[i] := cnt[i] + cnt[i-1]
for j := n-1 downto 0 do                             /* rearrange */
    b[cnt[a[j]] - 1] := a[j]
    cnt[a[j]] := cnt[a[j]] - 1
```

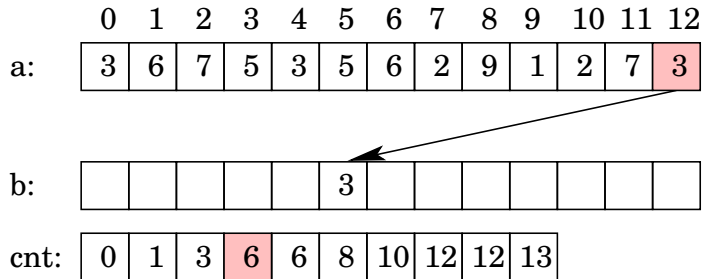
# Counting Sort

	0	1	2	3	4	5	6	7	8	9	10	11	12
a:	3	6	7	5	3	5	6	2	9	1	2	7	3
cnt:	0	1	2	3	0	2	2	2	0	1			

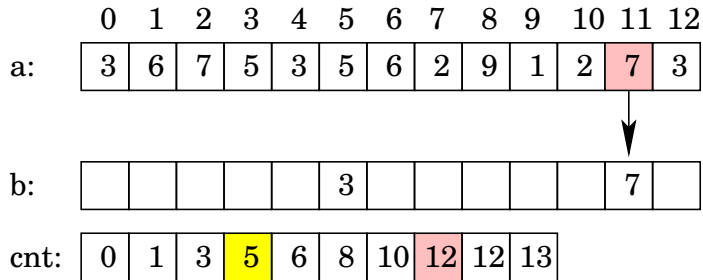
# Counting Sort

	0	1	2	3	4	5	6	7	8	9	10	11	12
a:	3	6	7	5	3	5	6	2	9	1	2	7	3
cnt:	0	1	2	3	0	2	2	2	0	1			
cnt:	0	1	3	6	6	8	10	12	12	13			

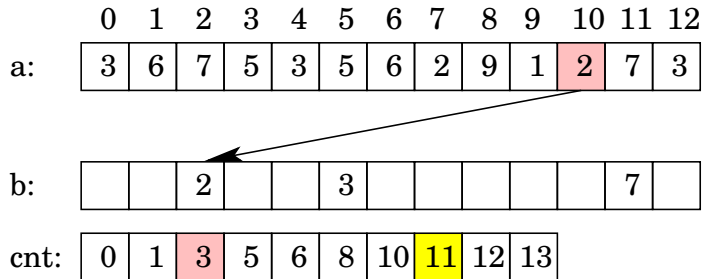
# Counting Sort



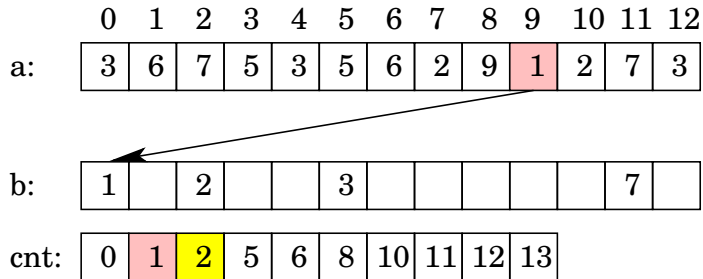
# Counting Sort



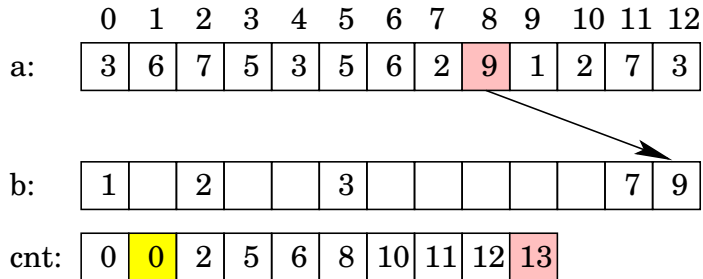
# Counting Sort



# Counting Sort

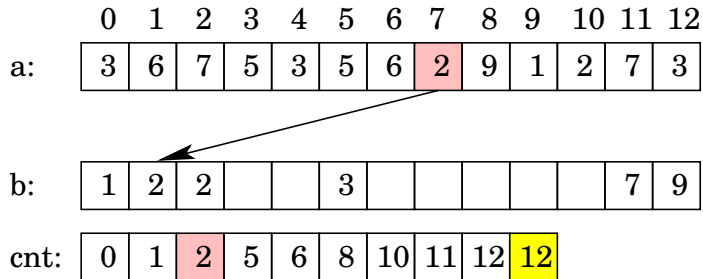


# Counting Sort

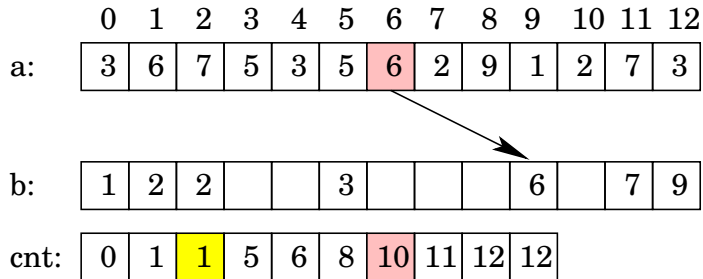




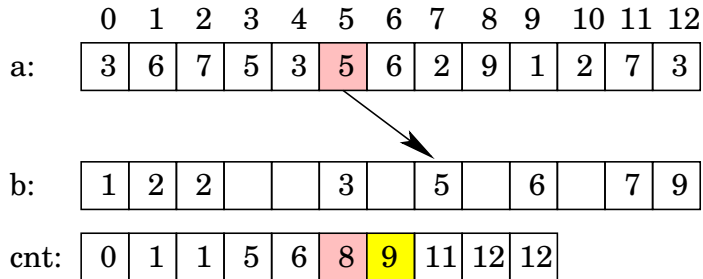
# Counting Sort



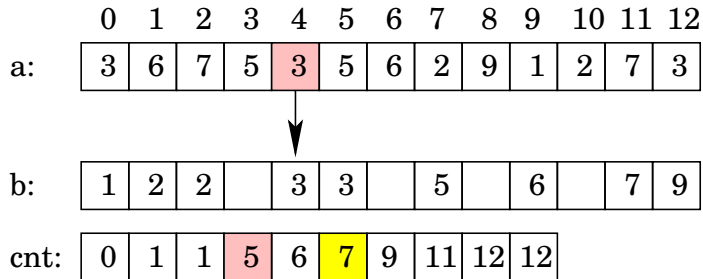
# Counting Sort



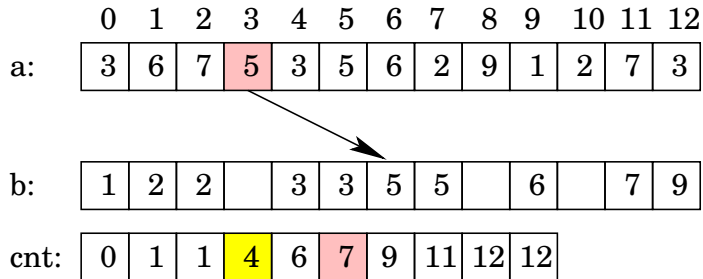
# Counting Sort



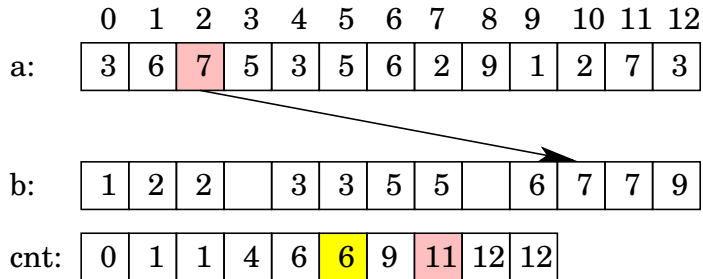
# Counting Sort



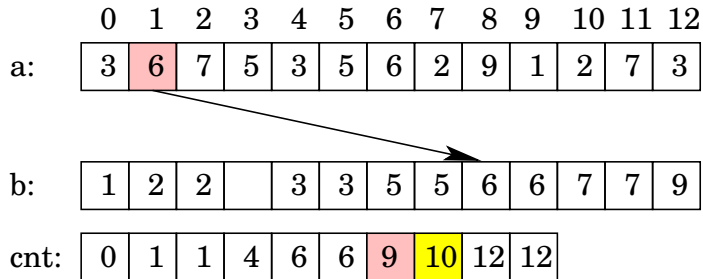
# Counting Sort



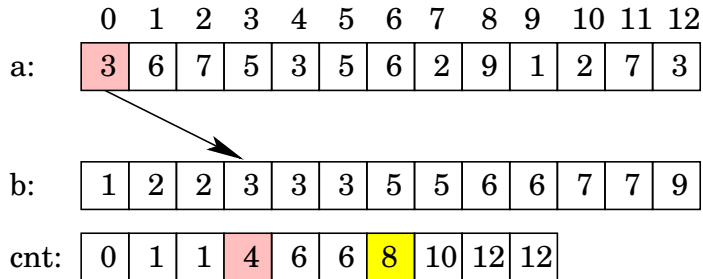
# Counting Sort



# Counting Sort



# Counting Sort

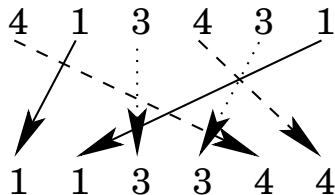




# Counting Sort

<i>Laufzeit:</i>	init	$\Theta(k)$
	count	$\Theta(n)$
	collect	$\Theta(k)$
	rearrange	$\Theta(n)$
		<hr/>
		$\Theta(n + k)$

Counting Sort ist ein *stabiles Sortierverfahren*: Gleiche Elemente stehen nach dem Sortieren in der gleichen Reihenfolge wie vor dem Sortieren.



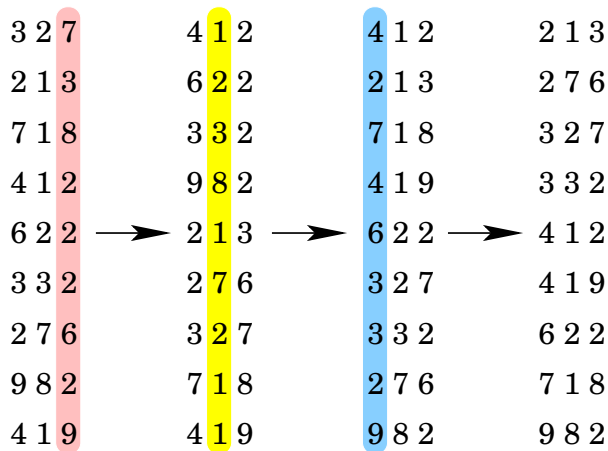
*Problem:* Counting Sort ist ineffizient bzgl. Laufzeit und Speicherplatz, wenn der Wertebereich groß ist im Vergleich zur Anzahl der Zahlen:  $\Theta(n + k) = \Theta(n^2)$  für  $k = n^2$

*Beispiel:* Es sollen 100.000 Datensätze mit einem 32Bit-Schlüssel sortiert werden, also gilt:  $k = 2^{32} \approx 4.300.000.000$

$$\begin{array}{rcl} n + k & \approx & 4.300.000.000 \\ n \cdot \log(n) & \approx & 1.600.000 \end{array}$$

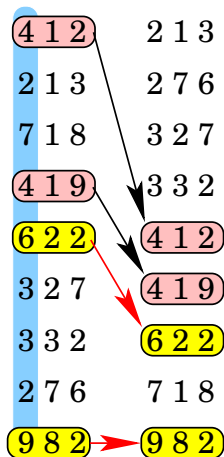
*Lösung:* Sortiere die Zahlen anhand der einzelnen Ziffern, beginnend mit der niederwertigsten Stelle. Verwende ein stabiles Sortierverfahren wie Counting Sort. Falls nötig, müssen führende Nullen eingefügt werden.

*Beispiel:*



*Korrektheit:* Induktion über die betrachtete Position.

- I.A. Nach der ersten Sortierphase sind die Zahlen bezüglich der Position 0 sortiert.
- I.V. Die Zahlen sind bezüglich der  $t - 1$  niederwertigsten Ziffern sortiert.
- I.S. Sortiere nach Ziffer  $t$ :
  - Zwei Zahlen, die sich an Position  $t$  unterscheiden, sind richtig sortiert.
  - Zwei Zahlen, die an Position  $t$  dieselbe Ziffer haben, sind nach I.V. richtig sortiert und bleiben aufgrund des stabilen Sortierverfahrens sortiert.



*Laufzeit:*

- Fasse jeweils  $r$  bit zu einer Ziffer zusammen.
- Bei einer Wortlänge von  $b$  bit müssen  $b/r$  Phasen durchlaufen werden.

*Frage:* Wie groß muss  $r$  im Verhältnis zu  $b$  sein, um eine gute Laufzeit zu erzielen?

- Counting Sort hat Laufzeit  $\Theta(n + k)$ . Hier:  $\Theta(n + 2^r)$
- Bei  $b/r$  Phasen erhalten wir als Gesamtlaufzeit:  $\Theta(b/r \cdot (n + 2^r))$
- Extremwert bestimmen: Bei welchem Wert von  $r$  wird die Laufzeit minimal?

*In der Praxis:* 32bit-Wörter in Gruppen von 8bit  $\Rightarrow$  4 Phasen

## *Sortieren*

- Quick-Sort – unterschiedliche Varianten
- Heap-Sort
- Untere Schranke
- Counting-/Radix-Sort
- *Bucket-Sort*

Wie bei Counting-Sort treffen wir Annahmen über die zu sortierenden Folgen.

- Counting-Sort: ganze Zahlen aus kleinem Wertebereich
- Bucket-Sort: Zahlen aus  $[0, 1)$  sind gleichverteilt

Die Werte des Arrays werden in verschiedene Fächer einsortiert.

**function** BUCKET-SORT( $A$ )

$n := \text{length}(A)$

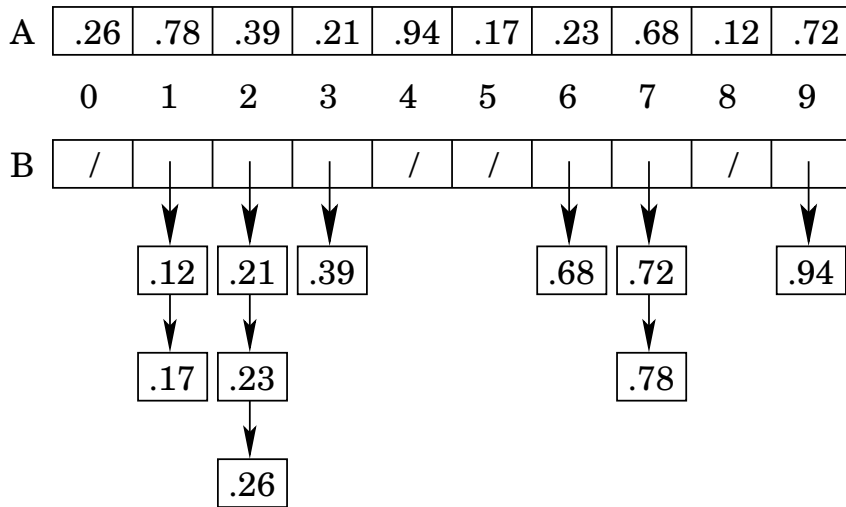
**for**  $i := 1$  to  $n$  **do**

        insert  $A[i]$  into list  $B[\lfloor n \cdot A[i] \rfloor]$

    put values back to  $A$  in order of lists  $B[0], \dots, B[n-1]$

Das Einfügen eines Elements in eine sortierte Liste der Länge  $n$  in Zeile 3 kostet  $\mathcal{O}(n)$  Schritte.

*Beispiel:*





*Laufzeit:*

- worst-case:  $\mathcal{O}(n^2)$
- best-case:  $\mathcal{O}(n)$
- average-case:  $\mathcal{O}(n)$

Sei  $X_i$  eine Zufallsvariable, die die Anzahl der Elemente im Fach  $B[i]$  beschreibt. Für jedes Element  $A[j]$  gilt:

$$Pr(A[j] \text{ fällt in Fach } i) = 1/n$$

Dies gilt unabhängig für alle  $j$ , also liegt ein Bernoulli-Experiment vor. Somit ist  $X_i$  binomialverteilt mit den Werten  $n$  und  $p = 1/n$ . Also gilt:

$$\begin{aligned} E[X_i] &= n \cdot p = 1 \\ Var[X_i] &= n \cdot p \cdot (1 - p) = 1 - 1/n \end{aligned}$$

Außerdem wissen wir:

$$\text{Var}[X_i] = E[X_i^2] - E^2[X_i] \iff E[X_i^2] = \text{Var}[X_i] + E^2[X_i]$$

Die erwartete Zeit zum Einfügen der  $X_i$  vielen Elemente in Fach  $i$  dauert  $E(\mathcal{O}(X_i^2)) = \mathcal{O}(E[X_i^2])$  Zeit. (denn:  $E[a \cdot X_i] = a \cdot E[X_i]$ )

Die erwartete Zeit zum Einsortieren aller Zahlen in die Fächer beträgt also:

$$\begin{aligned} \sum_{i=0}^{n-1} \mathcal{O}(E[X_i^2]) &= \mathcal{O}\left(\sum_{i=0}^{n-1} E[X_i^2]\right) = \mathcal{O}\left(\sum_{i=0}^{n-1} \text{Var}[X_i] + E^2[X_i]\right) \\ &= \mathcal{O}\left(\sum_{i=0}^{n-1} (2 - 1/n)\right) \\ &= \mathcal{O}(2n - 1) \end{aligned}$$

- Einleitung
- Entwurfsmethoden
- Sortieren
- *Auswahlproblem*
- Graphalgorithmen
- Spezielle Graphklassen
- Vorrangwarteschlangen
- Suchbäume
- Amortisierte Laufzeitanalyse
- Algorithmen für moderne Hardware
- Algorithmen für geometrische Probleme
- Randomisierte Algorithmen

## *Auswahlproblem (Median-Berechnung)*

- *Motivation*
- randomisiert
- worst-case

Einfache Selektionsaufgabe:

- Bestimme das Minimum einer Zahlenfolge.
- Bestimme das Maximum einer Zahlenfolge.

Unterscheide:

- Unsortiert Werte  $\Rightarrow$  Alle Werte müssen mindestens einmal betrachtet und verglichen werden.
- Sortierte Werte  $\Rightarrow$  Bei wahlfreiem Zugriff kann in einem einzigen Schritt das Minimum bzw. Maximum bestimmt werden.

*Hier:* Finde zu einer gegebenen Zahlenfolge das  $i$ -kleinste Element.

*Übung:* Wieviele Vergleiche benötigt man, um in einer unsortierten Teilfolge der Länge  $n$  sowohl das Minimum als auch das Maximum zu bestimmen?

## Idee:

- vergleiche Paare  $a[2i]$ ,  $a[2i + 1]$  für  $i = 0, \dots, \frac{n}{2}$  und tausche falls  $a[2i] > a[2i + 1]$
- das Minimum steht auf einer Position mit geradem Index
- das Maximum steht auf einer Position mit ungeradem Index

## Algorithmus:

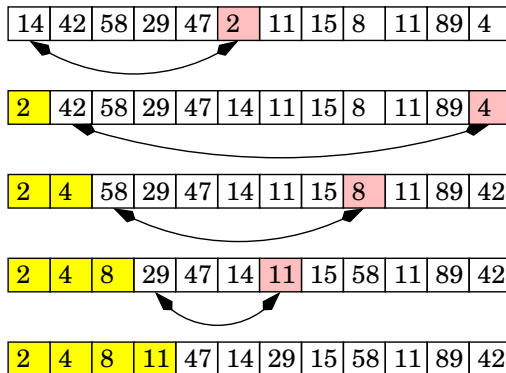
```
for  $i := 0, \dots, n - 1$  do step 2  
    if  $a[i] < a[i + 1]$  then  
        swap  $a[i]$  and  $a[i + 1]$   
  
minimum :=  $\min\{a[0], a[2], a[4], \dots\}$   
maximum :=  $\max\{a[1], a[3], a[5], \dots\}$ 
```

Anzahl Vergleiche:  $3 \cdot \frac{n}{2}$

## Versuch 1: *Iterative Selektion*

- Sortiere Werte mittels Selection Sort.
- Brich den Sortiervorgang ab, wenn  $i$  Elemente sortiert sind.

*Beispiel:* Ermitteln des viertkleinsten Elements einer Folge:



**Laufzeit:**  $\mathcal{O}(i \cdot n)$



nur für kleine Werte von  $i$  geeignet

## Versuch 2: *Sortieren und Selektieren*

Zunächst werden die Elemente mit einem effizienten Algorithmus, wie z.B. Mergesort oder Heapsort sortiert. Anschließend steht das gesuchte Element auf Position  $i$  des Arrays.

*Laufzeit:*  $\mathcal{O}(n \cdot \log(n))$

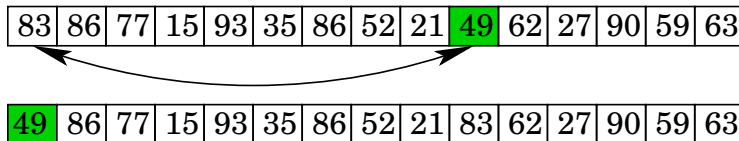


## *Auswahlproblem (Median-Berechnung)*

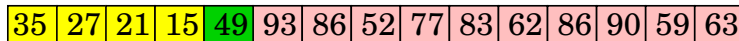
- Motivation
- *randomisiert*
- worst-case

`random-select(l,r,i)` basiert auf der Idee von Quicksort:

- Wähle aus allen Werten ein Pivot-Element zufällig aus und tausche es mit dem Element am linken Rand.



- $k := \text{partition}(l,r)$ : Teile die Folge in zwei Teilfolgen  $L$  und  $R$  auf.



- $i < k \Rightarrow$  Gib `random-select(l,k-1,i)` zurück.
- $i = k \Rightarrow$  Gib das Pivot-Element zurück.
- $i > k \Rightarrow$  Gib `random-select(k+1,r,i)` zurück.

Die Partitionierung erfolgt in Zeit  $\Theta(n)$ .

*Laufzeit:*

- Im günstigen Fall wird die Folge bei jedem rekursiven Aufruf mindestens um einen konstanten Faktor kleiner. Bei der folgenden Abschätzung werden bspw. immer  $\frac{1}{10}$  der Zahlen ausgeschlossen. Dann erhalten wir:

$$T(n) = T(9/10 \cdot n) + \Theta(n) \in \Theta(n)$$

- Im ungünstigen Fall wird die rekursiv zu durchsuchende Folge immer nur um ein Element kleiner. Dann erhalten wir:

$$T(n) = T(n - 1) + \Theta(n) \in \Theta(n^2)$$

*Average-case Laufzeit:* Alle Werte im Array der Länge  $n$  seien verschieden.

- Für jedes  $j \in \{1, \dots, n\}$  gilt: Das Pivot-Element wird mit Wahrscheinlichkeit  $1/n$  das  $j$ -kleinste Element des Arrays.
- Nach der Partitionierung enthält die Teilfolge  $L$  dann  $j - 1$  Elemente,  $R$  enthält  $n - j$  Elemente.

$$\begin{aligned}\overline{T}(n) &\leq \Theta(n) + \frac{1}{n} \cdot \sum_{j=1}^n \overline{T}(\max(j-1, n-j)) \\ &\leq \Theta(n) + \frac{2}{n} \cdot \sum_{j=\lfloor n/2 \rfloor}^{n-1} \overline{T}(j)\end{aligned}$$

Mittels vollständiger Induktion lässt sich leicht zeigen:

$$\sum_{j=\lfloor n/2 \rfloor}^{n-1} j \leq \frac{3}{8}n^2$$

Behauptung:  $\overline{T}(n) \leq c \cdot n$  für ein geeignetes  $c \in \mathbb{R}^+$

Wir überprüfen dies mittels der Einsetzungsmethode:

$$\begin{aligned}\overline{T}(n) &\leq \Theta(n) + \frac{2}{n} \cdot \sum_{j=\lfloor n/2 \rfloor}^{n-1} \overline{T}(j) \\ &\leq c_1 \cdot n + \frac{2}{n} \cdot \sum_{j=\lfloor n/2 \rfloor}^{n-1} c \cdot j \\ &= c_1 \cdot n + \frac{2c}{n} \cdot \sum_{j=\lfloor n/2 \rfloor}^{n-1} j \\ &\leq c_1 \cdot n + \frac{2c}{n} \cdot \frac{3}{8} n^2 = \left(c_1 + \frac{3}{4}c\right) \cdot n\end{aligned}$$

Für  $c \geq 4c_1$  ist damit obige Aussage gezeigt.

## *Auswahlproblem (Median-Berechnung)*

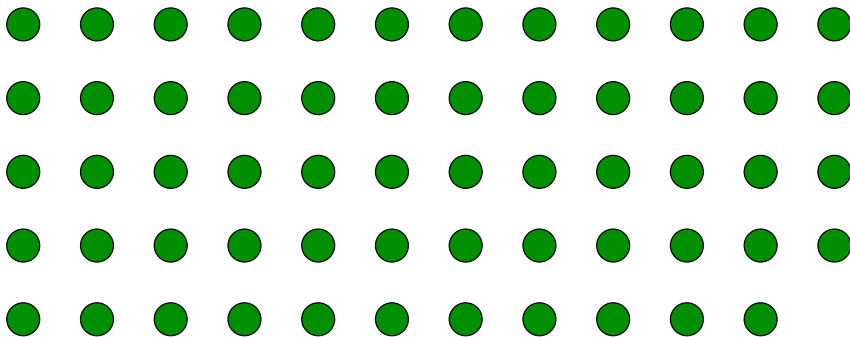
- Motivation
- randomisiert
- *worst-case*

- generate a good pivot recursively, numbers in array A
- due to Blum, Floyd, Pratt, Rivest, and Tarjan [1973]

SELECT(A, l, r, i)

- 1 divide the  $n = r - l + 1$  elements into groups of 5
- 2 find median of each 5-element group and put it into array B
- 3 recursively select the median  $x$  of the  $\lfloor n/5 \rfloor$  group medians to be the pivot  
→ SELECT(B, 0, n/5, n/10)
- 4 partition around the pivot  $x$  and let  $k := \text{partition}(A, l, r)$
- 5 if  $i < k$  return SELECT(A, l, k-1, i)
- 6 if  $i = k$  return pivot
- 7 if  $i > k$  return SELECT(A, k+1, r, i)

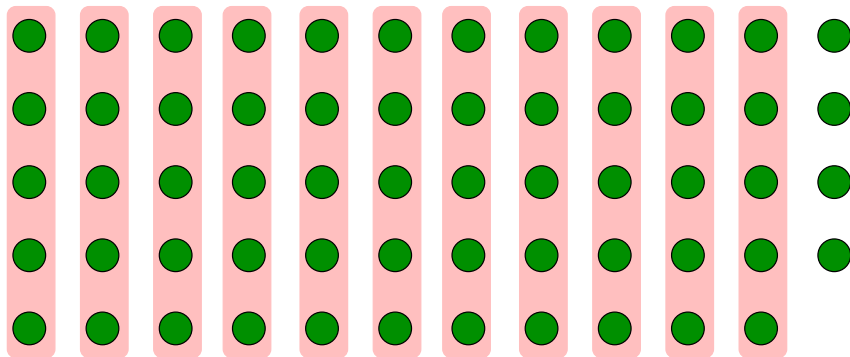
# Median-Strategie



initial:  $n = r - l + 1$  elements

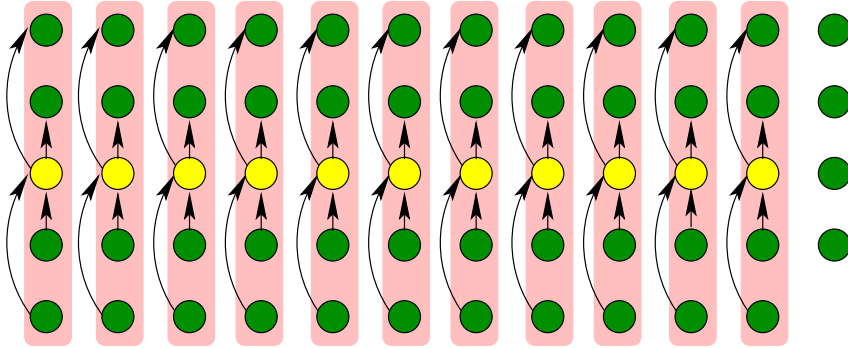


# Median-Strategie



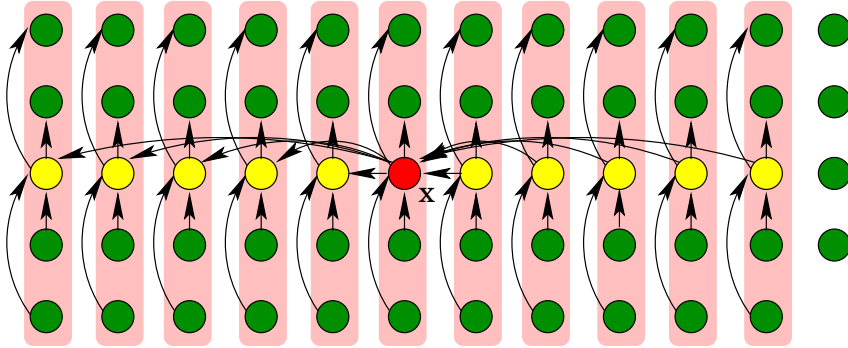
divide the  $n = r - l + 1$  elements into groups of 5

# Median-Strategie



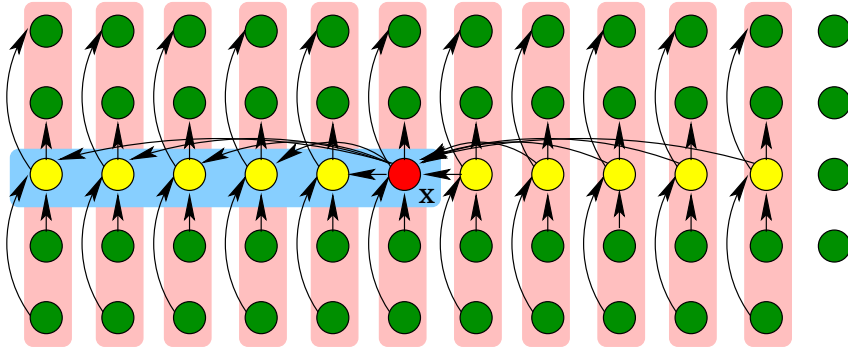
find the median of each 5-element group

# Median-Strategie



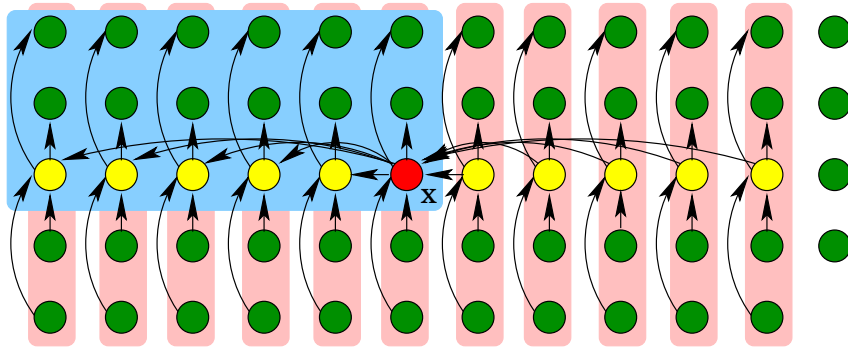
recursively **SELECT** the median  $x$  of the  $\lfloor n/5 \rfloor$  group medians to be the pivot

# Median-Strategie



at least half the group medians are  $\leq x$ , which is at least  $\lfloor \lfloor n/5 \rfloor / 2 \rfloor = \lfloor n/10 \rfloor$  group medians

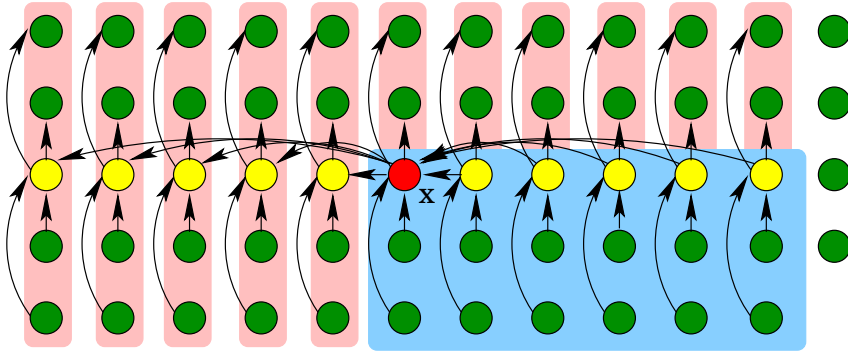
# Median-Strategie



at least half the group medians are  $\leq x$ , which is at least  $\lfloor \lfloor n/5 \rfloor / 2 \rfloor = \lfloor n/10 \rfloor$  group medians

- therefore, at least  $3\lfloor n/10 \rfloor$  elements are  $\leq x$

# Median-Strategie



at least half the group medians are  $\leq x$ , which is at least  $\lfloor \lfloor n/5 \rfloor / 2 \rfloor = \lfloor n/10 \rfloor$  group medians

- therefore, at least  $3\lfloor n/10 \rfloor$  elements are  $\leq x$
- similarly, at least  $3\lfloor n/10 \rfloor$  elements are  $\geq x$

*Laufzeit:*

- Aufwand zum Bestimmen des Medians der Mediane:  $T(n/5)$
- Aufwand zum Bestimmen des  $k$ -kleinsten Elements:  $T(7n/10)$

⇒ Aufwand gesamt:  $T(n) = T(n/5) + T(7n/10) + \Theta(n)$

Vermutung:  $T(n) \leq cn$

$$T(n) \leq \frac{1}{5}cn + \frac{7}{10}cn + kn = \frac{9}{10}cn + kn \stackrel{!}{\leq} cn$$

Wählen wir  $c$  groß genug, so ist die Ungleichung erfüllt:

$$\begin{aligned} \frac{9}{10}cn + kn \leq cn &\iff kn \leq cn - \frac{9}{10}cn = \frac{1}{10}cn \\ &\iff 10k \leq c \end{aligned}$$

## *Übung:*

- Implementieren Sie die vorgestellten Verfahren zur Median-Bestimmung und vergleichen Sie die Laufzeiten für verschieden große, zufällige Zahlenfolgen.
- Welches Verfahren würden Sie in der Praxis einsetzen?
- Implementieren Sie Quicksort mit einer verbesserten Wahl des Pivot-Elements und vergleichen Sie die Version mit den bereits früher erstellten Versionen.