Algorithmen

Markus Lohrey

Universität Siegen

Wintersemester 2016/2017

Überblick, Literatur

Überblick:

1. Grundlegendes
2. Divide & Conquer
3. Sortieren
4. Greedyalgorithmen
5. Dynamische Programmierung
6. Graphalgorithmen

Literatur:

- Cormen, Leiserson Rivest, Stein. Introduction to Algorithms (3. Auflage); MIT Press 2009
- Schöning, Algorithmik. Spektrum Akademischer Verlag 2001

Landau Symbole

Seien $f, g : \mathbb{N} \to \mathbb{N}$ Funktionen.

- Es gilt $g \in \mathcal{O}(f)$, falls
$$\exists c > 0\ \exists n_0\ \forall n \geq n_0 :\ g(n) \leq c \cdot f(n).$$
Also: g wächst nicht schneller als f.
- Es gilt $g \in o(f)$, falls
$$\forall c > 0\ \exists n_0\ \forall n \geq n_0 :\ g(n) \leq c \cdot f(n).$$
Also: g wächst echt langsamer als f.
- $g \in \Omega(f) \Leftrightarrow f \in \mathcal{O}(g)$
Also: g wächst mindestens so schnell wie f.
- $g \in \omega(f) \Leftrightarrow f \in o(g)$
Also: g wächst echt schneller als f.
- $g \in \Theta(f) \Leftrightarrow (f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(f))$
Dies heißt, g und f wachsen asymptotisch gleichschnell.

Jensen's Ungleichung

Sei $f : D \to \mathbb{R}$ eine Funktion, wobei $D \subseteq \mathbb{R}$ ein Intervall ist.

- f ist konvex, falls für alle $x, y \in D$ und all $0 \leq \lambda \leq 1$ gilt: $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)$.
- f ist konkav, falls für alle $x, y \in D$ und all $0 \leq \lambda \leq 1$ gilt: $f(\lambda x + (1-\lambda)y) \geq \lambda f(x) + (1-\lambda)f(y)$.

Jensens Ungleichung

Ist f konkav, so gilt für alle $x_1, \ldots, x_n \in D$ und alle $\lambda_1, \ldots, \lambda_n \geq 0$ mit $\lambda_1 + \cdots + \lambda_n = 1$:

$$f\left(\sum_{i=1}^{n} \lambda_i \cdot x_i\right) \geq \sum_{i=1}^{n} \lambda_i \cdot f(x_i).$$

Ist f konvex, so gilt für alle $x_1, \ldots, x_n \in D$ und alle $\lambda_1, \ldots, \lambda_n \geq 0$ mit $\lambda_1 + \cdots + \lambda_n = 1$:

$$f\left(\sum_{i=1}^{n} \lambda_i \cdot x_i\right) \leq \sum_{i=1}^{n} \lambda_i \cdot f(x_i).$$

Komplexitätsmaße

Wir beschreiben die Laufzeit eines Algorithmus A als eine Funktion in Abhängigkeit von der Eingabelänge n.

Standard: Komplexität im ungünstigsten Fall (worst case).

Maximale Laufzeit über alle Eingaben der Länge n:

$$t_{A,\text{worst}}(n) = \max\{t_A(x) \mid x \in X_n\},$$

wobei $X_n = \{x \mid |x| = n\}$.

Kritik: Unrealistisch, da Worst-Case Eingaben in der Praxis häufig nicht auftreten.

Komplexitätsmaße

Alternative: Komplexität im Mittel (average case).

Benötigt eine Wahrscheinlichkeitsverteilung auf der Menge X_n.

Standard: Gleichverteilung, d.h. $\text{Prob}(x) = \frac{1}{|X_n|}$.

Mittlerer Zeitbedarf:

$$\begin{aligned} t_{A,\text{Mittel}}(n) &= \sum_{x \in X_n} \text{Prob}(x) \cdot t_A(x) \\ &= \frac{1}{|X_n|} \sum_{x \in X_n} t_A(x) \qquad \text{(bei Gleichverteilung)} \end{aligned}$$

Problem: Häufig schwer zu analysieren.

Beispiel: Quicksort

Beim Quicksort-Algorithmus ist die Anzahl der Vergleiche im ungünstigsten Fall $t_Q(n) \in \Theta(n^2)$.

Mittlerer Anzahl der Vergleiche: $t_{Q,\text{Mittel}}(n) = 1.38 n \log n$

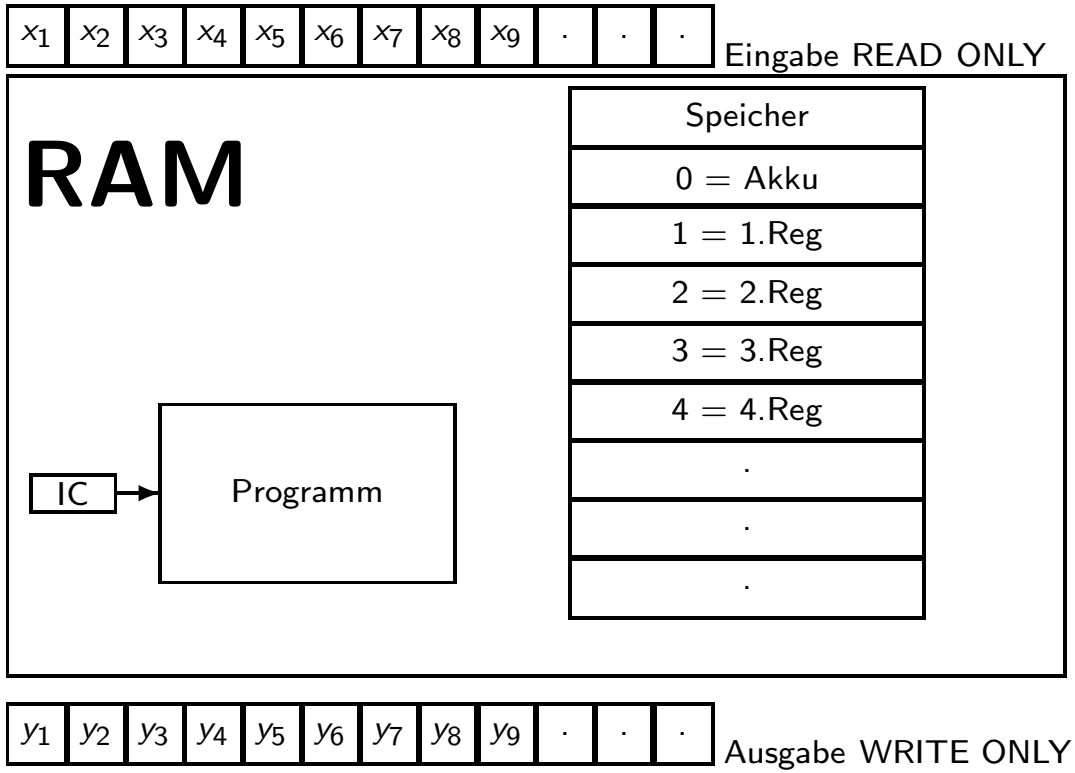
Maschinenmodelle: Turingmaschine

Die Turingmaschine (TM) ist ein mathematisch leicht exakt beschreibbares Berechnungsmodell.

Aber: Der zeitraubende Speicherzugriff (Bandzugriff) in der Realität nicht gegeben.

Arbeitet ein Algorithmus auf einer TM schnell, so *ist* er schnell!

Maschinenmodelle: Registermaschine (RAM)



Annahme: Elementare Registeroperationen (z.B. arithmetische Operationen $+, \times, -$, DIV, Vergleiche, bitweises UND bzw. ODER) können in einem Rechenschritt durchgeführt werden.

Überblick

- Lösen von Rekursionsgleichungen
- Mergesort
- Schnelle Multiplikation ganzer Zahlen
- Matrixmultiplikation nach Strassen
- Schnelle Fourier Transformation

Divide & Conquer: Grundidee

Als erstes zentrales Entwurfsprinzip für Algorithmen wollen wir Divide & Conquer kennenlernen:

Grundidee:

- Zerlege die Eingabe in mehrere (meistens ungefähr gleich große) Teile
- Löse das Problem auf jedem Teil der Eingabe separat (Rekursion!)
- Füge die Teillösungen zu einer Gesamtlösung zusammen.

Rekursionsgleichungen

Divide & Conquer führt auf natürliche Weise zu Rekursionsgleichungen.

Annahmen:

- Eingabe der Länge n wird in a viele Teile der Größe n/b zerlegt.
- Zerlegen sowie Zusammenfügen der Teillösungen benötigt Zeit $g(n)$.
- Für eine Eingabe der Länge 1 beträgt die Rechenzeit $g(1)$.

Dies führt für die Rechenzeit zu folgender Rekursionsgleichung:

$$\begin{aligned} t(1) &= g(1) \\ t(n) &= a \cdot t(n/b) + g(n) \end{aligned}$$

Technisches Problem: Was, wenn n nicht durch b teilbar ist?

- Lösung 1: Ersetze n/b durch $\lceil n/b \rceil$.
- Lösung 2: Wir setzen voraus, dass $n = b^k$ für ein $k \geq 0$.

  Falls dies nicht erfüllt ist: Strecke die Eingabe (Für jede Zahl n existiert ein $k \geq 0$ mit $n \leq b^k < bn$).

Lösen einfacher Rekursionsgleichungen

Theorem 1

Seien $a, b \in \mathbb{N}$ *und* $b > 1$, $g : \mathbb{N} \longrightarrow \mathbb{N}$ *und es gelte die Rekursionsgleichung:*

$$\begin{aligned} t(1) &= g(1) \\ t(n) &= a \cdot t(n/b) + g(n) \end{aligned}$$

Dann gilt für $n = b^k$ *(d.h. für* $k = \log_b(n)$*):*

$$t(n) = \sum_{i=0}^{k} a^i \cdot g\left(\frac{n}{b^i}\right).$$

Beweis: Induktion über k.

$k = 0$: Es gilt $n = b^0 = 1$ und $t(1) = g(1)$.

Lösen einfacher Rekursionsgleichungen

$k > 0$: Nach Induktion gilt

$$t\left(\frac{n}{b}\right) = \sum_{i=0}^{k-1} a^i \cdot g\left(\frac{n}{b^{i+1}}\right).$$

Also:

$$\begin{aligned} t(n) &= a \cdot t\left(\frac{n}{b}\right) + g(n) \\ &= a\left(\sum_{i=0}^{k-1} a^i \cdot g\left(\frac{n}{b^{i+1}}\right)\right) + g(n) \\ &= \sum_{i=1}^{k} a^i \cdot g\left(\frac{n}{b^i}\right) + a^0 g\left(\frac{n}{b^0}\right) \\ &= \sum_{i=0}^{k} a^i \cdot g\left(\frac{n}{b^i}\right). \end{aligned}$$

Mastertheorem I

Theorem 2 (Mastertheorem I)

Seien $a, b, c, d \in \mathbb{N}$, mit $b > 1$ und es gelte die Rekursionsgleichung:

$$\begin{aligned} t(1) &= d \\ t(n) &= a \cdot t(n/b) + d \cdot n^c \end{aligned}$$

Dann gilt für alle n der Form b^k mit $k \geq 0$:

$$t(n) \in \begin{cases} \Theta(n^c) & \text{falls } a < b^c \\ \Theta(n^c \log n) & \text{falls } a = b^c \\ \Theta(n^{\frac{\log a}{\log b}}) & \text{falls } a > b^c \end{cases}$$

Bemerkung: $\frac{\log a}{\log b} = \log_b a$. Falls $a > b^c$, so ist $\log_b a > c$.

Beweis Mastertheorem I

Sei $g(n) = dn^c$. Damit gilt nach Theorem 1 mit $k = \log_b n$:

$$t(n) = d \cdot n^c \cdot \sum_{i=0}^{k} \left(\frac{a}{b^c}\right)^i.$$

1. Fall: $a < b^c$

$$t(n) \leq d \cdot n^c \cdot \sum_{i=0}^{\infty} \left(\frac{a}{b^c}\right)^i = d \cdot n^c \cdot \frac{1}{1 - \frac{a}{b^c}} \in \mathcal{O}(n^c).$$

Außerdem gilt $t(n) \in \Omega(n^c)$. Hieraus folgt $t(n) \in \Theta(n^c)$.

2. Fall: $a = b^c$

$$t(n) = (k+1) \cdot d \cdot n^c \in \Theta(n^c \log n).$$

Beweis Mastertheorem I

3. Fall $a > b^c$

$$
\begin{aligned}
t(n) &= d \cdot n^c \cdot \sum_{i=0}^{k} \left(\frac{a}{b^c}\right)^i = d \cdot n^c \cdot \frac{\left(\frac{a}{b^c}\right)^{k+1} - 1}{\frac{a}{b^c} - 1} \\
&\in \Theta\left(n^c \cdot \left(\frac{a}{b^c}\right)^{\log_b(n)}\right) \\
&= \Theta\left(\frac{n^c \cdot a^{\log_b(n)}}{b^{c \log_b(n)}}\right) \\
&= \Theta\left(a^{\log_b(n)}\right) \\
&= \Theta\left(b^{\log_b(a) \cdot \log_b(n)}\right) \\
&= \Theta\left(n^{\log_b(a)}\right)
\end{aligned}
$$

Strecken der Eingabe macht nichts

Strecken der Eingabe auf b-Potenz Länge ändert nichts an der Aussage des Mastertheorems I.

Formal: Angenommen die Funktion t erfüllt die Rekursionsgleichung

$$\begin{aligned} t(1) &= d \\ t(n) &= a \cdot t(n/b) + d \cdot n^c \end{aligned}$$

für b-Potenzen n.

Definiere die Funktion $s : \mathbb{N} \to \mathbb{N}$ durch $s(n) = t(m)$, wobei m die kleinste b-Potenz größer-gleich n ist ($\rightsquigarrow n \leq m \leq bn$).

Dann gilt nach dem Mastertheorem I

$$s(n) = t(m) \in \begin{cases} \Theta(m^c) = \Theta(n^c) & \text{falls } a < b^c \\ \Theta(m^c \log m) = \Theta(n^c \log n) & \text{falls } a = b^c \\ \Theta(m^{\frac{\log a}{\log b}}) = \Theta(n^{\frac{\log a}{\log b}}) & \text{falls } a > b^c \end{cases}$$

Mastertheorem II

Theorem 3 (Mastertheorem II)

Sei $r > 0$, $\sum_{i=0}^{r} \alpha_i < 1$ *und für eine Konstante* c *sei*

$$t(n) \leq \left(\sum_{i=0}^{r} t(\lceil \alpha_i n \rceil) \right) + c \cdot n$$

Dann gilt $t(n) \in \mathcal{O}(n)$.

Beweis Mastertheorem II

Wähle ein $\varepsilon > 0$ und ein $n_0 > 0$ so, dass

$$\sum_{i=0}^{r} \lceil \alpha_i n \rceil \leq (\sum_{i=0}^{r} \alpha_i) \cdot n + (r+1) \leq (1-\varepsilon)n$$

für alle $n \geq n_0$.

Wähle ein γ so, dass $c \leq \gamma\varepsilon$ und $t(n) \leq \gamma n$ für alle $n < n_0$.

Für den Induktionsschritt ($n \geq n_0$) gilt:

$$\begin{aligned}
t(n) &\leq \left(\sum_{i=0}^{r} t(\lceil \alpha_i n \rceil) \right) + cn \\
&\leq \left(\sum_{i=0}^{r} \gamma \lceil \alpha_i n \rceil \right) + cn \qquad \text{(mit Induktion)} \\
&\leq (\gamma(1-\varepsilon) + c)n \\
&\leq \gamma n
\end{aligned}$$

Mergesort

Wir wollen eine Array A der Länge n sortieren, wobei $n = 2^k$ für ein $k \geq 0$.

Algorithmus mergesort

```
procedure mergesort(l, r)
var m : integer;
begin
  if (l < r) then
    m := (r + l) div 2;
    mergesort(l, m);
    mergesort(m + 1, r);
    merge(l, m, r);
  endif
endprocedure
```

Mergesort

Algorithmus merge

procedure merge(l, m, r)
var i, j, k : **integer**;
begin
  $i = l$; $j := m + 1$;
  for $k := 1$ **to** $r - l + 1$ **do**
    if $i = m + 1$ **or** $(i \leq m$ **and** $j \leq r$ **and** $A[j] \leq A[i])$ **then**
      $B[k] := A[j]$; $j := j + 1$
    else
      $B[k] := A[i]$; $i := i + 1$
    endif
  endfor
  for $k := 0$ **to** $r - l$ **do**
    $A[l + k] := B[k + 1]$
  endfor
endprocedure

Mergesort

Beachte: merge(l, m, r) arbeitet in Zeit $\mathcal{O}(r - l + 1)$.

Laufzeit: $t_{\text{ms}}(n) = 2 \cdot t_{\text{ms}}(n/2) + d \cdot n$ für Konstante d.

Mastertheorem I: $t_{\text{ms}}(n) \in \Theta(n \log n)$.

Wir werden noch sehen, dass $\mathcal{O}(n \log n)$ asymptotisch optimal für vergleichsbasierte Sortieralgorithmen ist.

Nachteil von Mergesort: kein In-Place-Sortieralgorithmus

Ein Sortieralgorithmus arbeitet In-Place, falls zu jedem Zeitpunkt nur eine konstante Zahl von Elementen des Eingabearrays A außerhalb von A gespeichert wird.

Wir werden noch In-Place-Sortieralgorithmen mit einer Laufzeit von $\mathcal{O}(n \log n)$ kennenlernen.

Multiplikation natürlicher Zahlen

Wir wollen zwei natürliche n-bit Zahlen multiplizieren, wobei $n = 2^k$ für ein $k \geq 0$.

Schulmethode: $\Theta(n^2)$ Bit-Operationen.

Anderer Ansatz:

$r =$ | A | B |

$s =$ | C | D |

Dabei sind A (C) die ersten und B (D) die letzten $n/2$ Bits von r (s), d.h.

$$r = A\,2^{n/2} + B; \qquad s = C\,2^{n/2} + D$$

$$r\,s = A\,C\,2^n + (A\,D + B\,C)\,2^{n/2} + B\,D$$

Mastertheorem I: $t_{\text{mult}}(n) = 4 \cdot t_{\text{mult}}(n/2) + \Theta(n) \in \Theta(n^2)$
Nichts gewonnen!

Schnelle Multiplikation nach A. Karatsuba, 1960

Berechne stattdessen besser rekursiv AC, $(A-B)(D-C)$ und BD.

Damit:

$$rs = A\,C\,2^n + (A-B)\,(D-C)\,2^{n/2} + (B\,D + A\,C)\,2^{n/2} + B\,D$$

Gesamtaufwand nach dem Mastertheorem I:

$$t_{\mathsf{mult}}(n) = 3 \cdot t_{\mathsf{mult}}(n/2) + \Theta(n) \in \Theta(n^{\frac{\log 3}{\log 2}}) = \Theta(n^{1.58496\ldots}).$$

Wir haben also durch den Divide & Conquer Ansatz den Exponenten des naiven Ansatzes von 2 auf 1.58496... heruntergesetzt.

Wie schnell kann man Multiplizieren?

1971 konnten Arnold Schönhage and Volker Strassen einen Algorithmus konstruieren, der zwei n-Bit Zahlen in Zeit $\mathcal{O}(n \log n \log\log n)$ auf einer Mehrband-Turingmaschine multipliziert.

Der Schönhage-Strassen Algorithmus basiert auf der schnellen Fouriertransformation (kommt noch) in geeigneten Restklassenringen.

In der Praxis ist der Schönhage-Strassen Algorithmus erst für Zahlen mit ca. 10.000 Dezimalstellen schneller als der Karatsuba Algorithmus, denoch findet er Anwendung in der Praxis.

Erst 2007 konnte der Schönhage-Strassen Algorithmus von Martin Fürer geschlagen werden. Sein Algorithmus hat eine Laufzeit von $\mathcal{O}(n \log n\, 2^{\log^* n})$. Hierbei ist $\log^* n$ die Zahl k, so dass $\log_2$ k-mal angewendet auf n eine Zahl ≤ 1 ergibt.

Matrixmultiplikation mittels naiven Divide & Conquer

Seien $A = (a_{i,j})_{1 \leq i,j \leq n}$ und $B = (b_{i,j})_{1 \leq i,j \leq n}$ zwei $(n \times n)$-Matrizen.

Für die Produktmatrix $AB = (c_{i,j})_{1 \leq i,j \leq n} = C$ gilt

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$$

$\rightsquigarrow \Theta(n^3)$ skalare Multiplikationen.

Divide & Conquer: A, B werden in 4 etwa gleichgroße Untermatrizen unterteilt, wobei sich das Produkt $AB = C$ wie folgt darstellen lässt:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Matrixmultiplikation naiver Divide-and-Conquer

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Dabei ergeben sich folgende Beziehungen:

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

Also:

$$t(n) = 8 \cdot t(n/2) + \Theta(n^2) \in \Theta(n^3)$$

Erneut keine Verbesserung.

Matrixmultiplikation nach Volker Strassen (1969)

Berechne das Produkt von 2×2 Matrizen mit 7 Multiplikationen:

$$
\begin{aligned}
M_1 &:= (A_{12} - A_{22})(B_{21} + B_{22}) \\
M_2 &:= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_3 &:= (A_{11} - A_{21})(B_{11} + B_{12}) \\
M_4 &:= (A_{11} + A_{12})B_{22} \\
M_5 &:= A_{11}(B_{12} - B_{22}) \\
M_6 &:= A_{22}(B_{21} - B_{11}) \\
M_7 &:= (A_{21} + A_{22})B_{11}
\end{aligned}
\qquad
\begin{aligned}
C_{11} &= M_1 + M_2 - M_4 + M_6 \\
C_{12} &= M_4 + M_5 \\
C_{21} &= M_6 + M_7 \\
C_{22} &= M_2 - M_3 + M_5 - M_7
\end{aligned}
$$

Laufzeit: $t(n) = 7 \cdot t(n/2) + \Theta(n^2)$.

Mastertheorem I ($a = 7,\ b = 2,\ c = 2$):

$$t(n) \in \Theta(n^{\log_2 7}) = \Theta(n^{2,81\ldots})$$

Die Geschichte nach Strassen

Strassens Resultat wurde über die Jahre stetig verbessert:

- Strassen 1969: $n^{2,81...}$
- Pan 1979: $n^{2,796...}$
- Bini, Capovani, Romani, Lotti 1979: $n^{2,78...}$
- Schönhage 1981: $n^{2,522...}$
- Romani 1982: $n^{2,517...}$
- Coppersmith, Winograd 1981: $n^{2.496...}$
- Strassen 1986: $n^{2,479...}$
- Coppersmith, Winograd 1987: $n^{2.376...}$
- Stothers 2010: $n^{2,374...}$
- Williams 2014: $n^{2,372873...}$

Konvolution von Polynomen

Betrachte zwei Polynome (mit Koeffizienten etwa aus $\mathbb{C}$):

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots a_n x^n,\ g(x) = b_0 + b_1 x + b_2 x^2 + \cdots b_m x^m$$

repräsentiert durch ihre Koeffizientenfolgen

$$f = (a_0, \ldots, a_n, a_{n+1}, \ldots, a_{N-1}),\ g = (b_0, \ldots, b_m, b_{m+1}, \ldots, b_{N-1})$$

wobei $N = n + m + 1$, $a_{n+1} = \cdots = a_{N-1} = b_{b+1} = \cdots = b_{N-1} = 0$.

Wir wollen das Produktpolynom

$$(fg)(x) = a_0 b_0 + (a_1 b_0 + a_0 b_1)x + \cdots + (a_0 b_{N-1} + \cdots + a_{N-1} b_0)x^{N-1}$$

berechnen, welches durch die Koeffizientenfolge

$$fg = (a_0 b_0, a_1 b_0 + a_0 b_1, \ldots, a_0 b_{N-1} + \cdots + a_{N-1} b_0),$$

(die Konvolution der Folgen f und g) repräsentiert wird.

Punktrepräsentation von Polynomen

Naive Berechnung von fg: $\mathcal{O}(N^2)$ skalare Operationen

FFT (nach James Cooley und John Tukey, 1965) reduziert die Zeit auf $\mathcal{O}(N \log(N))$

Grundidee: Punktrepräsentation von Polynomen

Ein Polynom f vom Grad $N-1$ kann eindeutig durch die Folge der Werte

$$(f(\zeta_0), f(\zeta_1), \ldots, f(\zeta_{N-1}))$$

repräsentiert werden, wobei $\zeta_0, \ldots, \zeta_{N-1}$ N verschiedene Werte aus dem Grundbereich (z.B. komplexe Zahlen), sind.

Offensichtlich gilt $(fg)(\zeta) = f(\zeta)g(\zeta) \quad \rightsquigarrow$

Die Punktrepräsentation der Konvolution von f und g kann in Zeit $\mathcal{O}(N)$ aus den Punktrepräsentationen von f und g berechnet werden.

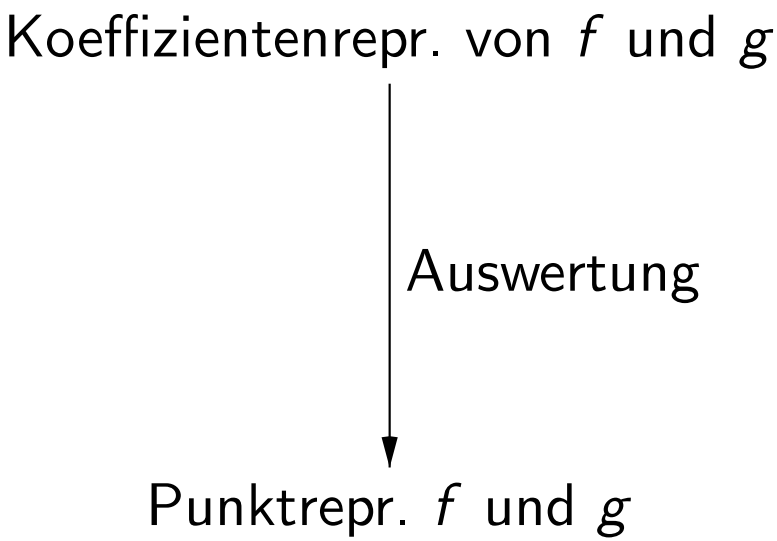
Grundprinzip der FFT

Grundprinzip der schnellen Fourier Transformation (FFT):

Koeffizientenrepr. von f und g

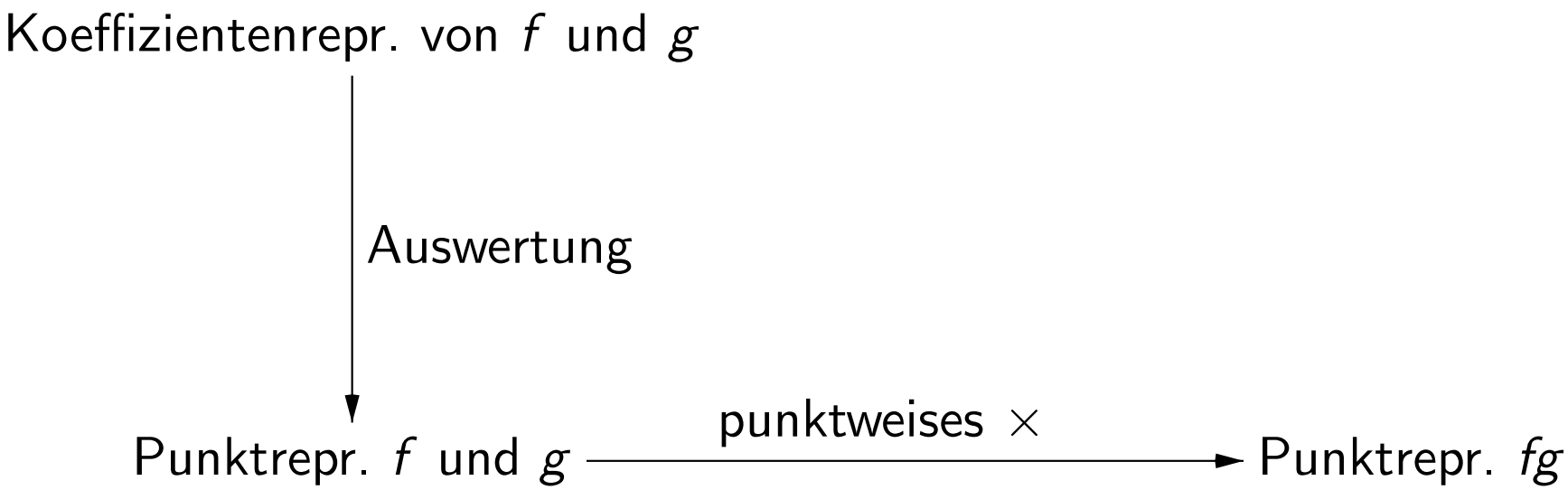
Grundprinzip der FFT

Grundprinzip der schnellen Fourier Transformation (FFT):



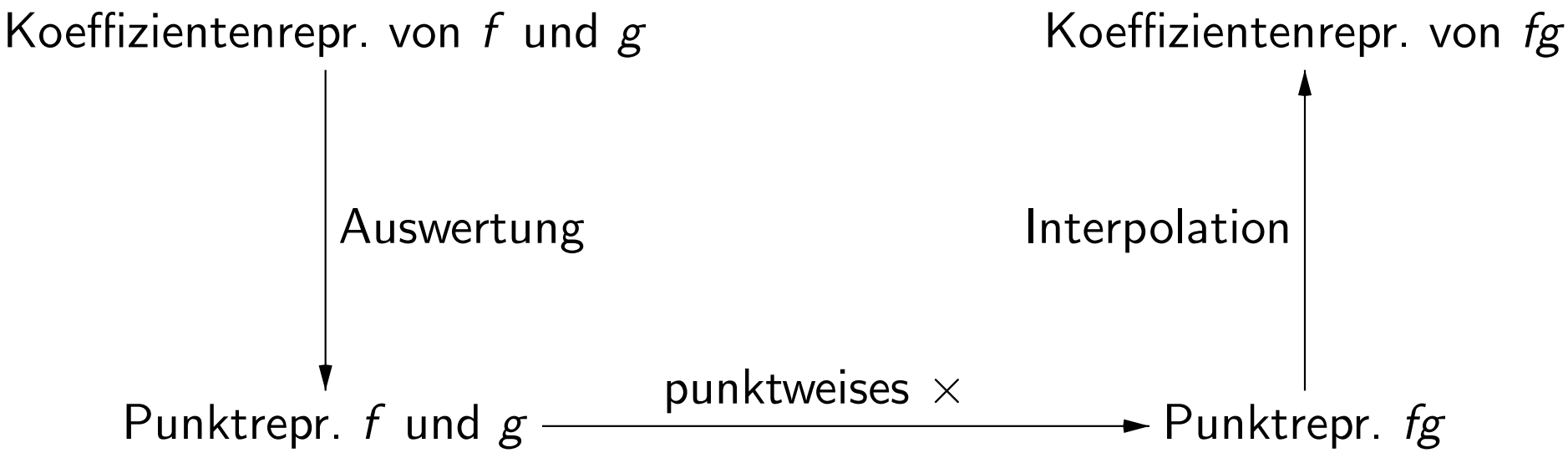
Grundprinzip der FFT

Grundprinzip der schnellen Fourier Transformation (FFT):



Grundprinzip der FFT

Grundprinzip der schnellen Fourier Transformation (FFT):



Einheitswurzeln

Der entscheidende Punkt ist die Auswahl der Punkte $\zeta_0, \zeta_1, \ldots, \zeta_{N-1}$.

Annahme: Die Koeffizienten der Polynome stammen aus einem Körper $\mathbb{F}$, so dass gilt:

- N hat ein multiplikatives Inverses in $\mathbb{F}$, d.h. die Charakteristik von $\mathbb{F}$ teilt nicht N.
- Das Polynom $X^N - 1$ hat N verschiedene Nullstellen – die N-ten Einheitswurzeln – welche sich als ω^i $(0 \leq i < N)$ für eine Nullstelle ω schreiben lassen.

Für $\mathbb{F} = \mathbb{C}$ sind die N-ten Einheitswurzeln etwa von der Form ω^j $(0 \leq j < N)$, wobei $\omega = e^{\frac{2\pi i}{N}}$.

Die Nullstelle ω wird auch als primitive N-te Einheitswurzel bezeichnet.

Einheitswurzeln

Einige nützliche Fakten aus der Algebra:

Sei ω eine primitive N-te Einheitswurzel.

- Für alle $i, j \in \mathbb{Z}$ gilt: $\omega^i = \omega^j$ g.d.w. $i \equiv j \mod N$.
- Für $i \in \mathbb{Z}$ ist ω^i eine primitive N-te Einheitswurzel g.d.w. $\mathrm{ggT}(i, N) = 1$ (wobei $\mathrm{ggT}(i, N)$ der größte gemeinsame Teiler von i und N ist).
- Insbesondere ist $\omega^{-1} = \omega^{N-1}$ eine primitive N-te Einheitswurzel.

Schnelle Fourier Transformation (FFT)

Fixiere eine primitive N-te Einheitswurzel ω.

Wir wählen die Punkte $\zeta_i = \omega^i$ $(0 \leq i \leq N-1)$ für die Auswertung von f und g.

Auswertung von $f = a_0 + a_1 x + \cdots a_{N-1} x^{N-1}$ an den Punkten $\omega^0 = 1, \omega^1, \ldots, \omega^{N-1}$ läuft auf eine Matrixmultiplikation hinaus:

$$\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^1 & \omega^2 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \ldots & \omega^{(N-1)^2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{N-1} \end{pmatrix} = \begin{pmatrix} f(1) \\ f(\omega) \\ f(\omega^2) \\ \vdots \\ f(\omega^{N-1}) \end{pmatrix}$$

Die durch die Matrix $F_N(\omega) = (\omega^{ij})_{0 \leq i,j < N}$ realisierte lineare Abbildung wird als diskrete Fourier Transformation bezeichnet.

Inverse FFT

Lemma 4

$(F_N(\omega))^{-1} = \frac{1}{N} F_N(\omega^{-1})$, d.h. das Inverse der Matrix $(\omega^{ij})_{0 \le i,j < N}$ ist $\left(\frac{\omega^{-ij}}{N}\right)_{0 \le i,j < N}$ (beachte: ω^{-1} ist eine primitive N-te Einheitswurzel).

Beweis: Da $x^N - 1 = (x-1) \cdot \sum_{j=0}^{N-1} x^j$ gilt, ist jedes ω^i für $0 < i < N$ eine Nullstelle von $\sum_{j=0}^{N-1} x^j$.

Daher gilt für alle $0 \le i < N$:

$$\sum_{j=0}^{N-1} \omega^{i \cdot j} = \begin{cases} 0 & \text{falls } i > 0 \\ N & \text{falls } i = 0 \end{cases}$$

Wir erhalten für alle $0 \le i, j < N$:

$$\sum_{k=0}^{N-1} \omega^{ik} \omega^{-kj} = \sum_{k=0}^{N-1} \omega^{k(i-j)} = \begin{cases} 0 & \text{falls } i \neq j \\ N & \text{falls } i = j \end{cases}$$

FFT mittels Divide & Conquer

Wir müssen nun noch die diskrete Fouriertransformation

$$f \mapsto F_N(\omega)f$$

(wobei $f = (a_0, a_1, \ldots, a_{N-1})^T$) in Zeit $\mathcal{O}(N \log(N))$ berechnen.

Dann kann die inverse diskrete Fouriertransformation (= Interpolation)

$$h \mapsto (F_N(\omega))^{-1}h = \frac{1}{N}F_N(\omega^{-1})h$$

in der gleichen Zeitschranke berechnet werden.

Die "Schulmethode" für die Multiplikation einer Matrix mit einem Vektor benötigt Zeit $\mathcal{O}(N^2)$: kein Gewinn gegenüber der "Schulmethode" für Polynommultiplikation.

Wir berechnen $F_N(\omega)f = (\omega^{ij})_{0 \leq i,j < N}f$ mittel Divide & Conquer.

FFT mittels Divide & Conquer

Angenommen N ist gerade. Für $f(x) = a_0 + a_1 x + \cdots + a_{N-1} x^{N-1}$ sei

$$
\begin{aligned}
f_0(x) &= a_0 + a_2 x^2 + a_4 x^4 + \cdots + a_{N-2} x^{N-2} \\
\widehat{f}_0(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{N-2} x^{\frac{N-2}{2}} \\
f_1(x) &= a_1 + a_3 x^2 + a_5 x^4 + \cdots + a_{N-1} x^{N-2} \\
\widehat{f}_1(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{N-1} x^{\frac{N-2}{2}}
\end{aligned}
$$

Also: $f(x) = f_0(x) + x f_1(x)$, $f_0(x) = \widehat{f}_0(x^2)$ und $f_1(x) = \widehat{f}_1(x^2)$.

Die Polynome $\widehat{f}_0(x)$ und $\widehat{f}_1(x)$ haben Grad $\leq \frac{N-2}{2}$.

Sei $0 \leq i < N$. Da ω^2 eine primitive $\frac{N}{2}$-Einheitswurzel ist, gilt:

$$
\begin{aligned}
(F_N(\omega) f_0)_i = f_0(\omega^i) = \widehat{f}_0(\omega^{2i}) = \widehat{f}_0(\omega^{2i \bmod N}) = \\
\widehat{f}_0(\omega^{2(i \bmod \frac{N}{2})}) = \widehat{f}_0((\omega^2)^{i \bmod \frac{N}{2}}) = (F_{\frac{N}{2}}(\omega^2) \widehat{f}_0)_{i \bmod \frac{N}{2}},
\end{aligned}
$$

FFT mittels Divide & Conquer

Wir erhalten

$$F_N(\omega)f_0 = \begin{pmatrix} F_{\frac{N}{2}}(\omega^2)\widehat{f_0} \\ F_{\frac{N}{2}}(\omega^2)\widehat{f_0} \end{pmatrix}$$

und analog

$$F_N(\omega)f_1 = \begin{pmatrix} F_{\frac{N}{2}}(\omega^2)\widehat{f_1} \\ F_{\frac{N}{2}}(\omega^2)\widehat{f_1} \end{pmatrix}.$$

Mit $f(x) = f_0(x) + xf_1(x)$ folgt

$$F_N(\omega)f = F_N(\omega)f_0 + (F_N(\omega)x \circ F_N(\omega)f_1),$$

wobei $F_N(\omega)x = (1, \omega, \omega^2, \ldots, \omega^{N-1})^T$ und "$\circ$" die punktweise Multiplikation von Vektoren bezeichnet.

FFT mittels Divide & Conquer

Wir haben somit die Berechnung von $F_N(\omega)f$ reduziert auf:

- Die Berechnung von $F_{\frac{N}{2}}(\omega^2)\widehat{f_0}$ und $F_{\frac{N}{2}}(\omega^2)\widehat{f_1}$ (2 FFTs der Dimension $N/2$)
- $\mathcal{O}(N)$ viele weitere arithmetische Operationen.

Wir erhalten somit die Rekursionsgleichung

$$T_{\text{fft}}(N) = 2\,T_{\text{fft}}(N/2) + dN$$

für eine Konstante d.

Mastertheorem I ($a = b = 2, c = 1$):

$$T_{\text{fft}}(N) \in \theta(N \log N).$$

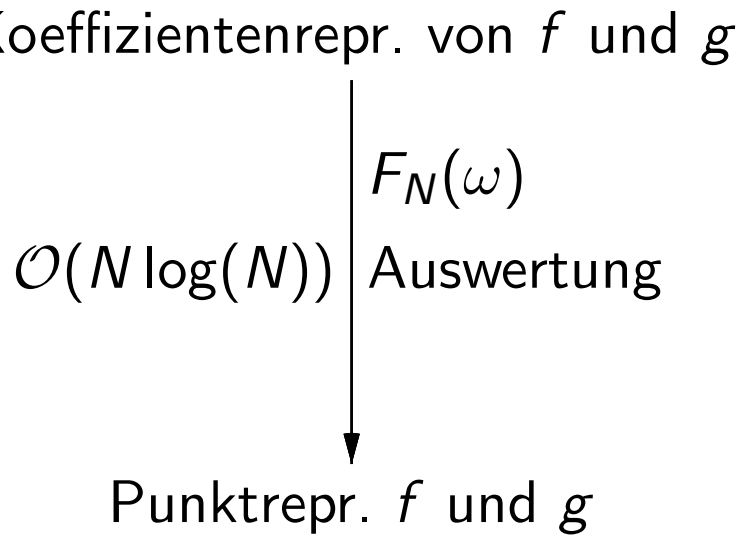
Schnelle Fourier Transformation (FFT)

Grundprinzip der FFT:

Koeffizientenrepr. von f und g

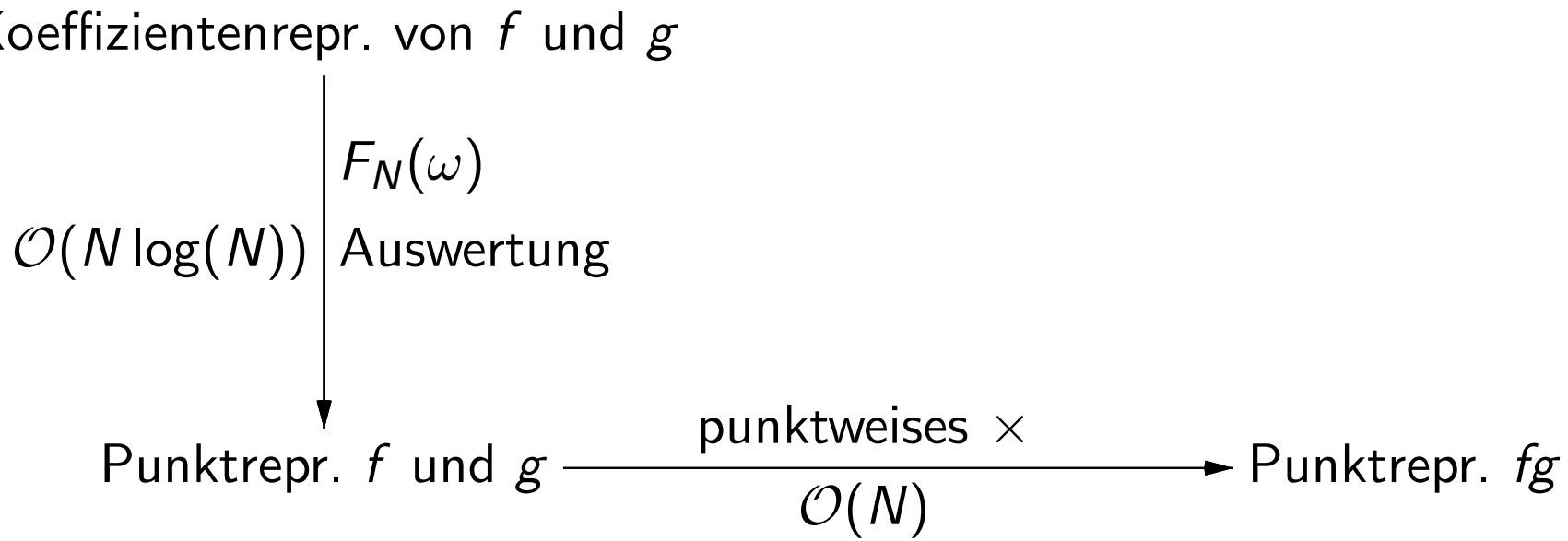
Schnelle Fourier Transformation (FFT)

Grundprinzip der FFT:



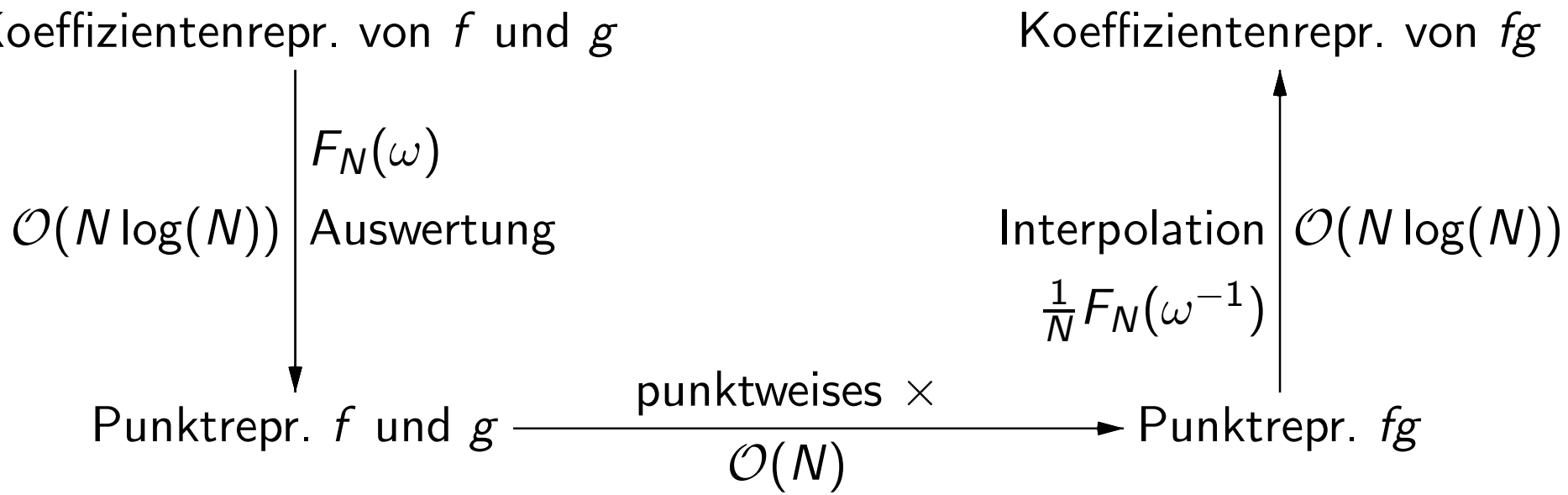
Schnelle Fourier Transformation (FFT)

Grundprinzip der FFT:



Schnelle Fourier Transformation (FFT)

Grundprinzip der FFT:



Überblick

- Untere Schranke für vergleichsbasierte Sortieralgorithmen
- Quicksort
- Heapsort
- Sortieren in linearer Zeit
- Medianberechnung

Vergleichsbasierte Sortieralgorithmen

Ein Sortieralgorithmus ist vergleichsbasiert falls die Elemente des Arrays einen Datentypen bilden, dessen einzige Operation der Vergleich zweier Elemente ist.

Wir gehen für die folgende Betrachtung davon aus, dass das Input-Array $A[1, \ldots, n]$ folgende Eigenschaften hat:

- $A[i] \in \{1, \ldots, n\}$ für alle $1 \leq i \leq n$.
- $A[i] \neq A[j]$ für $i \neq j$

In anderen Worten: Die Eingabe ist eine Permutation der Liste $[1, 2, \ldots, n]$.

Der Sortieralgorithmus soll diese Liste sortieren.

Andere Sichtweise: Der Sortieralgorithmus soll die Permutation $[i_1, i_2, \ldots, i_n]$ ausgeben, so dass $A[i_k] = k$ für alle $1 \leq k \leq n$ gilt.

Beispiel: Bei Eingabe $[2, 3, 1]$ soll $[3, 1, 2]$ ausgegeben werden.

Untere Schranke für den Worst-Case

Theorem 5

Für jeden vergleichsbasierten Sortieralgorithmus und jedes n existiert ein Array der Länge n, auf dem der Algorithmus mindestens

$$n \log_2(n) - \log_2(e)n \geq n \log_2(n) - 1,443n$$

viele Vergleiche macht.

Beweis: Wir führen den Algorithmus in Gedanken auf einem Array $A[1, \ldots, n]$ aus, ohne dabei die konkreten Werte $A[i]$ zu kennen.

Dies ergibt einen Entscheidungsbaum, der wie folgt konstruiert ist.

Angenommen, der Algorithmus vergleicht als erstes $A[i]$ und $A[j]$.

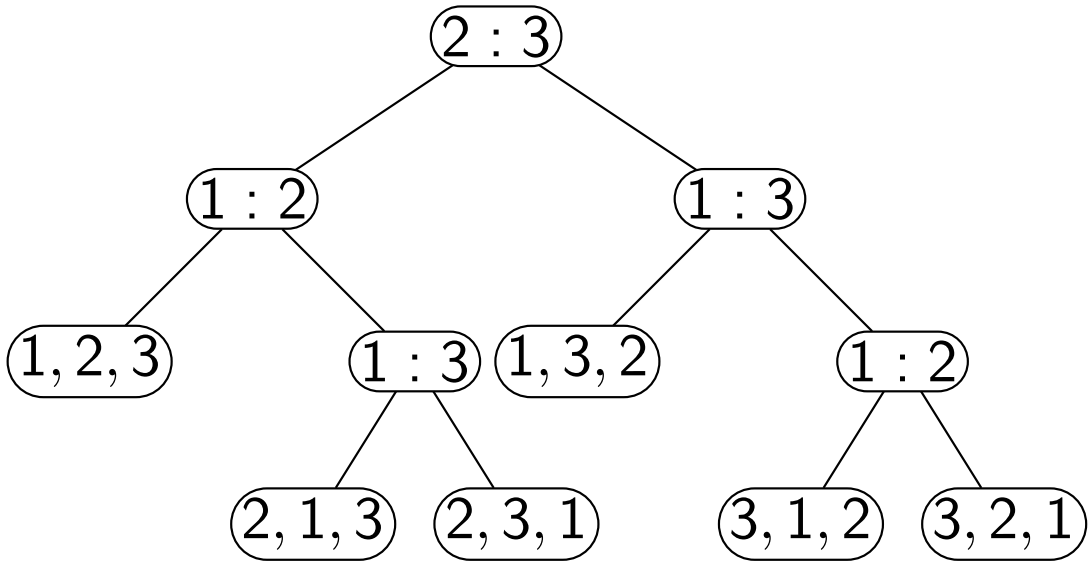
Wir beschriften die Wurzel des Entscheidungsbaums mit $i : j$.

Der linke (rechte) Teilbaum ergibt sich, indem wir den Algorithmus unter der Annahme $A[i] < A[j]$ ($A[i] > A[j]$) weiter laufen lassen.

Untere Schranke für den Worst-Case

Dies ergibt einen Binärbaum mit $n!$ vielen Blättern, denn jede Eingabepermutation muss zu einem anderen Blatt führen.

Beispiel: Hier ist ein Entscheidungsbaum für das Sortieren eines Arrays der Länge 3.



Untere Schranke für den Worst-Case

Beachte: Die Tiefe (= max. Anzahl der Kanten von der Wurzel zu einem Blatt) des Entscheidunbsbaums ist die maximale Anzahl von Vergleichen des Algorithmus auf einem Array der Länge n.

Ein Binärbaum mit N vielen Blättern hat Tiefe mindestens $\log_2(N)$.

Stirlings Formel (wir benötigen nur $n! > \sqrt{2\pi n}(n/e)^n$) impliziert

$$\log_2(n!) \geq n\log_2(n) - \log_2(e)n + \Omega(\log n) \geq n\log_2(n) - 1,443n.$$

Also gibt es eine Eingabepermutation, für die der Algorithmus mindestens $n\log_2(n) - 1,443n$ viele Vergleiche macht. □

Untere Schranke für den Average-Case

Ein vergleichsbasierter Sortieralgorithmus macht sogar auf fast allen Permutationen mindestens $n \log_2(n) - 2,443n$ viele Vergleiche.

Theorem 6

Für jeden vergleichsbasierten Sortieralgorithmus gilt: Der Anteil aller Permutationen, auf denen der Algorithmus mindestens

$$\log_2(n!) - n \geq n \log_2(n) - 2,443n$$

viele Vergleiche macht, ist mindestens $1 - 2^{-n+1}$.

Für den Beweis benötigen wir ein einfaches Lemma:

Lemma 7

Sei $A \subseteq \{0,1\}^*$ *mit* $|A| = N$, *und sei* $1 \leq n < \log_2(N)$. *Dann haben mindestens* $(1 - 2^{-n+1})N$ *viele Wörter in* A *Länge* $\geq \log_2(N) - n$.

Untere Schranke für den Average-Case

Betrachte nun wieder den Entscheidungsbaum; er hat $n!$ Blätter, und jedes Blatt entspricht einer Permutation der Zahlen $\{1, \ldots, n\}$.

Jede der $n!$ vielen Permutationen kann daher durch ein Wort über dem Alphabet $\{0, 1\}$ repräsentiert werden:

- 0 bedeutet: Gehe im Entscheidungsbaum zum linken Kind.
- 1 bedeutet: Gehe im Entscheidunbsbaum zum rechten Kind.

Lemma 7 $\rightsquigarrow$ Der Entscheidungsbaum hat mindestens $(1 - 2^{-n+1})n!$ viele Wurzel-Blatt Pfade der Länge $\geq \log_2(n!) - n \geq n \log_2(n) - 2,443n$. □

Untere Schranke für den Average-Case

Korollar

Jeder vergleichsbasierte Sortieralgorithmus benötigt im Durchschnitt mindestens $n \log_2(n) - 2,443n$ viele Vergleiche zum Sortieren eines Arrays der Länge n (für n groß genug).

Beweis: Wegen Theorem 6 werden im Durchschnitt mindestens

$$\begin{aligned}
(1 - 2^{-n+1}) \cdot (\log_2(n!) - n) + 2^{-n+1} &= \\
\log_2(n!) - n - \frac{\log_2(n!) - n - 1}{2^{n-1}} &\geq \\
n \log_2(n) - 2,443n + \Omega(\log_2 n) - \frac{\log_2(n!) - n - 1}{2^{n-1}} &\geq \\
n \log_2(n) - 2,443n &
\end{aligned}$$

viele Vergleiche gemacht. □

Quicksort

Der Quicksort-Algorithmus (Tony Hoare, 1962):

- Wähle ein Array-Element $A[i]$ (das Pivotelement).
- Partitionieren: Sortiere das Array so um, so dass links (bzw. rechts) vom Pivoelement alle Elemente stehen die kleiner gleich (bzw. größer) als das Pivoelement sind (benötigt $n-1$ Vergleiche).
- Wende den Algorithmus rekursiv auf die Subarrays links und rechts vom Pivoelement an.

Kritisch: die Wahl des Pivotelements.

- Laufzeit ist optimal, falls das Pivotelement gleich dem mittleren Element des Feldes (Median) ist.
- In der Praxis bewährt: die Median-aus-Drei-Methode.

Partitionieren

Zunächst geben wir die Prozedur zum Partitionieren eines Subarrays $A[\ell, \ldots, r]$ bzgl. eines Pivotelements $P = A[p]$ an, wobei $\ell < r$ und $\ell \leq p \leq r$ gelte.

Ergebnis dieser Prozedur ist ein Index $m \in \{\ell, \ldots, r\}$, der folgende Eigenschaften erfüllt:

- $A[m] = P$
- $A[k] \leq P$ für alle $\ell \leq k \leq m - 1$
- $A[k] > P$ für alle $m + 1 \leq k \leq r$

Partitionieren

Algorithmus Partitionieren

function partitioniere($A[\ell \ldots r]$: **array of integer**, p : **integer**) : **integer**
begin
  swap(p, r);
  $P := A[r]$;
  $i := \ell - 1$;
  for $j := \ell$ **to** $r - 1$ **do**
    if $A[j] \leq P$ **then**
      $i := i + 1$;
      swap(i, j)
    endif
  endfor
  swap$(i + 1, r)$
  return $i + 1$
endfunction

Partitionieren

Folgende Invarianten gelten vor jeder Iteration der **for**-Schleife:

- $A[r] = P$
- Für alle $\ell \leq k \leq i$ gilt $A[k] \leq P$
- Für alle $i+1 \leq k \leq j-1$ gilt $A[k] > P$

Somit gilt vor der **return**-Anweisung:

- $A[k] \leq P$ für alle $\ell \leq k \leq i+1$
- $A[k] > P$ für alle $i+2 \leq k \leq r$
- $A[i+1] = P$

Beachte: partitioniere($A[\ell \ldots r]$) macht $r - \ell$ viele Vergleiche.

Quicksort

Algorithmus Quicksort

```
procedure quicksort(A[ℓ ... r] : array of integer)
begin
  if ℓ < r then
    p := Index des Medians von A[ℓ], A[(ℓ + r) div 2], A[r];
    m := partitioniere(A[ℓ ... r], p);
    quicksort(A[ℓ ... m − 1]);
    quicksort(A[m + 1 ... r]);
  endif
endprocedure
```

Worst-Case Laufzeit: $\mathcal{O}(n^2)$.

Tritt ein, wenn nach jedem Aufruf von partitioniere($A[\ell \ldots r]$, p) eines der beiden Teilarrays ($A[\ell \ldots m-1]$ oder $A[m+1 \ldots r]$) leer ist.

Quicksort: Durschnittsanalyse

Durchschnittsanalyse unter der Annahme einer zufälliger Auswahl des Pivotelements.

Alternativ: Eingabearray ist zufällig angeordnet.

Es sei $Q(n)$ die mittlere Anzahl der Schlüsselvergleiche bei einem Eingabearray der Länge n.

Theorem 8

Es gilt $Q(n) = 2(n+1)H(n) - 4n$, *wobei*

$$H(n) := \sum_{k=1}^{n} \frac{1}{k}$$

die n-te harmonische Zahl ist.

Quicksort: Durschnittsanalyse

Beweis: Für $n = 1$ gilt offensichtlich $Q(1) = 0 = 2 \cdot 2 \cdot 1 - 4 \cdot 1$.

Für $n \geq 2$ gilt:

$$\begin{aligned} Q(n) &= (n-1) + \frac{1}{n} \sum_{i=1}^{n} [Q(i-1) + Q(n-i)] \\ &= (n-1) + \frac{2}{n} \sum_{i=1}^{n} Q(i-1) \end{aligned}$$

Dabei ist:

- $(n-1)$ = Zahl der Vergleiche beim Partitionieren
- $Q(i-1) + Q(n-i)$ = mittlere Zahl der Vergleiche für das rekursive Sortieren der beiden Teilhälften.

Der Faktor $1/n$ ergibt sich, da alle Positionen für das Pivotelement gleich wahrscheinlich sind.

Quicksort: Durschnittsanalyse

Damit gilt:

$$nQ(n) = n(n-1) + 2\sum_{i=1}^{n} Q(i-1)$$

Also:

$$\begin{aligned} nQ(n) - (n-1)Q(n-1) &= n(n-1) + 2\sum_{i=1}^{n} Q(i-1) \\ &\quad -(n-1)(n-2) - 2\sum_{i=1}^{n-1} Q(i-1) \\ &= n(n-1) - (n-2)(n-1) + 2Q(n-1) \\ &= 2(n-1) + 2Q(n-1) \end{aligned}$$

Wir erhalten:

$$\begin{aligned} nQ(n) &= 2(n-1) + 2Q(n-1) + (n-1)Q(n-1) \\ &= 2(n-1) + (n+1)Q(n-1) \end{aligned}$$

Quicksort: Durschnittsanalyse

Indem wir durch $n(n+1)$ teilen, erhalten wir

$$\frac{Q(n)}{n+1} = \frac{2(n-1)}{n(n+1)} + \frac{Q(n-1)}{n}$$

Hieraus folgt mit Induktion nach n:

$$\begin{aligned}
\frac{Q(n)}{n+1} &= \sum_{k=1}^{n} \frac{2(k-1)}{k(k+1)} \\
&= 2\sum_{k=1}^{n} \frac{(k-1)}{k(k+1)} \\
&= 2\Big(\sum_{k=1}^{n} \frac{k}{k(k+1)} - \sum_{k=1}^{n} \frac{1}{k(k+1)}\Big)
\end{aligned}$$

Quicksort: Durschnittsanalyse

$$
\begin{aligned}
\frac{Q(n)}{n+1} &= 2\left[\sum_{k=1}^{n}\frac{1}{k+1} - \sum_{k=1}^{n}\frac{1}{k(k+1)}\right] \\
&= 2\left[\sum_{k=1}^{n}\frac{2}{k+1} - \sum_{k=1}^{n}\frac{1}{k}\right] \\
&= 2\left[2\left(\frac{1}{n+1} + H(n) - 1\right) - H(n)\right] \\
&= 2H(n) + \frac{4}{n+1} - 4.
\end{aligned}
$$

Quicksort: Durschnittsanalyse

Schließlich erhält man für $Q(n)$:

$$\begin{aligned} Q(n) &= 2(n+1)H(n) + 4 - 4(n+1) \\ &= 2(n+1)H(n) - 4n. \qquad \square \end{aligned}$$

Es ist $H(n) - \ln n \approx 0{,}57721\ldots =$ Eulersche Konstante. Also:

$$\begin{aligned} Q(n) &\approx 2(n+1)(0{,}58 + \ln n) - 4n \\ &\approx 2n \ln n - 2{,}8n \approx 1{,}38n \log n - 2{,}8n. \end{aligned}$$

Theoretische Grenze: $\log(n!) \approx n \log n - 1{,}44n$;
Quicksort ist im Mittel um 38% schlechter.

Die Durchschnittsanalyse der Median-aus-Drei Methode liefert $1{,}18n \log n - 2{,}2n$.
Dies ist im Mittel nur noch um 18% schlechter.

Heaps

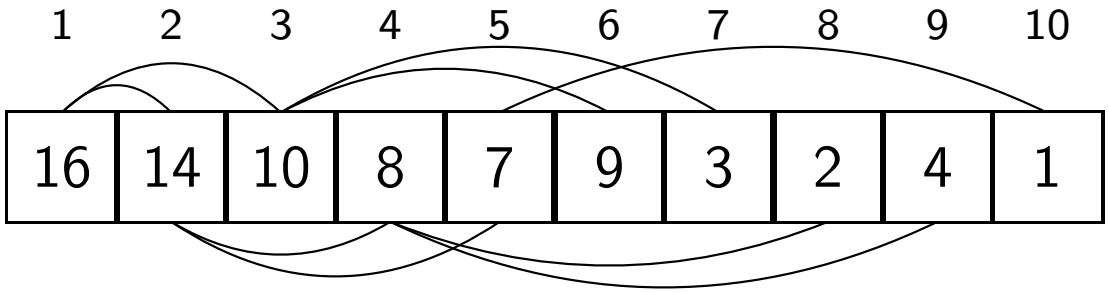
Definition 9

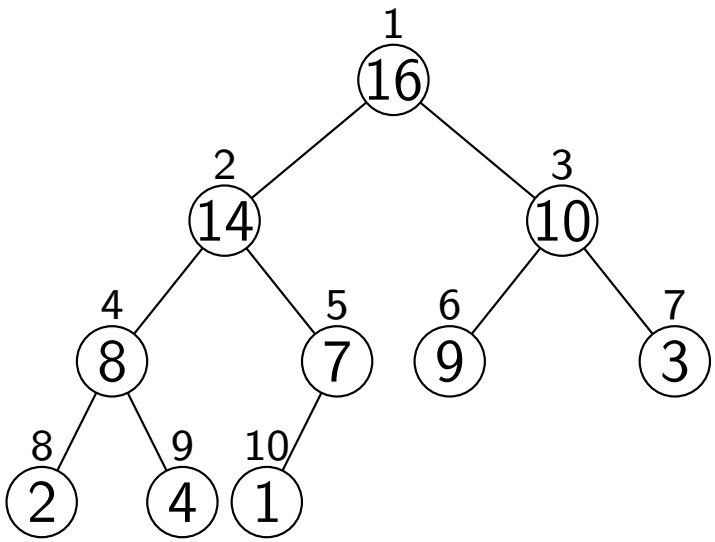
Ein (Max-)Heap ist ein Feld $a[1 \ldots n]$ mit den Eigenschaften:

- $a[i] \geq a[2i]$ für alle $i \geq 1$ mit $2i \leq n$
- $a[i] \geq a[2i+1]$ für alle $i \geq 1$ mit $2i+1 \leq n$

Heaps

Beispiel:



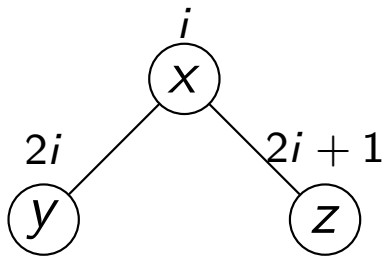


Einsinkprozess

Als erstes wollen wir die Einträge eines Arrays $a[1, \ldots, n]$ so permutieren, dass danach die Heap-Bedingung erfüllt ist.

Angenommen, das Subarray $a[i+1, \ldots, n]$ erfüllt bereits die Heap-Bedingung.

Um die Heap-Bedingung auch für i zu erzwingen, lassen wir $a[i]$ einsinken:



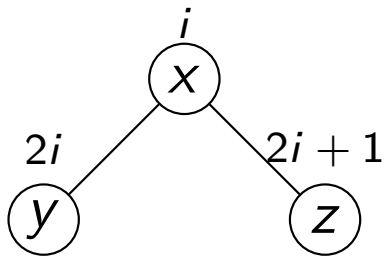
Mit 2 Vergleichen können wir $\max\{x, y, z\}$ bestimmen.

Einsinkprozess

Als erstes wollen wir die Einträge eines Arrays $a[1, \ldots, n]$ so permutieren, dass danach die Heap-Bedingung erfüllt ist.

Angenommen, das Subarray $a[i+1, \ldots, n]$ erfüllt bereits die Heap-Bedingung.

Um die Heap-Bedingung auch für i zu erzwingen, lassen wir $a[i]$ einsinken:



Mit 2 Vergleichen können wir $\max\{x, y, z\}$ bestimmen.

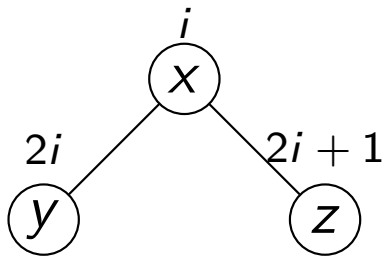
Ist x das Max., so stoppt der Einsinkprozess.

Einsinkprozess

Als erstes wollen wir die Einträge eines Arrays $a[1, \ldots, n]$ so permutieren, dass danach die Heap-Bedingung erfüllt ist.

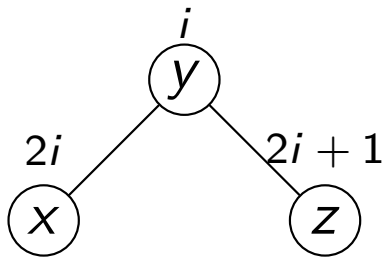
Angenommen, das Subarray $a[i+1, \ldots, n]$ erfüllt bereits die Heap-Bedingung.

Um die Heap-Bedingung auch für i zu erzwingen, lassen wir $a[i]$ einsinken:



Mit 2 Vergleichen können wir $\max\{x, y, z\}$ bestimmen.

Ist y das Max., so vertauschen wir x und y und machen bei $2i$ weiter.

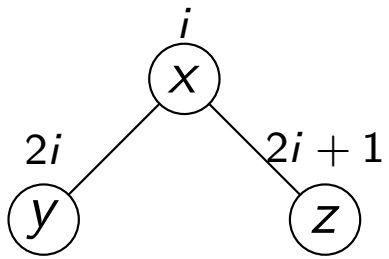


Einsinkprozess

Als erstes wollen wir die Einträge eines Arrays $a[1, \ldots, n]$ so permutieren, dass danach die Heap-Bedingung erfüllt ist.

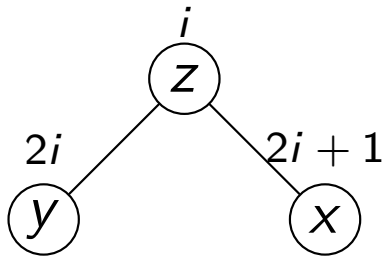
Angenommen, das Subarray $a[i+1, \ldots, n]$ erfüllt bereits die Heap-Bedingung.

Um die Heap-Bedingung auch für i zu erzwingen, lassen wir $a[i]$ einsinken:



Mit 2 Vergleichen können wir $\max\{x, y, z\}$ bestimmen.

Ist z das Max., so vertauschen wir x und z und machen bei $2i+1$ weiter.



Reheap

Algorithmus Reheap

```
procedure reheap(i, n: integer)                    (* i ist die Wurzel *)
var m: integer;
begin
  if i ≤ n/2 then
    m := max{a[i], a[2i], a[2i + 1]};              (* 2 Vergleiche! *)
    if (m ≠ a[i]) ∧ (m = a[2i]) then
      swap(i, 2i);                                 (* vertausche x, y *)
      reheap(2i, n)
    elsif (m ≠ a[i]) ∧ (m = a[2i + 1]) then
      swap(i, 2i + 1);                             (* vertausche x, z *)
      reheap(2i + 1, n)
    endif
  endif
endprocedure
```

Heap-Aufbau

Algorithmus Build Heap

procedure build-heap(n: **integer**)
begin
  for $i := \lfloor \frac{n}{2} \rfloor$ **downto** 1 **do**
    reheap(i, n)
  endfor
endprocedure

Invariante: Vor dem Aufruf von reheap(i, n) erfüllt das Subarray $a[i+1, \ldots, n]$ die Heap-Bedingung.

Für $i = \lfloor \frac{n}{2} \rfloor$ ist dies trivialerweise richtig.

Angenommen die Invariante gilt für i.

Die Heap-Bedingung ist daher nur für i eventuell verletzt.

Nach dem Einsinken von $a[i]$ gilt dann die Heap-Bedingung auch für i.

Zeitanalyse für Heap-Aufbau

Theorem 10

Built-heap läuft in Zeit $\mathcal{O}(n)$.

Beweis: Einsinken von $a[i]$ kostet $2 \cdot$ (Höhe des Teilbaums unter $a[i]$) viele Vergleiche.

Wir führen die Analyse für $n = 2^k - 1$ durch.

Dann haben wir eine vollen Binärbaum der Tiefe $k - 1$ vorliegen.

Es gibt dann

- 2^0 Bäume der Höhe $k - 1$,
- 2^1 Bäume der Höhe $k - 2$,

  $\vdots$
- 2^i Bäume der Höhe $k - 1 - i$,

  $\vdots$
- 2^{k-1} Bäume der Höhe 0.

Zeitanalyse für Heap-Aufbau

Daher sind zum Heapaufbau maximal

$$\begin{aligned} 2 \cdot \sum_{i=0}^{k-1} 2^i (k-1-i) &= 2 \cdot \sum_{i=0}^{k-1} 2^{k-1-i} i \\ &= 2^k \cdot \sum_{i=0}^{k-1} i \cdot 2^{-i} \\ &\leq (n+1) \cdot \sum_{i \geq 0} i \cdot 2^{-i} \end{aligned}$$

viele Vergleiche nötig.

Behauptung: $\sum_{j \geq 0} j \cdot 2^{-j} = 2$

Beweise der Behauptung: Für $|z| < 1$ gilt:

$$\sum_{j \geq 0} z^j = \frac{1}{1-z}.$$

Zeitanalyse für Heap-Aufbau

Ableiten ergibt

$$\sum_{j \geq 0} j \cdot z^{j-1} = \frac{1}{(1-z)^2},$$

und damit

$$\sum_{j \geq 0} j \cdot z^{j} = \frac{z}{(1-z)^2}.$$

Einsetzen von $z = 1/2$ ergibt

$$\sum_{j \geq 0} j \cdot 2^{-j} = 2$$

Standard Heapsort (W. J. Williams, 1964)

Algorithmus Heapsort

```
procedure heapsort(n: integer)
begin
   build-heap(n)
   for i := n downto 2 do
      swap(1, i);
      reheap(1, i − 1)
   endfor
endprocedure
```

Theorem 11

Standard Heapsort sortiert ein Array mit n Elementen und erfordert $2n \log_2 n + \mathcal{O}(n)$ *Vergleiche.*

Standard Heapsort

Beweis:

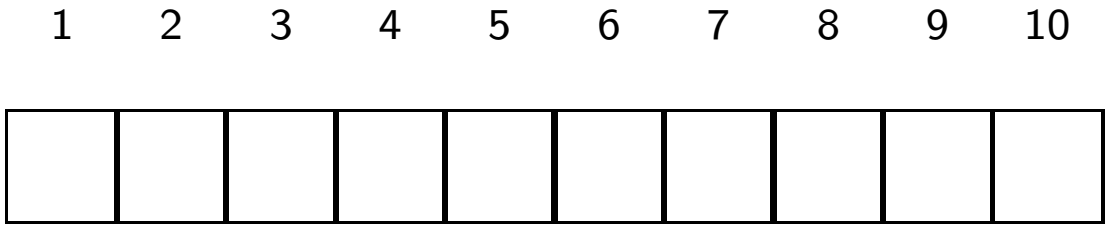
Korrektheit: Nach build-heap(n) ist $a[1]$ das maximale Element des Arrays.

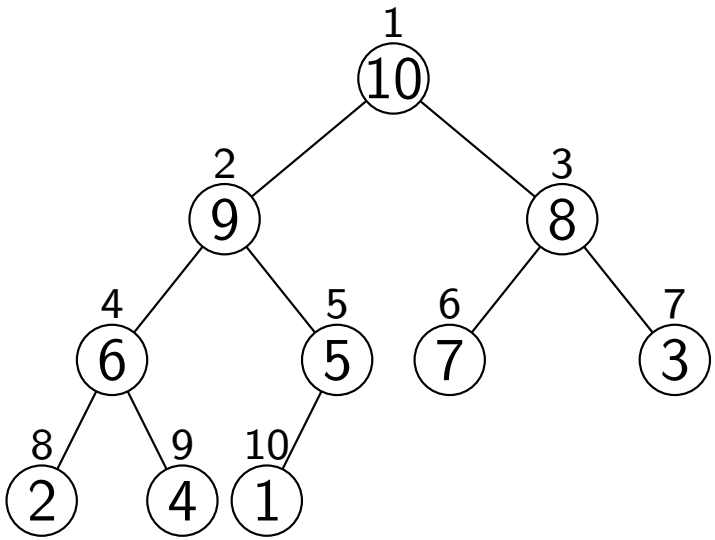
Dieses wird mittels swap$(1, n)$ an seine korrekte Position (n) transportiert.

Nach Induktion wird während der restlichen Laufzeit das Subarray $a[1, \ldots, n-1]$ sortiert.

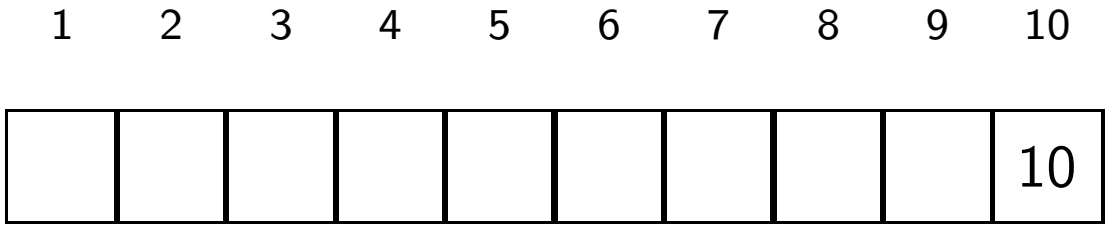
Laufzeit: Der Heap-Aufbau erfordert $\mathcal{O}(n)$ und der Abbau durch Einsinken $2n \log_2 n$ Vergleiche. □

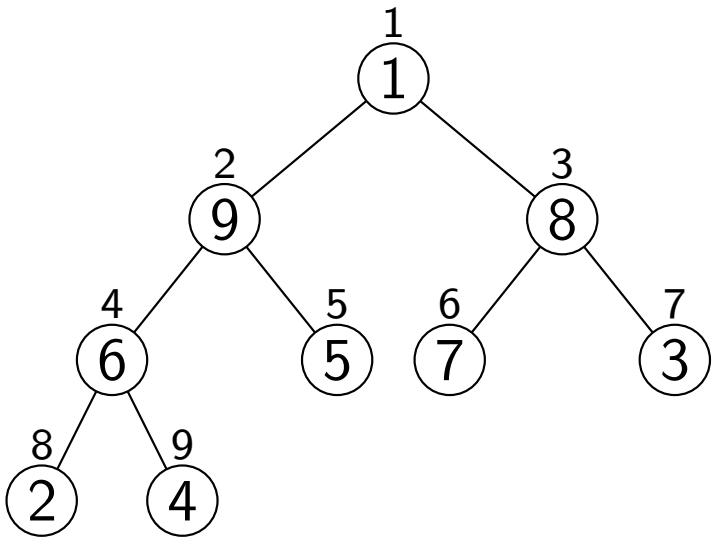
Beispiel für Standard Heapsort



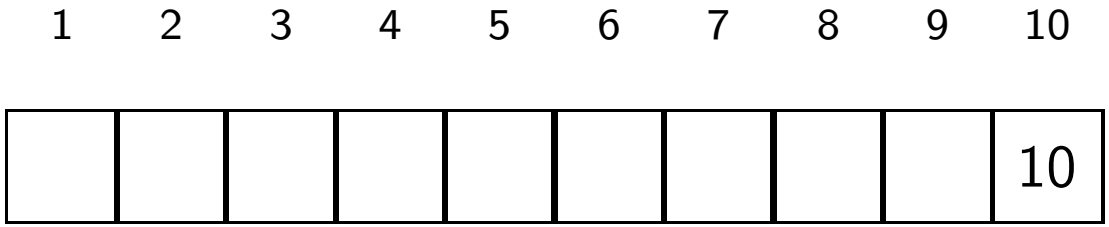


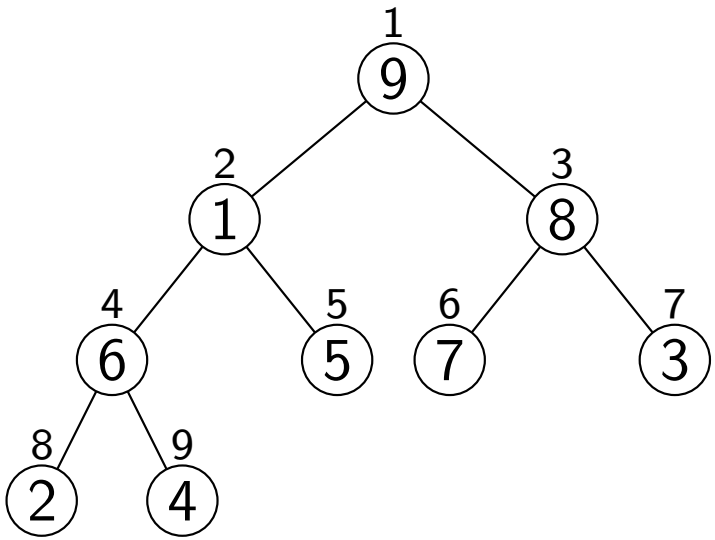
Beispiel für Standard Heapsort



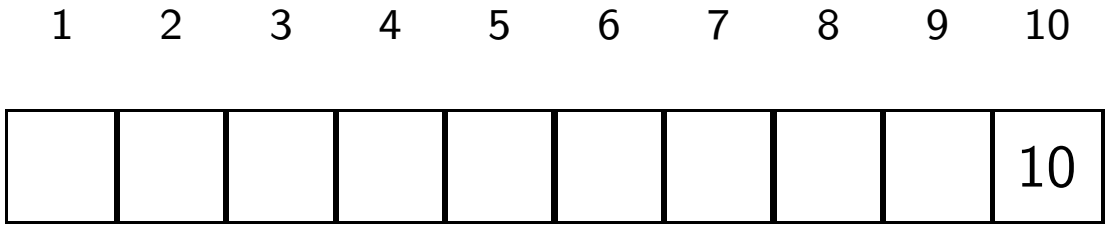


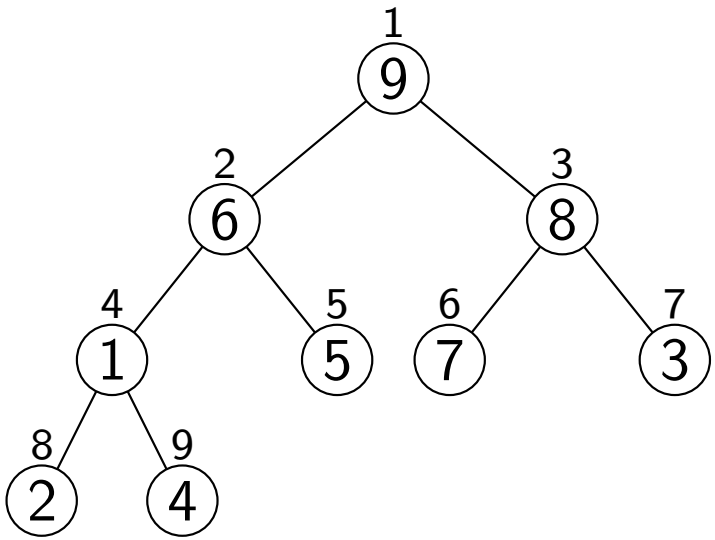
Beispiel für Standard Heapsort



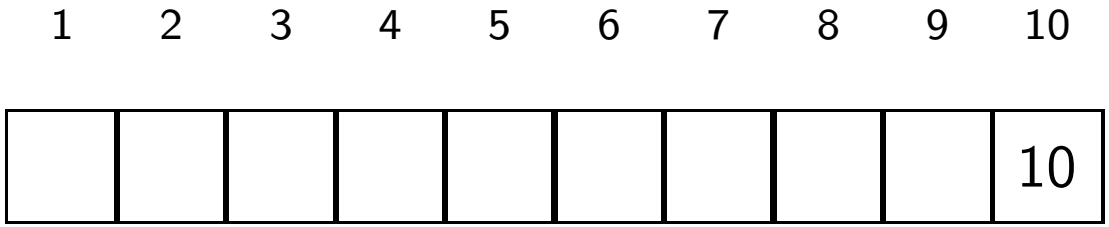


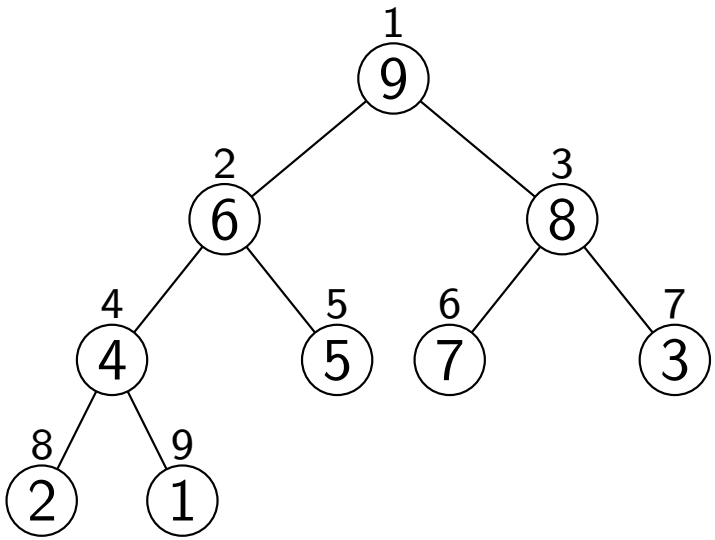
Beispiel für Standard Heapsort



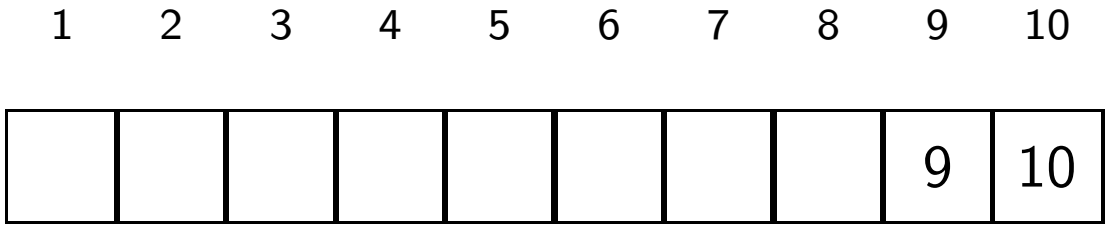


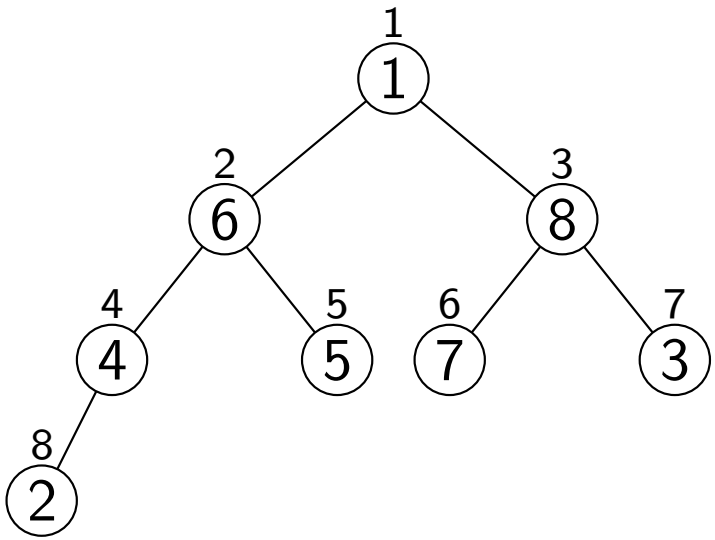
Beispiel für Standard Heapsort



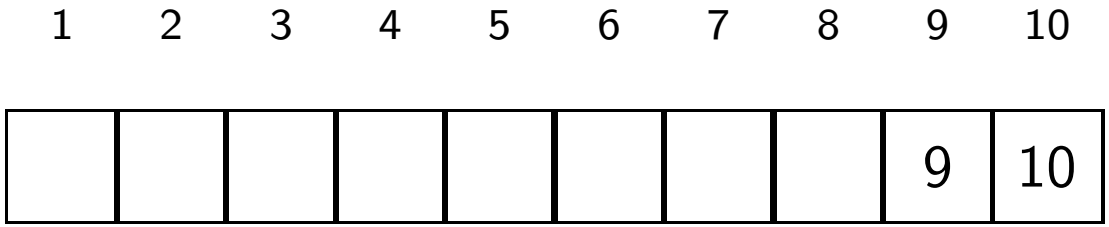


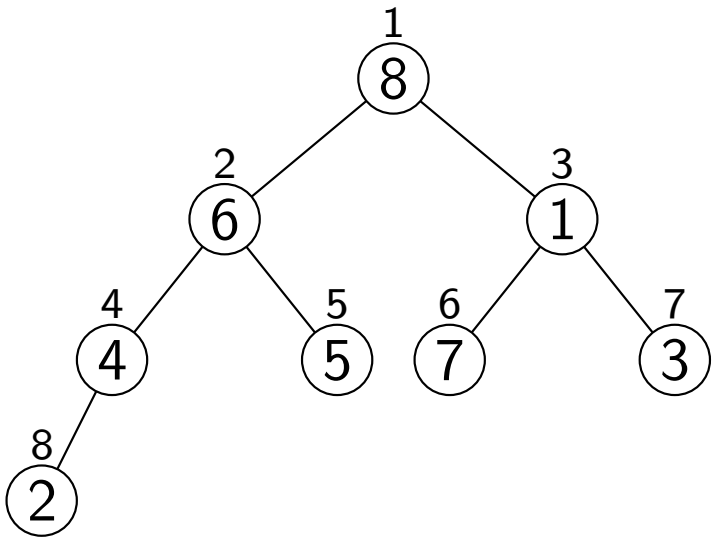
Beispiel für Standard Heapsort



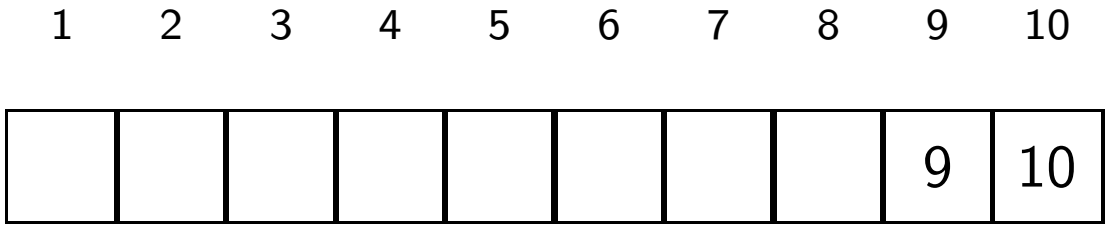


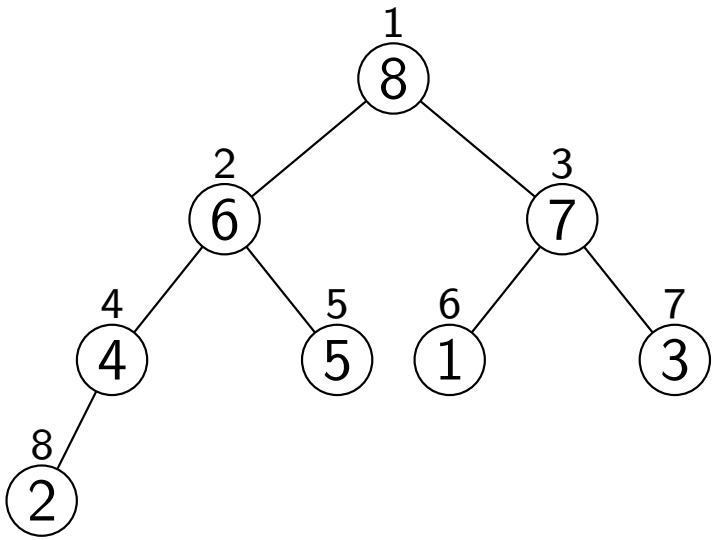
Beispiel für Standard Heapsort



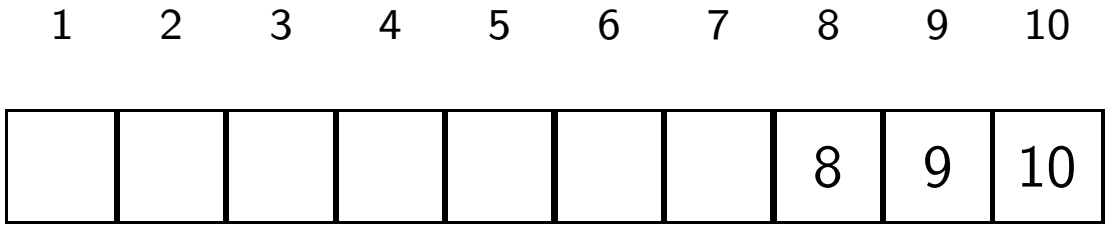


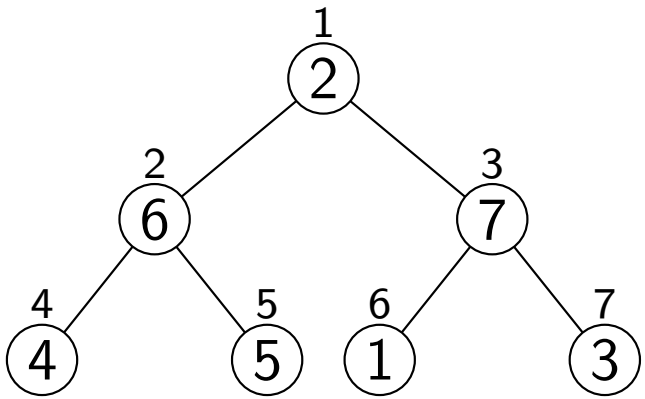
Beispiel für Standard Heapsort



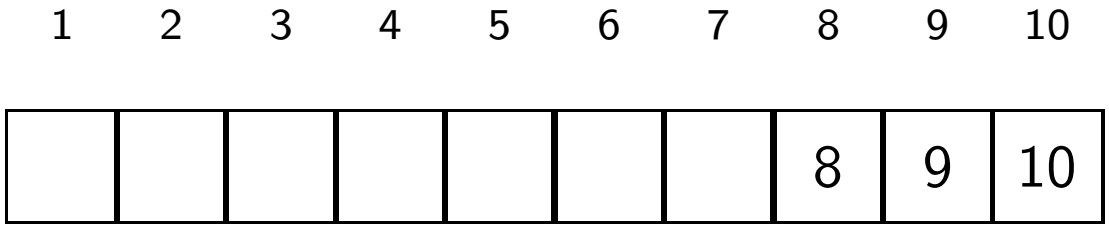


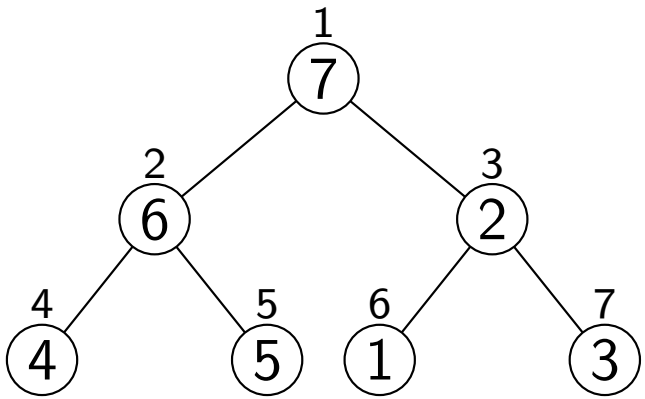
Beispiel für Standard Heapsort



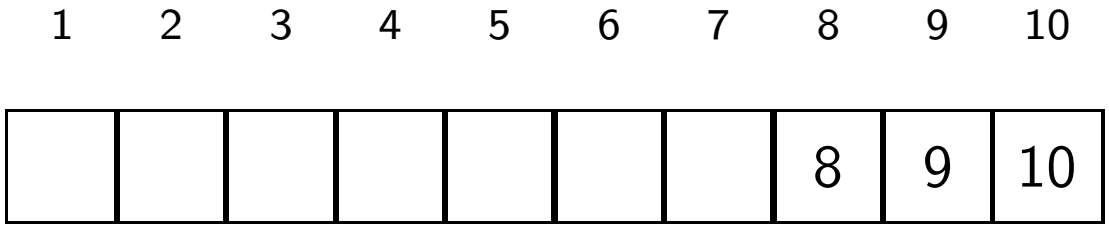


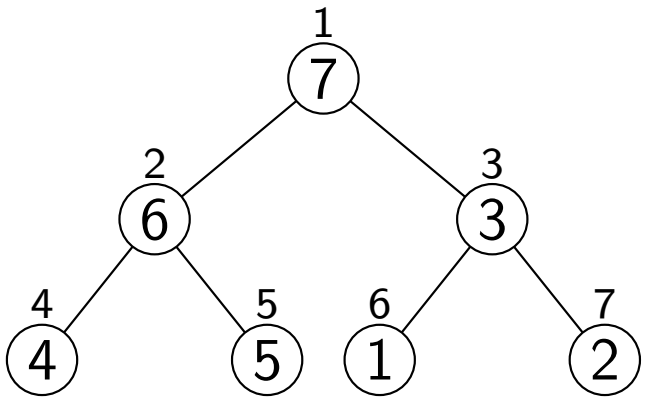
Beispiel für Standard Heapsort



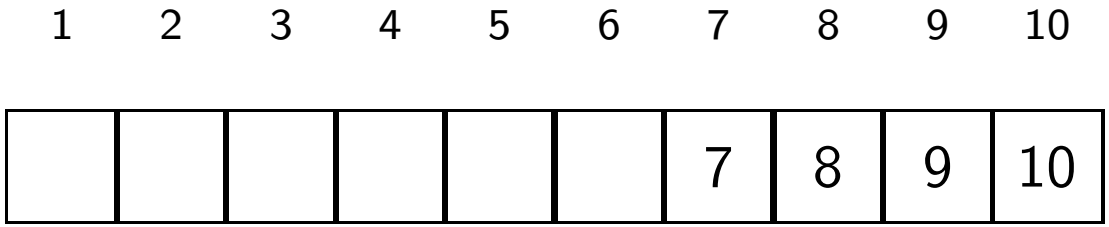


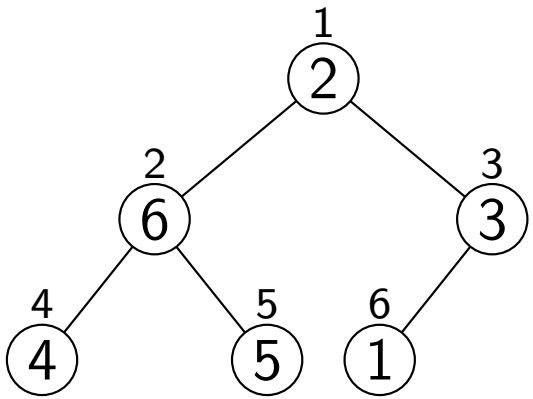
Beispiel für Standard Heapsort



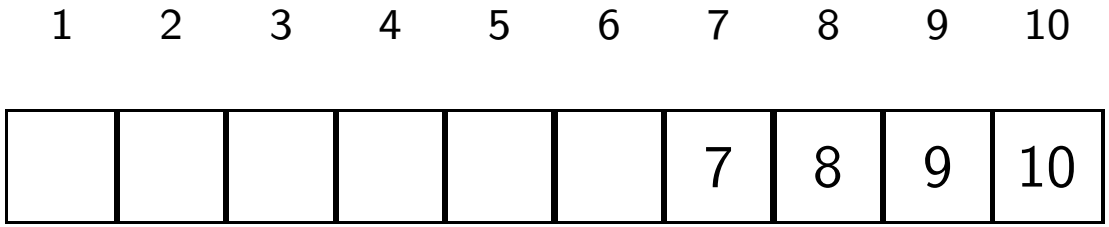


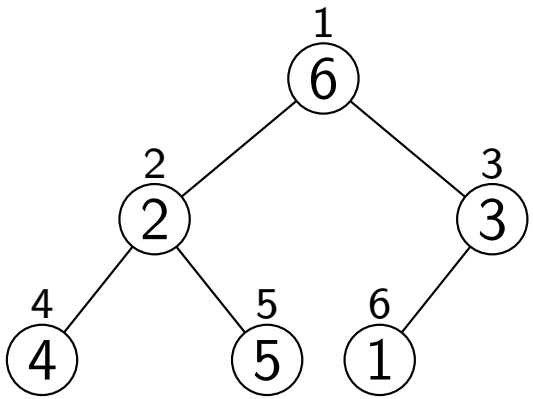
Beispiel für Standard Heapsort



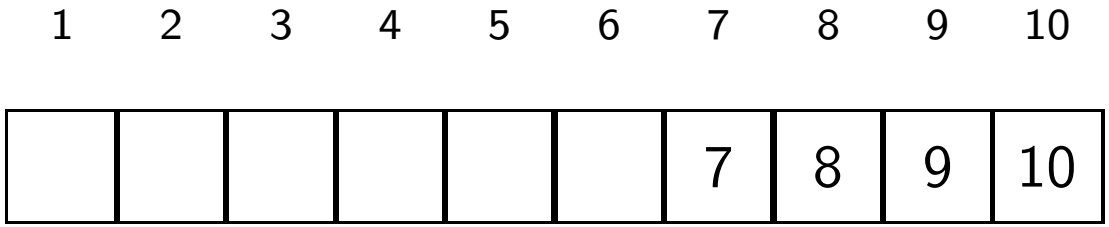


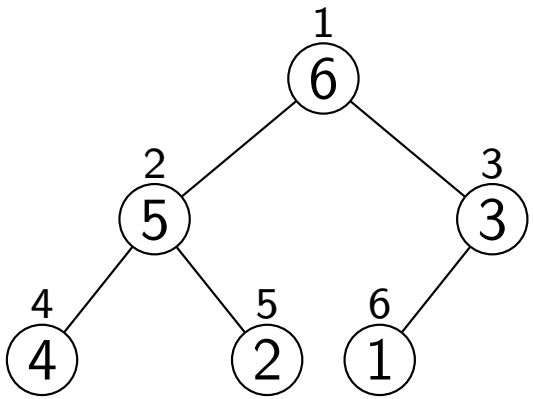
Beispiel für Standard Heapsort



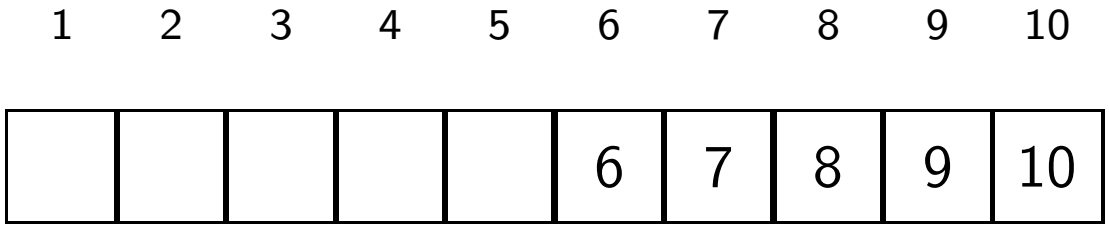


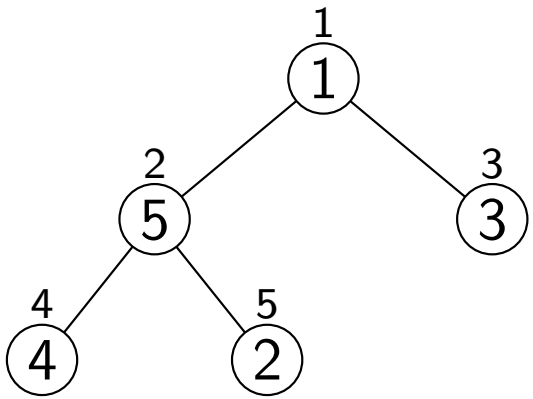
Beispiel für Standard Heapsort



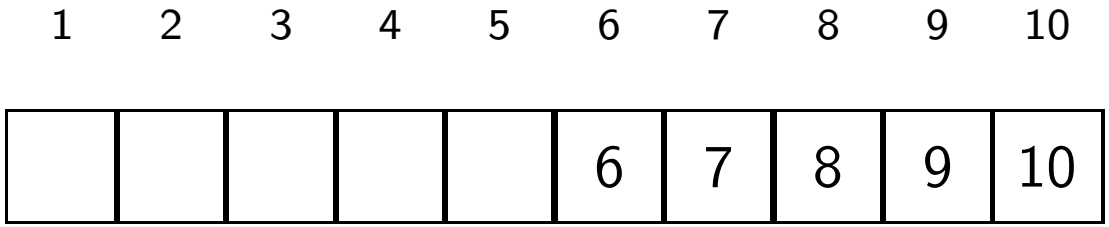


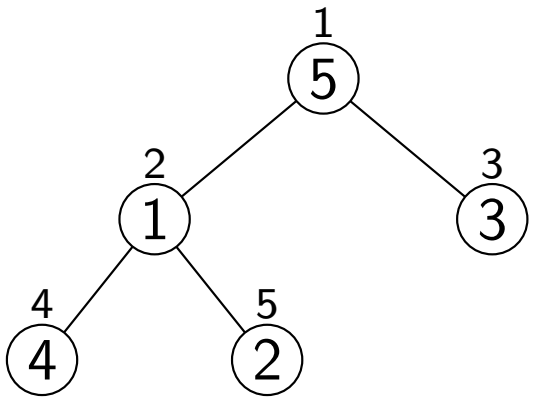
Beispiel für Standard Heapsort



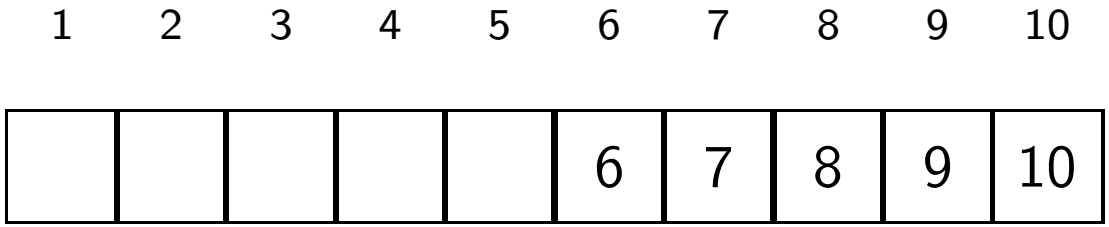


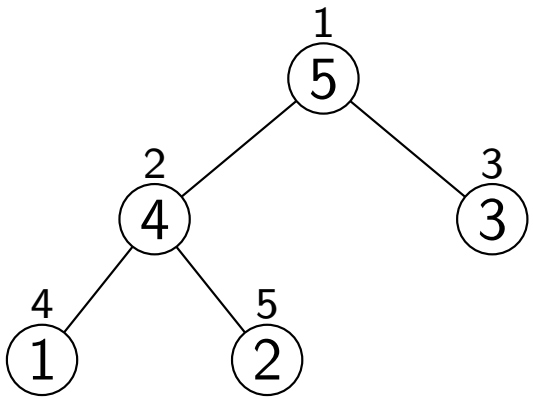
Beispiel für Standard Heapsort



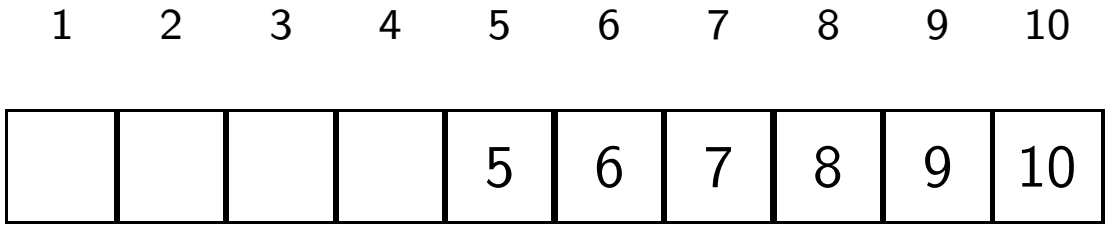


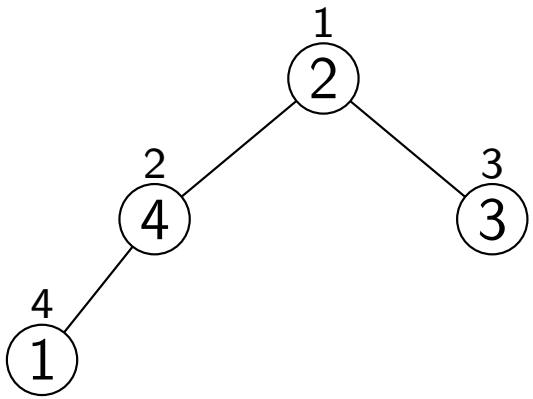
Beispiel für Standard Heapsort



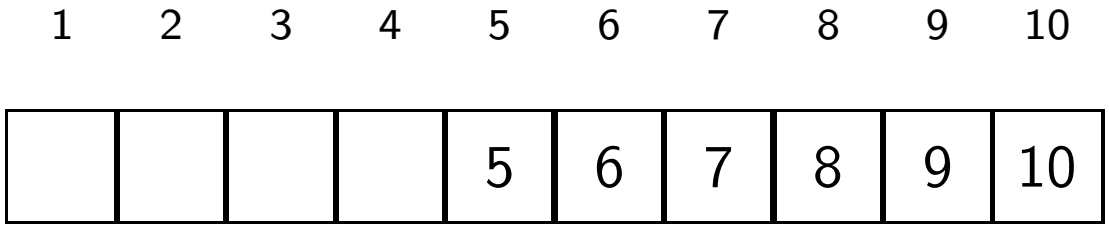


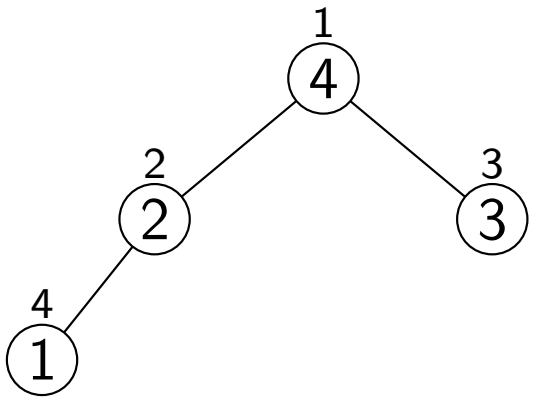
Beispiel für Standard Heapsort



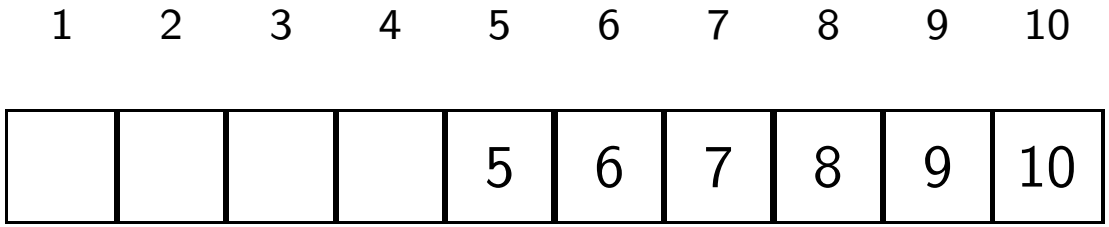


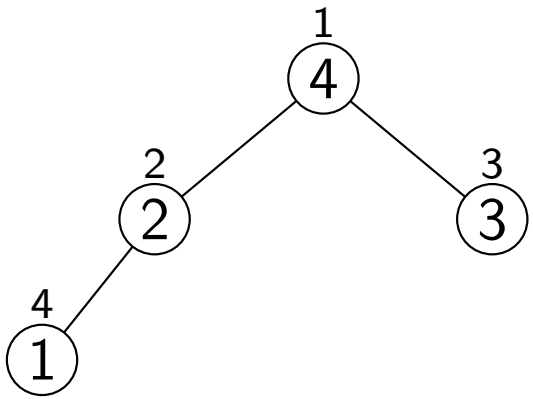
Beispiel für Standard Heapsort



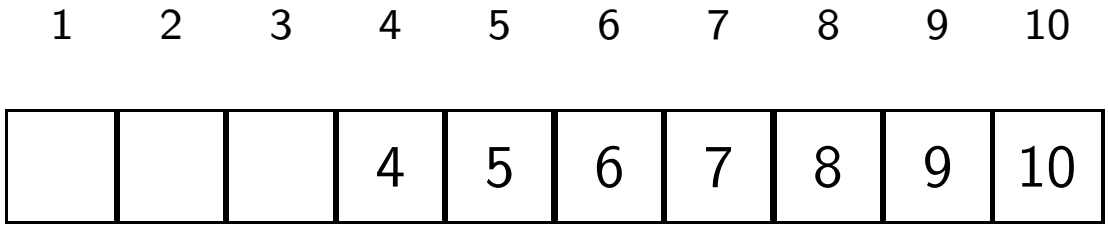


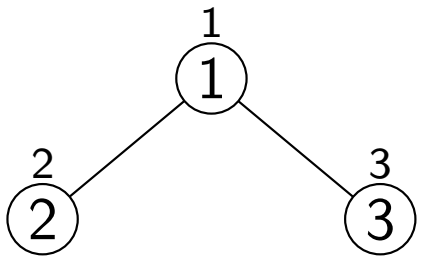
Beispiel für Standard Heapsort



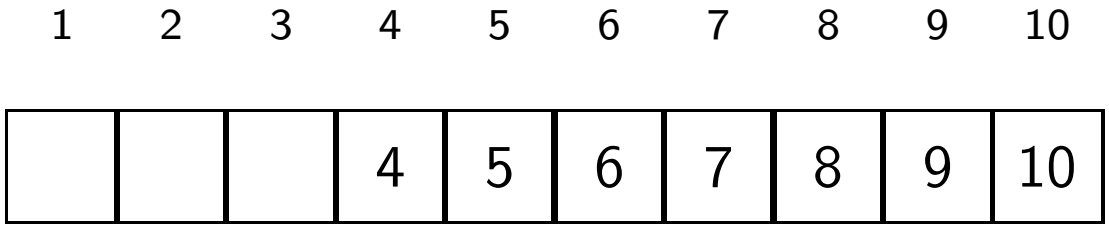


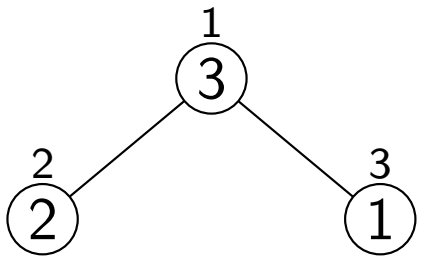
Beispiel für Standard Heapsort



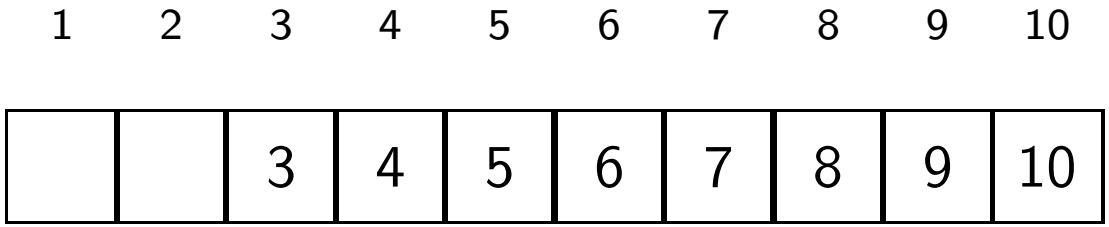


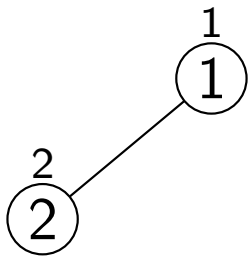
Beispiel für Standard Heapsort



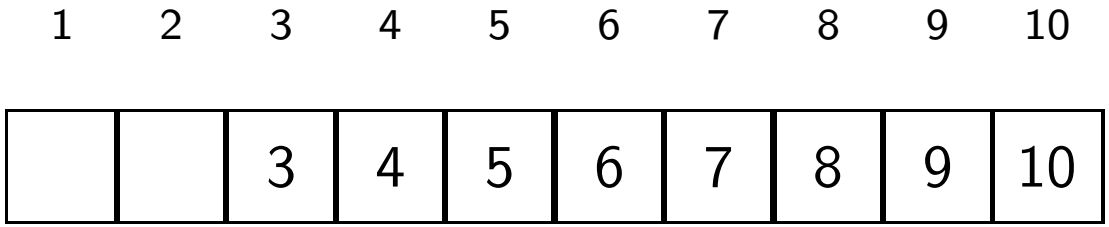


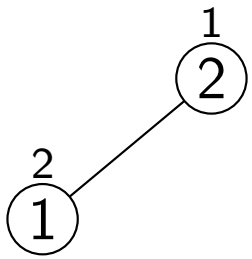
Beispiel für Standard Heapsort



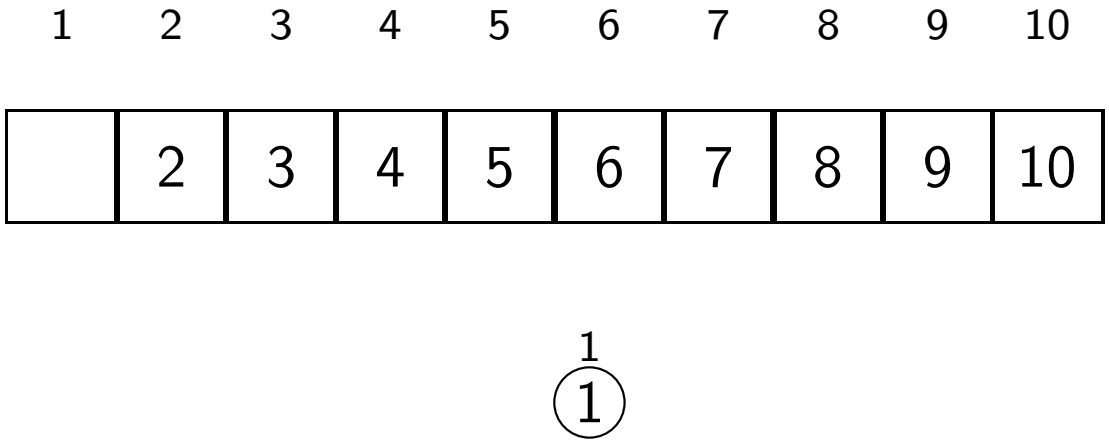


Beispiel für Standard Heapsort





Beispiel für Standard Heapsort



Beispiel für Standard Heapsort

1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10

Bottom-Up-Heapsort

Bemerkung: Eine Analyse der Durchschnittskomplexität von Heapsort ergibt einen mittlerer Aufwand von $2n \log_2 n$ Vergleichen (kommt noch). Damit ist Standard-Heapsort zu Quicksort nicht konkurrenzfähig.

Bottom-Up-Heapsort benötigt wesentlich weniger Vergleiche.

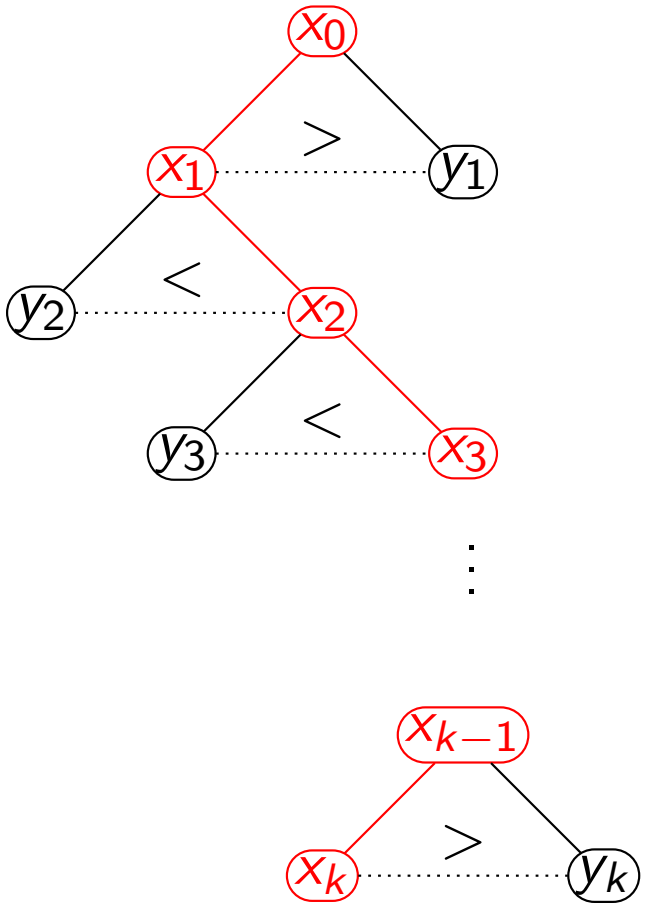
Nach dem Vertauschen swap$(1, i)$ wird zuerst der potentielle Einsinkpfad des Elementes $a[i]$ bestimmt.

Es wird der Weg verfolgt, der immer zum größeren der beiden Nachfolger führt (Kosten: nur $\log n$ statt $2 \log_2 n$ Vergleiche).

Da erwartungsgemäß $a[i]$ tief einsinken wird, bietet sich anschließend eine bottom-up Bestimmung der tatsächlichen Position auf dem Einsinkpfad an.

Hoffnung: Die bottom-up Bestimmung erfordert insgesamt nur $\mathcal{O}(n)$ Vergleiche.

Der Einsinkpfad



Das Einsinken wird auf dem Pfad $[x_0, x_1, x_2, \ldots, x_{k-1}, x_k]$ geschehen, welcher mittels $\log_2 n$ Vergleiche bestimmt werden kann.

Einsortieren auf dem Einsinkpfad

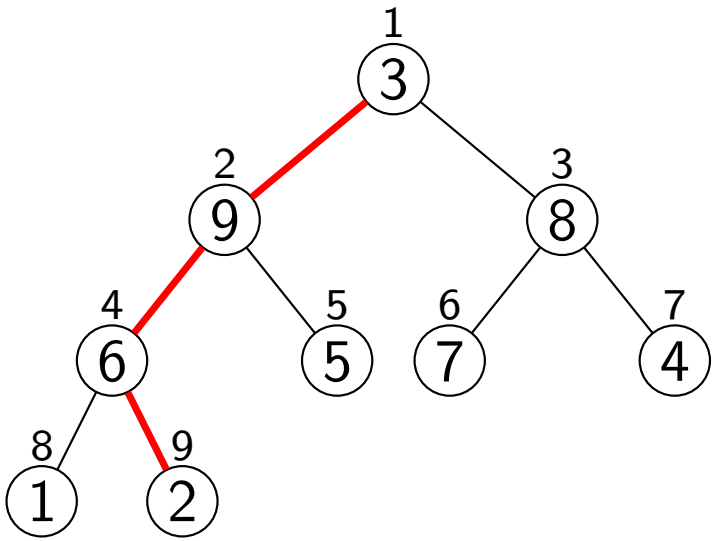
Wir bestimmen jetzt vom Blatt des Einsinkpfades aus (also bottom-up) die tatsächliche Position p auf dem Einsinkpfad.

Ist diese Position p gefunden, so werden die Elemente $x_0, \ldots, x_p$ zyklisch vertauscht (x_0 geht an die Stelle von x_p, und $x_1, \ldots, x_p$ rutschen hoch).

Einsortieren auf dem Einsinkpfad

Wir bestimmen jetzt vom Blatt des Einsinkpfades aus (also bottom-up) die tatsächliche Position p auf dem Einsinkpfad.

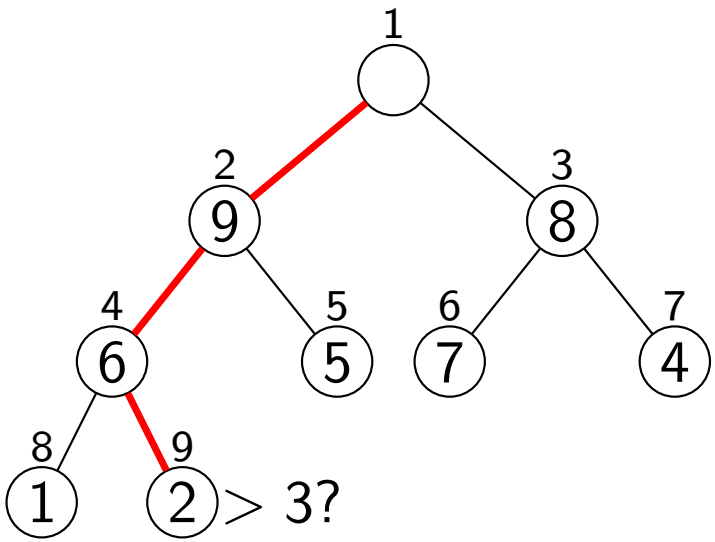
Ist diese Position p gefunden, so werden die Elemente $x_0, \ldots, x_p$ zyklisch vertauscht (x_0 geht an die Stelle von x_p, und $x_1, \ldots, x_p$ rutschen hoch).



Einsortieren auf dem Einsinkpfad

Wir bestimmen jetzt vom Blatt des Einsinkpfades aus (also bottom-up) die tatsächliche Position p auf dem Einsinkpfad.

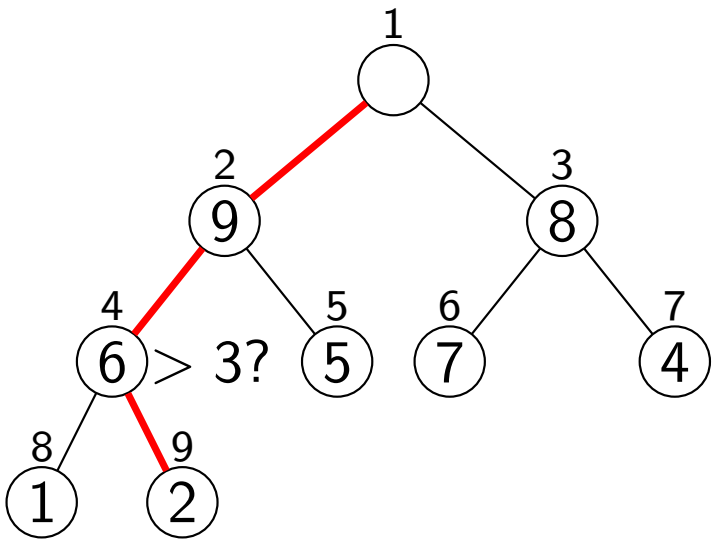
Ist diese Position p gefunden, so werden die Elemente $x_0, \ldots, x_p$ zyklisch vertauscht (x_0 geht an die Stelle von x_p, und $x_1, \ldots, x_p$ rutschen hoch).



Einsortieren auf dem Einsinkpfad

Wir bestimmen jetzt vom Blatt des Einsinkpfades aus (also bottom-up) die tatsächliche Position p auf dem Einsinkpfad.

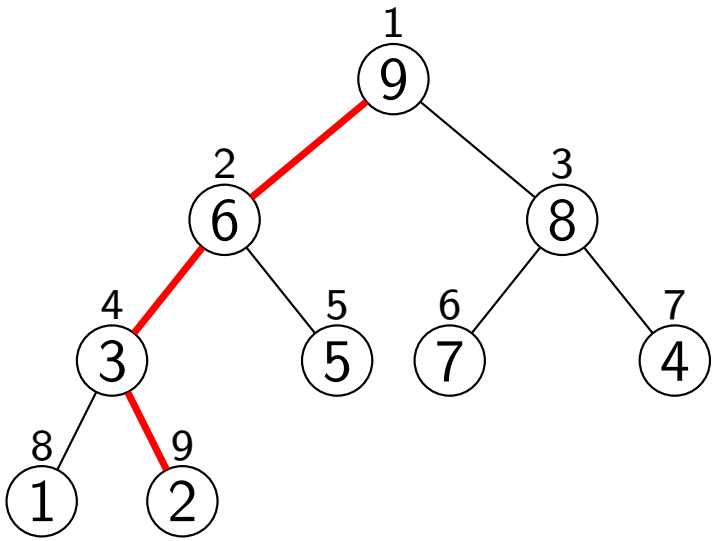
Ist diese Position p gefunden, so werden die Elemente $x_0, \ldots, x_p$ zyklisch vertauscht (x_0 geht an die Stelle von x_p, und $x_1, \ldots, x_p$ rutschen hoch).



Einsortieren auf dem Einsinkpfad

Wir bestimmen jetzt vom Blatt des Einsinkpfades aus (also bottom-up) die tatsächliche Position p auf dem Einsinkpfad.

Ist diese Position p gefunden, so werden die Elemente $x_0, \ldots, x_p$ zyklisch vertauscht (x_0 geht an die Stelle von x_p, und $x_1, \ldots, x_p$ rutschen hoch).



Average Analyse von Heapsort

Unser Ziel ist der Beweis des folgenden Satzes:

Theorem 12

Standard-Heapsort macht auf einem Anteil von mindestens $1 - 2^{-(n-1)}$ *vieler Eingabepermutationen mindestens* $2n \log_2(n) - \mathcal{O}(n)$ *viele Vergleiche.*
Bottomup-Heapsort macht auf einem Anteil von mindestens $1 - 2^{-(n-1)}$ *vieler Eingabepermutationen höchstens* $n \log_2(n) + \mathcal{O}(n)$ *viele Vergleiche.*

Beweis: Durch ein informationstheoretisches Argument.

Ein Sortieralgorithmus produziert aus einer Permutation von $[1, \ldots, n]$ die sortierte Liste $[1, \ldots, n]$.

Wir können daher die Eingabepermutation dadurch spezifizieren, indem wir zusätzliche Information in Form eines $\{0, 1\}$-Strings angeben, welche es erlaubt, den Algorithmus von der Ausgabe $[1, \ldots, n]$ rückwärts laufen zu lassen.

Average Analyse von Heapsort

Im Fall von Standard-Heapsort: Wir merken uns die Einsinkpfade, d.h. jedes mal, wenn wir ein Element mit dem linken (bzw. rechten) Kind vertauschen, schreiben wir eine 0 bzw. 1 auf. Dadurch wird Heapsort reversibel.

Aber: Wir müssen aufeinanderfolgende Einsinkpfade ($= \{0,1\}$-Strings) voneinander trennen.

1. Alternative: Wir kodieren ein Wort $w = a_1 a_2 \cdots a_{t-1} a_t \in \{0,1\}^*$ durch

$$c_1(w) = a_1 0 a_2 0 \cdots a_{t-1} 0 a_t 1.$$

Beachte: $|c_1(w)| = 2|w|$.

2. Alternative: Wir kodieren ein Wort $w = a_1 a_2 \cdots a_{t-1} a_t \in \{0,1\}^*$ duch

$$c_2(w) = c_1(\text{Binärdarstellung von } t) a_1 \cdots a_t$$

Dann gilt $|c_2(w)| = |w| + 2\log_2(|w|)$.

Average Analyse von Heapsort

Beachte: $c_2(\varepsilon) = 01$, da $0 =$ Binärdarstellung der Zahl 0.

Wir kodieren den Einsinkpfad $w = a_1 a_2 \cdots a_t \in \{0,1\}^*$ durch

$$c_2'(w) = c_1(\text{Binärdarstellung von } \log_2(n) - t)a_1 \cdots a_t.$$

Beachte: $t \leq \log_2(n)$, denn jeder Einsinkpfad hat Länge höchstens $\log_2 n$.

Unser Argument, welches gezeigt hat, dass der Heapaufbau nur $\mathcal{O}(n)$ viele Vergleiche benötigt, zeigt: In Phase 1 wird ein $\{0,1\}$-String der Länge $\mathcal{O}(n)$ erzeugt.

Wir analysieren nun die Länge des in Phase 2 erzeugten $\{0,1\}$-Strings.

Average Analyse von Heapsort

Seien $t_1, \ldots, t_n$ die Längen der Einsinkpfade während Phase 2.

Also wird in Phase 2 ein $\{0,1\}$-String der Länge

$$\sum_{i=1}^{n}(t_i + 2\log_2(\log_2(n) - t_i)) = \sum_{i=1}^{n} t_i \; + \; 2\sum_{i=1}^{n} \log_2(\log_2(n) - t_i))$$

generiert.

Definiere den Mittelwert

$$\bar{t} = \frac{\sum_{i=1}^{n} t_i}{n}.$$

Die Funktion f mit $f(x) = \log_2(\log_2(n) - x)$ ist konkav auf ihrem Definitionsbereich $(-\infty, \log_2(n))$.

Dann folgt aus Jensens Ungleichung (Folie 4):

$$\log_2(\log_2(n) - \bar{t}) \geq \sum_{i=1}^{n} \frac{1}{n} \cdot \log_2(\log_2(n) - t_i)).$$

Average Analyse von Heapsort

Also gilt:

$$\sum_{i=1}^{n} t_i \;+\; 2\sum_{i=1}^{n} \log_2(\log_2(n) - t_i)) \quad \leq \quad n\bar{t} + 2n\log_2(\log_2(n) - \bar{t}).$$

Insgesamt erhalten wir: Die Eingabepermutation σ von $[1, \ldots, n]$ kann durch einen $\{0,1\}$-String der Länge

$$I(\sigma) \leq cn + n\bar{t} + 2n\log_2(\log_2(n) - \bar{t})$$

kodiert werden, wobei c eine Konstante (für Phase 1) ist.

Lemma 7 impliziert, dass für mindestens $(1 - 2^{-n+1})n!$ viele Eingabepermutationen gilt:

$$cn + n\bar{t} + 2n\log_2(\log_2(n) - \bar{t}) \geq I(\sigma) \geq \log_2(n!) - n \geq n\log_2(n) - 2,443n$$

Umgeformt mit $d = 2,443 + c$:

$$\bar{t} \geq \log_2(n) - 2\log_2(\log_2(n) - \bar{t}) - d. \qquad (1)$$

Average Analyse von Heapsort

Wegen $\bar{t} \geq 0$ folgt:

$$\bar{t} \geq \log_2(n) - 2\log_2(\log_2(n)) - d. \tag{2}$$

Aus (1) und (2) folgt die bessere Abschätzung

$$\bar{t} \geq \log_2(n) - 2\log_2(2\log_2(\log_2(n)) + d) - d. \tag{3}$$

Diese Abschätzung können wir wieder in (1) einsetzen, ...

Allgemein erhalten wir für alle $i \geq 1$:

$$\bar{t} \geq \log_2(n) - \alpha_i - d,$$

wobei $\alpha_1 = 2\log_2(\log_2(n))$ und $\alpha_{i+1} = 2\log_2(\alpha_i + d)$.
(Beweis durch Induktion über $i \geq 1$)

Average Analyse von Heapsort

Nun gilt für alle $x \geq \max\{10, d\}$:

$$2\log_2(x+d) \leq 2\log_2(2x) = 2\log_2(x) + 2 \leq 0,9 \cdot x.$$

Solange also $\alpha_i \geq \max\{10, d\}$ gilt, haben wir $\alpha_{i+1} \leq 0,9 \cdot \alpha_i$.

Daher existiert eine Konstante α mit

$$\bar{t} \geq \log_2(n) - \alpha - d. \tag{4}$$

Also gilt für mindestens $(1 - 2^{-n+1})n!$ viele Eingabepermutationen

$$\sum_{i=1}^{n} t_i \geq n\log_2 n - \mathcal{O}(n).$$

Average Analyse von Heapsort

Die Aussage von Theorem 12 für Standard-Heapsort folgt nun einfach :

In Phase 2 macht Standard-Heapsort $2\sum_{i=1}^{n} t_i$ viele Vergleiche.

Also macht Standard-Heapsort für mindestens $(1-2^{-n+1})n!$ viele Eingabepermutationen mindestens $2n\log_2 n - \mathcal{O}(n)$ viele Vergleiche.

Nun zu Bottomup-Heapsort: Dieser macht in Phase 2 höchstens

$$n\log_2(n) + \sum_{i=1}^{n}(\log_2(n) - t_i) = 2n\log_2(n) - \sum_{i=1}^{n} t_i$$

viele Vergleiche.

Also macht Bottomup-Heapsort für mindestens $(1-2^{-n+1})n!$ viele Eingabepermutationen höchstens

$$\mathcal{O}(n) + 2n\log_2(n) - \sum_{i=1}^{n} t_i \leq n\log_2(n) + \mathcal{O}(n)$$

viele Vergleiche. □

Variante nach Svante Carlsson, 1986

Es kann gezeigt werden, dass Bottom-up Heapsort im schlechtesten Fall höchstens $1.5n \log n + \mathcal{O}(n)$ viele Vergleiche macht.

Wenn man nach Carlsson auf dem Einsinkpfad eine binäre Suche anwendet, so kommt man auf einen worst-case Aufwand von höchstens $n \log n + \mathcal{O}(n \log \log n)$.

Eine binäre Suche ist aber in der Praxis zu aufwendig und außerdem steigt man in den Pfad i.A. zu hoch ein.

Counting-Sort

Zur Erinnerung: Die $\Omega(n \log n)$-Schranke gilt nur für vergleichsbasierte Sortieralgorithmen.

Wenn wir gewisse Annahmen über die Arrayeinträge machen, können wir in Zeit $\mathcal{O}(n)$ sortieren.

Annahme: Die Arrayelemente $A[1], \ldots A[n]$ sind Zahlen aus dem Bereich $[0, k]$.

Counting-Sort (siehe nächste Folie) sortiert unter dieser Annahme in Zeit $\mathcal{O}(k + n)$.

Gilt also $k \in \mathcal{O}(n)$, so sortiert Counting-Sort in Linearzeit.

Counting-Sort

Algorithmus Counting-Sort

procedure counting-sort(Array $A[1, n]$ mit $A[1], \ldots A[n] \in [0, k]$)
begin
  var Arrays $C[0, k]$, $B[1, n]$
  for $i := 0$ **to** k **do**
    $C[i] := 0$
  for $i := 1$ **to** n **do**
    $C[A[i]] := C[A[i]] + 1$
  for $i := 1$ **to** k **do**
    $C[i] := C[i] + C[i - 1]$
  for n **downto** 1 **do**
    $B[C[A[i]]] := A[i]$;
    $C[A[i]] := C[A[i]] - 1$
endprocedure

Counting-Sort

Nach den ersten drei for-Schleifen ist $C[i]$ die Anzahl der Array-Einträge, die $\leq i$ sind.

Die Anweisung $B[C[A[i]]] := A[i]$ stellt dann das Array-Element $A[i]$ an die richtige Position $C[A[i]]$.

Bemerkung: Counting-Sort ist ein stabiler Sortieralgorithmus.

Dies bedeutet: Gilt $A[i] = A[j]$ für $i < j$, so steht in dem sortierten Array B der Array-Eintrag $A[i]$ links von $A[j]$.

Dies ist relevant, wenn die Array-Einträge neben den Schlüsseln, nach denen sortiert wird, noch andere Informationen enthalten.

Stabilität von Counting-Sort wird auf der nächsten Folie für Radix-Sort benötigt.

Radix-Sort

Wir benutzen nun Counting-Sort, um ein Array $A[1, n]$ zu sortieren, wobei $A[1], \ldots, A[n]$ d-stellige Zahlen zur Basis k sind (wobei die niederwertigste Stelle ganz links ist).

Radix-Sort sortiert solch ein Array in Zeit $\mathcal{O}(d(n + k))$.

Gilt also $d \in \mathcal{O}(1)$ und $k \in \mathcal{O}(n)$ (womit sich Zahlen im Bereich $\mathcal{O}(n^d)$ darstellen lassen), so arbeitet Radix-Sort in Linearzeit.

Algorithmus Radix-Sort

---

procedure radix-sort(Array $A[1, n]$ mit $A[1], \ldots A[n]$)
begin
  for $i := 1$ **to** d **do**
    Sortiere das Array A mittels Counting-Sort nach der i-ten Stelle.
  endfor
endprocedure

Medianberechnung

Gegeben: Array $a[1, \ldots, n]$ von Zahlen und $1 \leq k \leq n$.

Gesucht: k-kleinste Element m, d.h. die Zahl $m \in \{a[i] \mid 1 \leq i \leq n\}$ so, dass

$$|\{i \mid a[i] < m\}| \leq k - 1 \quad \text{und} \quad |\{i \mid a[i] > m\}| \leq n - k$$

Spezialfall: $k = \lceil n/2 \rceil \quad \rightsquigarrow$ Median

Naiver Ansatz:

- Sortiere Array a in Zeit $\mathcal{O}(n \log n)$
- Gebe das k-te Element des sortierten Arrays zurück.

Median der Medianen

Ziel: Berechne k-kleinstes Element in Linearzeit.

Idee: Bestimme ein Pivotelement (wie bei Quicksort) als Median der Mediane aus 5:

- Wir teilen das Feld in Fünferblöcken auf.
- In jedem Block wird der Median bestimmt (mit 6 Vergleichen möglich).
- Wir bestimmen rekursiv den Median p dieses Feldes (mit dem gesamten Algorithmus). Der Wert p wird als Pivotelement im folgenden verwendet.

  Kosten: $T(\frac{n}{5})$.

Quicksortschritt

Mit dem Pivot p partitioniere das Array so, dass für gewisse $m_1 < m_2$ gilt:

$$\begin{array}{lcll} a[i] & < & p & \text{für } 1 \leq i \leq m_1 \\ a[i] & = & p & \text{für } m_1 < i \leq m_2 \\ a[i] & > & p & \text{für } m_2 < i \leq n \end{array}$$

Kosten: maximal n Schritte.

Fallunterscheidung:

1. $k \leq m_1$: Suche das k-te Element rekursiv in $a[1], \ldots, a[m_1]$.
2. $m_1 < k \leq m_2$: Das Ergebnis ist p.
3. $k > m_2$: Suche das $(k - m_2)$-te Element in $a[m_2 + 1], \ldots, a[n]$.

30 – 70 Aufteilung

Die Wahl des Pivots als Median-aus-Fünf ergibt die folgende Ungleichungen für m_1, m_2:

$$\frac{3}{10}n \leq m_2 \qquad \text{und} \qquad m_1 \leq \frac{7}{10}n$$

Damit ergeben sich die Kosten für den Rekursionsschritt als $T(\frac{7n}{10})$.

Zeitanalyse

Sei $T(n)$ die Gesamtzahl der Vergleiche.

Wir erhalten folgende Rekursionsgleichung für $T(n)$:

$$T(n) \leq T\left(\lceil \frac{n}{5} \rceil\right) + T\left(\lceil \frac{7n}{10} \rceil\right) + \mathcal{O}(n)$$

Aus dem Mastertheorem II folgt damit $T(n) \in \mathcal{O}(n)$.

Etwas genauer

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + \frac{6n}{5} + \frac{2n}{5}$$

Hierbei ist:

- $\frac{6n}{5}$ der Aufwand für die Bestimmung der Blockmediane und
- $\frac{2n}{5}$ der Aufwand für den Zerlegungsschritt.

Damit kann nun gezeigt werden, dass damit $T(n) \leq 16n$ gilt:

Mit $\frac{1}{5} + \frac{7}{10} = \frac{9}{10}$ erhalten wir $T(n) \leq T(\frac{9n}{10}) + \frac{8n}{5}$ und damit $T(n) \leq 10 \cdot \frac{8n}{5} = 16n$.

Quickselect

Quickselect ist ein randomisierter Algorithmus zur Medianberechnung:

Algorithmus

---

function quickselect($A[\ell \ldots r]$: **array of integer**, k : **integer**) : **integer**
begin
  if $\ell = r$ **then return** $A[\ell]$
  else
    $p := \text{random}(\ell, r)$;
    $m :=$ partitioniere($A[\ell \ldots r]$, p);
    $k' := (m - \ell + 1)$;
    if $k = k'$ **then return** $A[m]$
    elsif $k < k'$ **then return** quickselect($A[\ell \ldots m-1]$, k)
    else return quickselect($A[m+1 \ldots r]$, $k - k'$)
    endif
  endif
endfunction

Analyse von Quickselect

Es sei $Q(n)$ die mittlere Anzahl von Vergleichen, die Quickselect auf einem Array mit n Elementen macht.

Dann gilt:

$$Q(n) \leq (n-1) + \frac{1}{n} \sum_{i=1}^{n} Q(\max\{i-1, n-i\})$$

Hierbei ist

- $(n-1)$ wiederum die Anzahl der Vergleiche für das Partitionieren und
- $Q(\max\{i-1, n-i\})$ die mittlere Anzahl der Vergleiche für den rekursiven Aufruf auf *einem* der beiden Teilfelder.

Dabei machen wir die Annahme, dass wir stets im größeren Teilfeld suchen, daher $\leq$.

Analyse von Quickselect

Es gilt:

$$\begin{aligned} Q(n) &\leq (n-1) + \frac{1}{n}\sum_{i=1}^{n} Q(\max\{i-1, n-i\}) \\ &= (n-1) + \frac{1}{n}\left(\sum_{i=\lceil \frac{n}{2} \rceil}^{n-1} Q(i) + \sum_{i=\lfloor \frac{n}{2} \rfloor}^{n-1} Q(i)\right) \end{aligned}$$

Behauptung: $Q(n) \leq 4 \cdot n$:

Beweis mit Induktion: OK für $n = 1$.

Sei $n \geq 2$ und $Q(i) \leq 4 \cdot i$ für alle $i < n$.

Analyse von Quickselect

1. Fall: n gerade.

Dann gilt:

$$\begin{aligned} Q(n) &\leq (n-1) + \frac{2}{n}\sum_{i=\frac{n}{2}}^{n-1} Q(i) \\ &= (n-1) + \frac{8}{n}\sum_{i=\frac{n}{2}}^{n-1} i \\ &= (n-1) + \frac{8}{n}\left(\frac{(n-1)n}{2} - \frac{(\frac{n}{2}-1)\frac{n}{2}}{2}\right) \\ &= (n-1) + 4(n-1) - n + 2 \\ &= 4n - 3 \leq 4n \end{aligned}$$

Analyse von Quickselect

2. Fall: n ungerade.

Dann gilt:

$$
\begin{aligned}
Q(n) &\leq (n-1) + \frac{2}{n} \sum_{i=\lceil \frac{n}{2} \rceil}^{n-1} Q(i) + \frac{1}{n} Q\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \\
&\leq (n-1) + \frac{8}{n} \sum_{i=\lceil \frac{n}{2} \rceil}^{n-1} i + 2 \\
&= (n-1) + \frac{8}{n} \cdot \left(\frac{(n-1)n}{2} - \frac{(\lceil \frac{n}{2} \rceil - 1)\lceil \frac{n}{2} \rceil}{2} \right) + 2 \\
&\leq (n-1) + 4(n-1) - n + 2 + 2 \\
&= 4n - 1 \leq 4 \cdot n.
\end{aligned}
$$

Untere Schranke für Mediansuche

Theorem 13 (Blum, Floyd, Pratt, Rivest, Tarjan 1973)

Für jeden vergleichsbasierten Algorithmus zur Mediansuche und jedes n existiert ein Array der Länge n, auf dem der Algorithmus mindestens $1,5n - \mathcal{O}(\log n)$ viele Vergleiche macht.

Für den Beweis verwenden wir das folgende Lemma.

Lemma 14

Für jeden vergleichsbasierten Algorithmus zur Bestimmung des Minimums eines Arrays gilt: Der Algorithmus macht auf jedem Array der Länge n mindestens $n - 1$ viele Vergleiche.

Untere Schranke für Mediansuche

Beweis: Wir lassen den Algorithmus auf allen Arrays $A[1, \ldots, n]$, wobei $i \mapsto A[i]$ eine Permutation von $\{1, \ldots, n\}$ ist, laufen.

Wir erhalten so einen Entscheidungsbaum T_n.

Für ein Eingabearray A der Länge n erhalten wir einen Pfad p in T_n.

Sei $A[m] = 1$ das Minimum des Arrays.

Dann muss für jeden Index $i \neq m$ ein Knoten auf dem Pfad p existieren, wo i mit einem Index j verglichen wird, für den $A[i] > A[j]$ gilt.

Also muss der Pfad p aus mindestens $n - 1$ Knoten bestehen. □

Untere Schranke für Mediansuche

Beweis von Theorem 13: Sei n o.B.d.A. ungerade.

Wir lassen den Algorithmus auf allen Arrays $A[1,\ldots,n]$, wobei $i \mapsto A[i]$ eine Permutation von $\{1,\ldots,n\}$ ist, laufen.

Wir erhalten so einen Entscheidungsbaum T_n.

Für eine Eingabepermutation A endet der durch A bestimmte Pfad in einem mit i beschrifteten Blatt, wobei $A[i] = \lceil n/2 \rceil$.

Behauptung: Die Anzahl der Blätter des Baums T_n ist mindestens

$$\binom{n}{\frac{n-1}{2}} 2^{\frac{n-3}{2}} \geq \frac{2^n}{n} \cdot 2^{\frac{n-3}{2}} = \frac{1}{n} 2^{\frac{3n-3}{2}}$$

Untere Schranke für Mediansuche

Also existiert in T_n ein Pfad der Länge mindestens

$$\log_2 \left(\frac{1}{n} 2^{\frac{3n-3}{2}} \right) = 1,5n - \mathcal{O}(\log n).$$

Beweis der Behauptung:

Für ein Array A sei

$$M_A = \{i \mid 1 \leq i \leq n, A[i] < \lceil n/2 \rceil\}$$

(Menge aller Indizes i, so das $A[i]$ kleiner als der Median ist).

Beachte: $|M_A| = \lfloor n/2 \rfloor$.

Behauptung 1: Wenn $M_A \neq M_B$, dann sind die durch A und B bestimmten Pfade in T_n verschieden.

Untere Schranke für Mediansuche

Beweis von Behauptung 1: Angenommen $M_A \neq M_B$ aber A und B bestimmen den gleichen Pfad p in T_n.

Insbesondere steht der Median an der gleichen Position m in A und B: $A[m] = B[m] = \lceil n/2 \rceil$.

Wegen $M_A \neq M_B$ existiert $1 \leq i_0 \leq n$ mit $i_0 \in M_A \setminus M_B$.

Also gilt $A[i_0] < \lceil n/2 \rceil = A[m]$ und $B[i_0] > \lceil n/2 \rceil = B[m]$.

Seien $i_1, \ldots, i_k$ so, dass $i_k = m$, $A[i_{s+1}] = A[i_s] + 1$ für $0 \leq s \leq k-1$.

Insbesondere gilt $i_1, \ldots, i_{k-1} \in M_A$.

O.B.d.A. gelte $i_1, \ldots, i_{k-1} \in M_B$, d.h.

$$B[i_1], \ldots, B[i_{k-1}] < \lceil n/2 \rceil = B[i_k] < B[i_0].$$

Also gilt $A[i_0] < A[i_s]$ und $B[i_0] > B[i_s]$ für alle $1 \leq s \leq k$.

Dies bedeutet, dass auf dem Pfad p kein Vergleich der Form $i_0 : i_s$ $(1 \leq s \leq k)$ stattfindet.

Untere Schranke für Mediansuche

Wir definieren nun ein drittes Array C durch

$$C[j] = \begin{cases} \lceil n/2 \rceil = A[i_k] & \text{für } j = i_0 \\ A[i_{s-1}] & \text{für } j = i_s,\ 1 \leq s \leq k \\ A[j] & \text{sonst} \end{cases}$$

Dann bestimmt C den gleichen Pfad durch den Entscheidungsbaum T_n wie A.

Dies widerpricht jedoch $C[i_0] = \lceil n/2 \rceil$, $i_0 \neq m$, was Behauptung 1 zeigt.

Wegen Behauptung 1 können wir jedem Blatt v von T_n eine Menge $M_v \subseteq \{1, \ldots, n\}$ zuordnen, so dass $M_A = M_v$ für jedes Array A, das zum Blatt v führt, gilt.

Ausserdem gibt es für jede Menge $M \subseteq \{1, \ldots, n\}$ mit $|M| = \lfloor n/2 \rfloor$ ein Blatt v mit $M_v = M$.

Insbesondere gibt es mindestens $\binom{n}{\lfloor n/2 \rfloor}$ viele Blätter.

Untere Schranke für Mediansuche

Behauptung 2: Für jede Menge $M \subseteq \{1, \ldots, n\}$ mit $|M| = \lfloor n/2 \rfloor$ gibt es mindestens $2^{(n-3)/2}$ viele Blätter v mit $M_v = M$.

Beweis von Behauptung 2:

Fixiere $M \subseteq \{1, \ldots, n\}$ mit $|M| = \lfloor n/2 \rfloor$.

Fixiere außerdem eine beliebige Abbildung $f : M \mapsto \{1, \ldots, \lfloor n/2 \rfloor\}$.

Sei nun $\mathcal{E}$ die Menge aller Eingabearrays A mit der Eigenschaft: $A[i] = f(i)$ für alle $i \in M$.

Dann gilt:

- Für jedes Array $A \in \mathcal{E}$ gilt $M_A = M$.
- Also führt A im Entscheidungsbaum T_n zu einem Blatt v mit $M_v = M$.
- Ein Arraymenge $\mathcal{E}$ entspricht der Menge der Arrays mit den $\lceil n/2 \rceil$ Einträgen $\lceil n/2 \rceil, \ldots, n$.

Untere Schranke für Mediansuche

Aus dem Entscheidungsbaum T_n konstruieren wir nun einen neuen Entscheidungsbaum $T_{\lceil n/2 \rceil}$, welcher das Minimum eines Arrays mit den $\lceil n/2 \rceil$ Einträgen $\lceil n/2 \rceil, \ldots, n$ bestimmt.

Für jeden inneren Knoten v von T_n, der mit dem Vergleich $i : j$ beschriftet ist, tue folgendes (T_v sei der in v gewurzelte Teilbaum):

- Wenn $i, j \notin M$, dann tue nichts.
- Wenn $i \in M$, $j \notin M$, dann ersetzte T_v durch den Teilbaum, der zu $A[i] < A[j]$ gehört (linker Teilbaum).
- Wenn $i, j \in M$ mit $f(i) < f(j)$, dann ersetzte T_v durch den Teilbaum, der zu $A[i] < A[j]$ gehört (linker Teilbaum).

So behalten wir in T_n genau die Pfade, die für Eingabearrays aus $\mathcal{E}$ durchlaufen werden.

Untere Schranke für Mediansuche

Der resultierende Entscheidungsbaum $T_{\lceil n/2 \rceil}$ hat folgende Eigenschaften:

- Für jedes Blatt v von $T_{\lceil n/2 \rceil}$ gilt $M_v = M$.
- $T_{\lceil n/2 \rceil}$ ist ein Entscheidungsbaum zur Bestimmung des Minimums eines Arrays mit den $\lceil n/2 \rceil$ Einträgen $\lceil n/2 \rceil, \ldots, n$.
- Aus dem vorhergehenden Punkt und Lemma 14 folgt, dass jeder Pfad von $T_{\lceil n/2 \rceil}$ mindestens Länge $\lceil n/2 \rceil - 1 = \lfloor n/2 \rfloor$ Knoten hat.
- Also hat $T_{\lceil n/2 \rceil}$ mindestens $2^{\lfloor n/2 \rfloor - 1} = 2^{(n-3)/2}$ viele Blätter.

Dies zeigt Behauptung 2 und damit Theorem 13. □

Man kann die untere Schranke $1,5n - \mathcal{O}(\log n)$ verbessern zu $2n + o(n)$ (Brent, John 1985).

Andererseits kann man den Median mit $2,95n + o(n)$ vielen Vergleichen finden (Dor, Zwick 1995)

Überblick

- Matroide
- Kruskals Algorithmus für aufspannende Bäume
- Dijkstras Algorithmus für kürzeste Pfade

Optimalitätsprinzip

Greedy („gierig“) bezeichnet Lösungsstrategien, die auf der schrittweisen Berechnung von Teillösungen (lokalen Optima) basieren.

Dieses Verfahren eignet sich für Probleme, bei denen jede Folge von lokal optimalen Berechnungsschritten ein globales Optimum findet (Optimalitätsprinzip).

Diese Technik funktioniert, wenn die zugrundeliegende Struktur ein Matroid bildet.

Optimierungsprobleme

Sei E eine endliche Menge und $U \subseteq 2^E$ eine Menge von Teilmengen von E.

Das Paar (E, U) ist ein Teilmengensystem, falls gilt:

- $\emptyset \in U$
- Wenn $A \subseteq B \in U$ dann gilt auch $A \in U$

Ein Menge $A \in U$ ist maximal (bzgl. $\subseteq$) falls für alle $B \in U$ gilt: Wenn $A \subseteq B$, dann $A = B$.

Das zu (E, U) gehörende Optimierungsproblem ist:

- Eingabe: Eine Gewichtsfunktion $w : E \to \mathbb{R}$
- Ausgabe: Eine maximale Menge $A \in U$ mit $w(A) \geq w(B)$ für alle maximalen Mengen $B \in U$, wobei

$$w(C) = \sum_{a \in C} w(a)$$

(wir nennen A ein optimale Lösung).

Optimierungsprobleme

Zur Lösung eines solchen Optimierungssystems kann man versuchen, den folgenden kanonischen Greedy-Algorithmus anzuwenden:

Algorithmus kanonischer Greedy-Algorithmus

---

procedure find-optimum(Teilmengensystem (E, U), $w : E \to \mathbb{R}$)
begin
  ordne Menge E nach absteigenden Gewichten als $e_1, e_2, \ldots, e_n$ mit $w(e_1) \geq w(e_2) \geq \cdots \geq w(e_n)$
  $T := \emptyset$
  for $k := 1$ **to** n **do**
    if $T \cup \{e_k\} \in U$ **then** $T := T \cup \{e_k\}$
  endfor
  return (T)
endprocedure

Matroide

Beachte: Die vom kanonischen Greedy-Algorithmus berechnete Lösung ist eine maximale Teilmenge.

Leider gibt es aber Teilmengensysteme für die der kanonische Greedy-Algorithmus keine optimale Lösung findet (Übung).

Ein Teilmengensystem (E, U) ist ein Matroid, falls folgende Eigenschaft gilt (Austauscheigenschaft):

$$\forall A, B \in U : |A| < |B| \implies \exists x \in B \setminus A : A \cup \{x\} \in U$$

Bemerkung: Wenn (E, U) ein Matroid ist, dann haben alle maximalen Teilmengen in U die gleiche Mächtigkeit.

Beispiel: Sei E eine endliche Menge und $k \leq |E|$. Dann ist

$$(E, \{A \subseteq E \mid |A| \leq k\})$$

ein Matroid.

Matroide

Theorem 15

Sei (E, U) ein Teilmengensystem. Der kanonische Greedy-Algorithmus berechnet für jede Gewichtsfunktion $w : E \to \mathbb{R}$ eine optimale Lösung genau dann, wenn (E, U) ein Matroid ist.

Beweis: Sei zunächst (E, U) ein Matroid.

Sei $w : E \to \mathbb{R}$ eine Gewichtsfunktion und sei $E = \{e_1, e_2, \ldots, e_n\}$ mit

$$w(e_1) \geq w(e_2) \geq \cdots \geq w(e_n).$$

Sei $T = \{e_{i_1}, \ldots, e_{i_k}\}$ mit $i_1 < i_2 < \cdots < i_k$ die vom kanonischen Greedy-Algorithmus berechnete Lösung.

Annahme: Es gibt eine maximale Menge $S = \{e_{j_1}, \ldots, e_{j_l}\} \in U$ mit $w(S) > w(T)$, wobei $j_1 < j_2 < \cdots < j_l$.

Da (E, U) ein Matroid ist, gilt $k = l$.

Matroide

Wegen $w(S) > w(T)$ gibt es ein $1 \leq p \leq k$ mit $w(e_{j_p}) > w(e_{i_p})$.

Da die Gewichte absteigend sortiert wurden, muss $j_p < i_p$ gelten.

Wende nun die Austauscheigenschaft an auf

$$A = \{e_{i_1}, \ldots, e_{i_{p-1}}\} \in U \quad \text{und} \quad B = \{e_{j_1}, \ldots, e_{j_p}\} \in U.$$

Wegen $|A| < |B|$ gibt es ein Element $e_{j_q} \in B \setminus A$ mit $A \cup \{e_{j_q}\} \in U$.

Also gilt $j_q \leq j_p < i_p$.

Dann hätte der kanonische Greedy-Algorithmus aber auch e_{j_q} ausgewählt, was ein Widerspruch ist.

Matroide

Sei nun (E, U) kein Matroid, d.h. die Austauscheigenschaft ist verletzt.

Seien $A, B \in U$ mit $|A| < |B|$, so dass für alle $b \in B \setminus A$ gilt: $A \cup \{b\} \notin U$.

Sei $r = |B|$ und damit $|A| \leq r - 1$.

Definiere die Gewichtsfunktion $w : E \to \mathbb{R}$ wie folgt:

$$w(x) = \begin{cases} r+1 & \text{für } x \in A \\ r & \text{für } x \in B \setminus A \\ 0 & \text{sonst} \end{cases}$$

Der kanonische Greedy-Algorithmus berechnet daher eine Lösung T mit $A \subseteq T$ und $T \cap (B \setminus A) = \emptyset$.

Daher gilt: $w(T) = (r+1) \cdot |A| \leq (r+1)(r-1) = r^2 - 1$.

Sei nun $S \in U$ eine maximale Teilmenge mit $B \subseteq S$.

Dann gilt $w(S) \geq w(B) \geq r^2$. □

Aufspannende Teilbäume

Sei $G = (V, E)$ ein endlicher ungerichteter Graph.

Ein Pfad von $u \in V$ nach $v \in V$ ist eine Folge von Knoten $(u_1, u_2, \ldots, u_n)$ mit $u_1 = u$, $u_n = v$ und $(u_i, u_{i+1}) \in E$ für alle $1 \leq i \leq n-1$.

G ist zusammenhängend, falls für alle $u, v \in V$ mit $u \neq v$ ein Pfad von u nach v existiert.

Ein Kreis ist ein Pfad $(u_1, u_2, \ldots, u_n)$ mit $n \geq 3$, $u_i \neq u_j$ für alle $1 \leq i < j \leq n$ und $(u_n, u_1) \in E$.

G ist ein Baum, falls G zusammenhängend ist und keine Kreise hat.

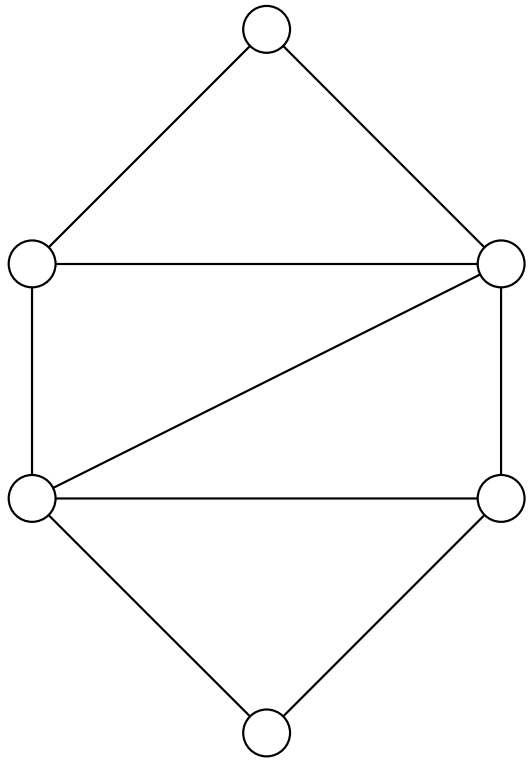
Übung: Für jeden Baum $T = (V, E)$ gilt $|E| = |V| - 1$.

Sei nun $G = (V, E)$ zusammenhängend. Ein aufspannender Teilbaum von G ist eine Teilmenge $F \subseteq E$ von Kanten, so dass (V, F) ein Baum ist.

Übung: Für jeden zusammenhängenden Graphen existiert ein aufspannender Teilbaum.

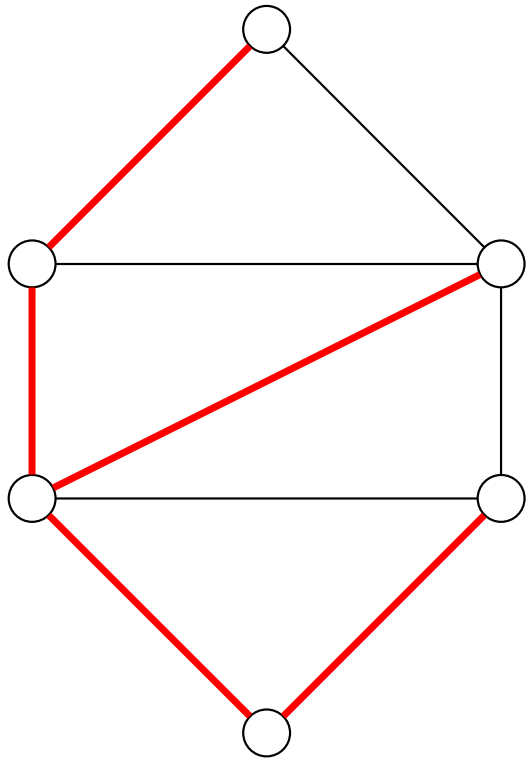
Aufspannende Teilbäume

Beispiel:



Aufspannende Teilbäume

Beispiel:



Matroid der kreisfreien Kantenmengen

Sei $G = (V, E)$ wieder zusammenhängend, und sei $w : E \to \mathbb{R}$ eine Gewichtsfunktion.

Das Gewicht eines aufspannenden Teilbaums $F \subseteq E$ ist

$$w(F) = \sum_{e \in F} w(e).$$

Ziel: Berechne einen aufspannenden Teilbaum mit maximalen Gewicht.

Das folgende Lemma erlaubt uns, den kanonischen Greedy-Algorithmus zu verwenden:

Lemma 16

Das Teilmengensystem $(E, \{A \subseteq E \mid (V, A)$ *ist kreisfrei*$\})$ *ist ein Matroid.*

Matroid der kreisfreien Kantenmengen

Beweis: Seien $A, B \subseteq E$ kreisfreie Kantenmengen mit $|A| < |B|$.

Seien $V_1, V_2 \ldots, V_n$ die Zusammenhangskomponenten des Graphens (V, A).

Dann gilt $|A| = \sum_{i=1}^{n}(|V_i| - 1)$, denn der auf V_i induzierte Teilgraph von (V, A) ist ein Baum, hat also $|V_i| - 1$ viele Kanten.

Für jede Kante $e = (u, v) \in B$ gilt genau einer der beiden folgenden Fälle:

1. Es gibt ein $1 \leq i \leq n$ mit $u, v \in V_i$.
2. Es gibt $i \neq j$ mit $u \in V_i$ und $v \in V_j$.

In B können aber nur höchstens $\sum_{i=1}^{n}(|V_i| - 1) = |A|$ vielen Kanten vom Typ 1 vorhanden sein (wenn nicht, würde B einen Kreis in einer der Mengen V_i haben).

Also gibt es eine Kante $e \in B$, die zwei Zusammenhangskomponenten von (V, A) verbindent.

Also ist $A \cup \{e\}$ wieder kreisfrei. □

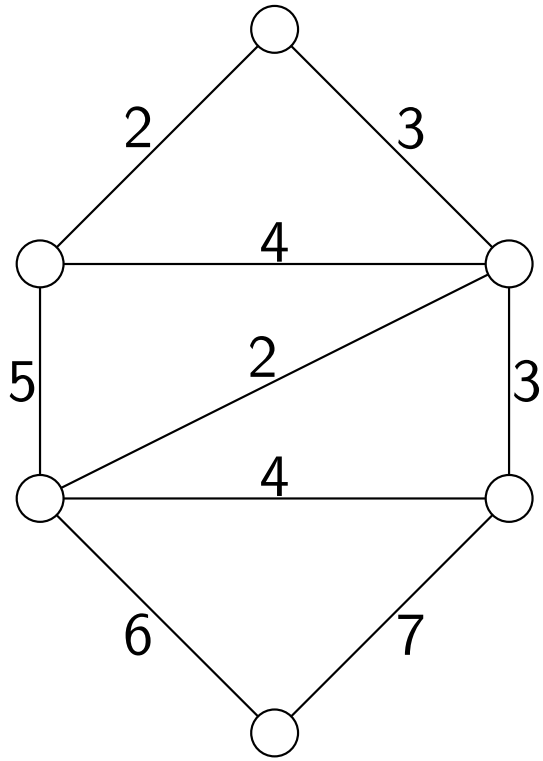
Kruskals Algorithmus

Algorithmus Kruskals Algorithmus

procedure kruskal(Kanten-gewichteter zusammenhängender Graph (V, E, w)
begin
  ordne Menge E nach absteigenden Gewichten als $e_1, e_2, \ldots, e_n$ mit $w(e_1) \geq w(e_2) \geq \cdots \geq w(e_n)$
  $F := \emptyset$
  for $k := 1$ **to** n **do**
    if e_k verbindet verschiedene Zusammenhangskomp. von (V, F) **then**
      $F := F \cup \{e_k\}$
  endfor
  return (F)
endprocedure

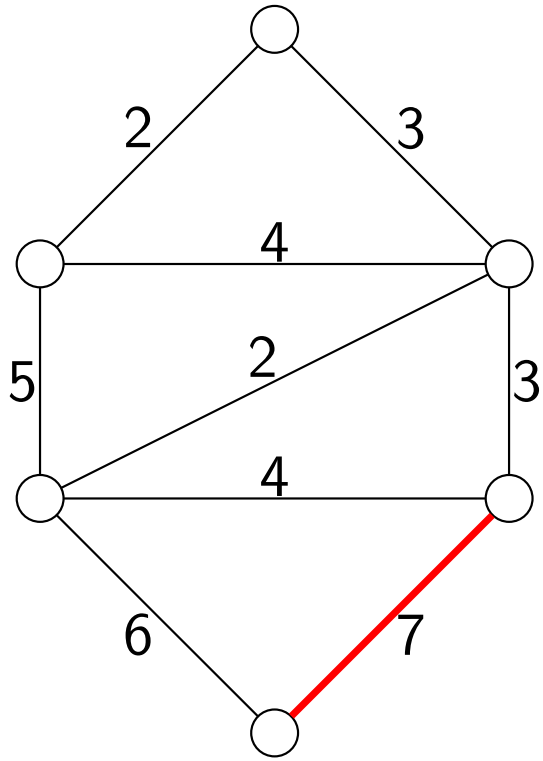
Kruskals Algorithmus

Beispiel für Kruskals Algorithmus:



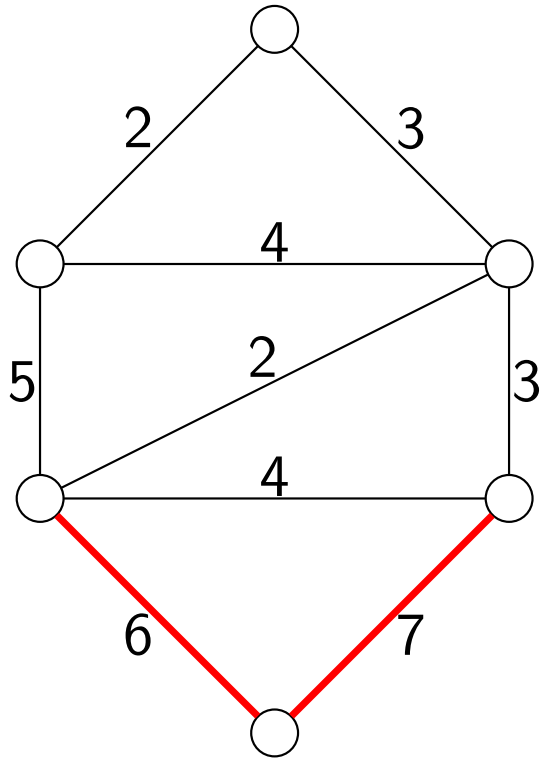
Kruskals Algorithmus

Beispiel für Kruskals Algorithmus:



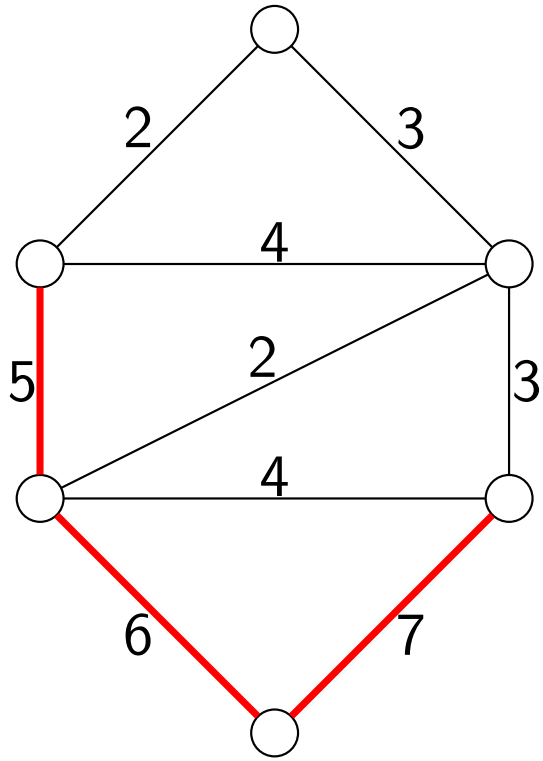
Kruskals Algorithmus

Beispiel für Kruskals Algorithmus:



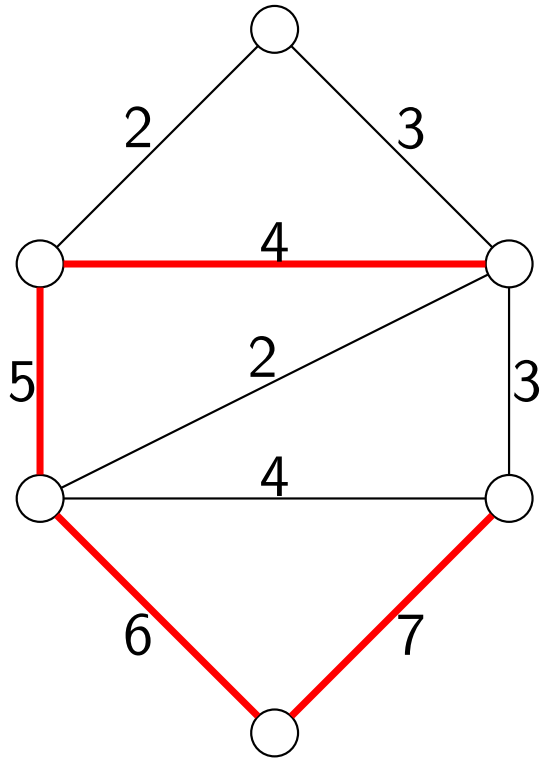
Kruskals Algorithmus

Beispiel für Kruskals Algorithmus:



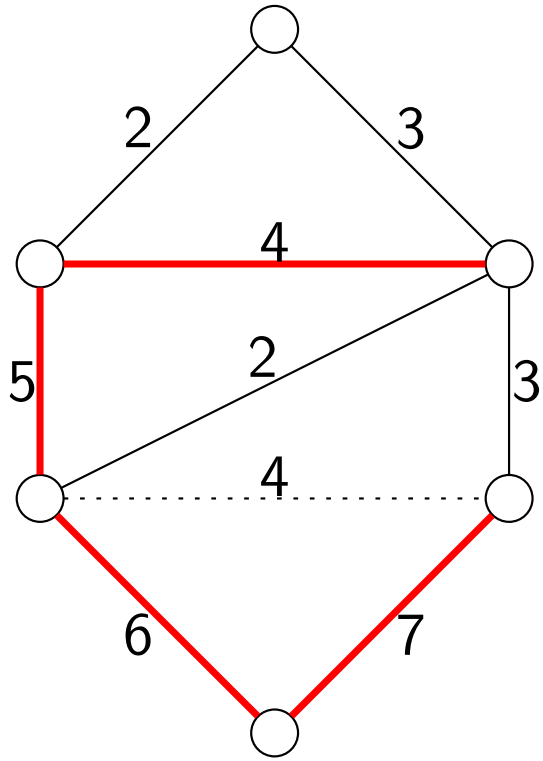
Kruskals Algorithmus

Beispiel für Kruskals Algorithmus:



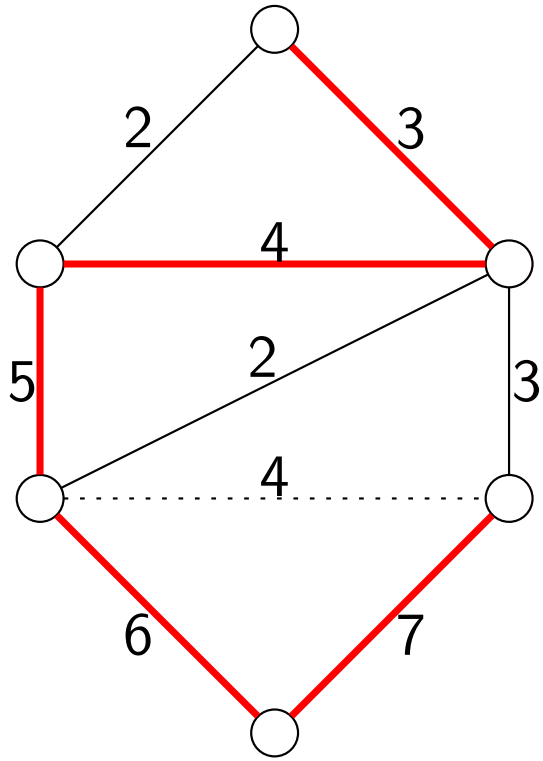
Kruskals Algorithmus

Beispiel für Kruskals Algorithmus:



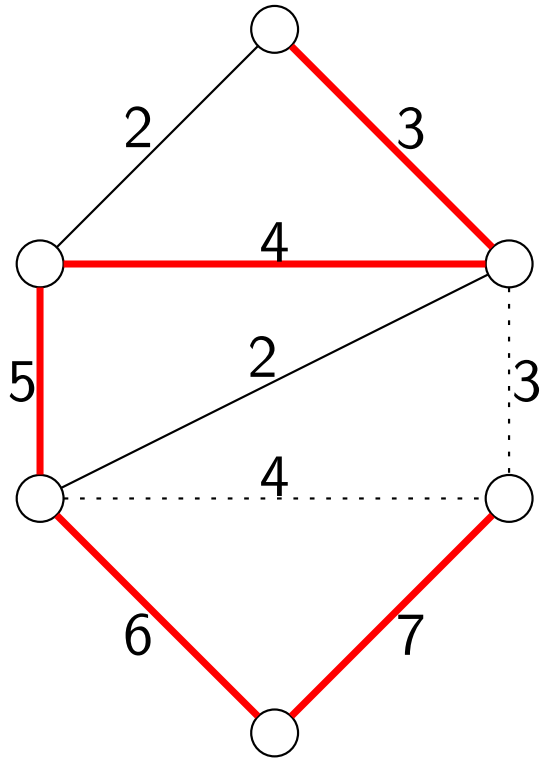
Kruskals Algorithmus

Beispiel für Kruskals Algorithmus:



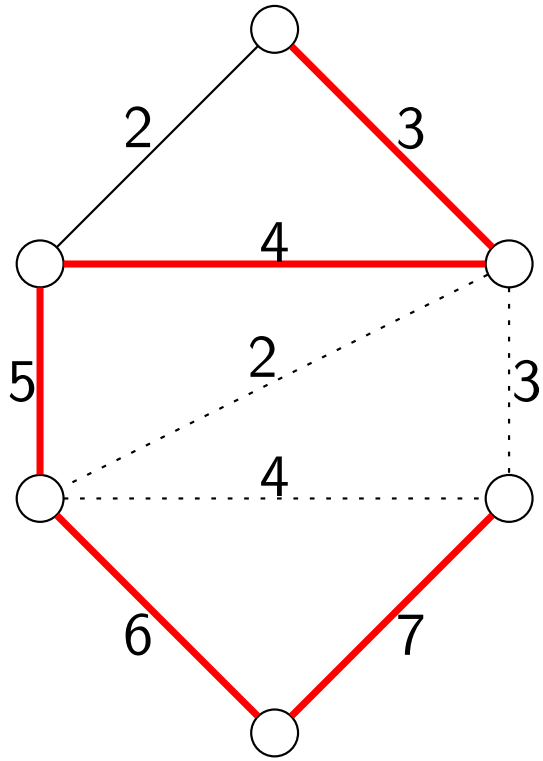
Kruskals Algorithmus

Beispiel für Kruskals Algorithmus:



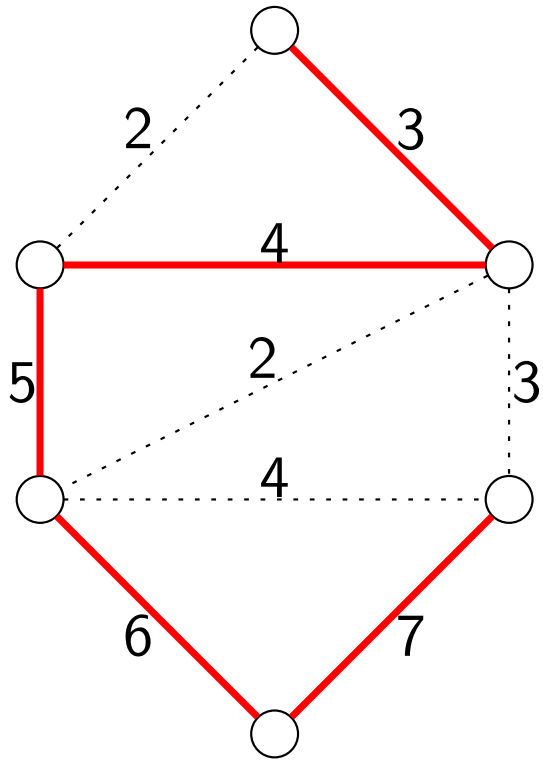
Kruskals Algorithmus

Beispiel für Kruskals Algorithmus:



Kruskals Algorithmus

Beispiel für Kruskals Algorithmus:



Laufzeit von Kruskals Algorithmus

Beachte: Da G zusammenhängend ist, gilt $|V| - 1 \leq |E| \leq |V|^2$.

Sortieren der Kanten nach ihrem Gewicht benötigt Zeit $\mathcal{O}(|E| \log |E|) = \mathcal{O}(|E| \log |V|)$.

Die Zusammenhangskomponenten von (V, F) kann man sich als eine Partition der Knotenmenge V verwalten.

Wir beginnen mit der Partition $\{\{v\} \mid v \in V\}$.

In jedem Durchlauf der **for**-Schleife ($|E|$ viele) testen wir, ob die Endknoten der Kante e_k in verschiedenen Mengen A, B der Partition liegen.

Falls dies der Fall ist, ersetzen wir in der Partition A und B durch die Menge $A \cup B$.

Wir werden hierfür später eine Datenstruktur (Union-Find) entwickeln, welche diese Operationen in Zeit insgesamt $\mathcal{O}(\alpha(|V|) \cdot |E|)$ für eine sehr schwach wachsende Funktion α bewerkstelligt.

Dies liefert dann eine Laufzeit von $\mathcal{O}(|E| \log |V|)$ für Kruskals Algorithmus.

Kürzeste Wege

Weiteres Beispiel für die Greedy-Strategie: Bestimmung kürzester Wege in einem Kanten-gewichteten gerichteten Graphen $G = (V, E, \gamma)$.

- V ist die Knotenmenge.
- $E \subseteq V \times V$ ist die Kantenmenge, wobei $(x, x) \notin E$ für alle $x \in V$ gelten soll.
- $\gamma : E \to \mathbb{N}$ ist die Gewichtsfunktion.

Gewicht eines Pfades $(v_0, v_1, v_2, \ldots, v_n)$: $\sum_{i=0}^{n-1} \gamma(v_i, v_{i+1})$

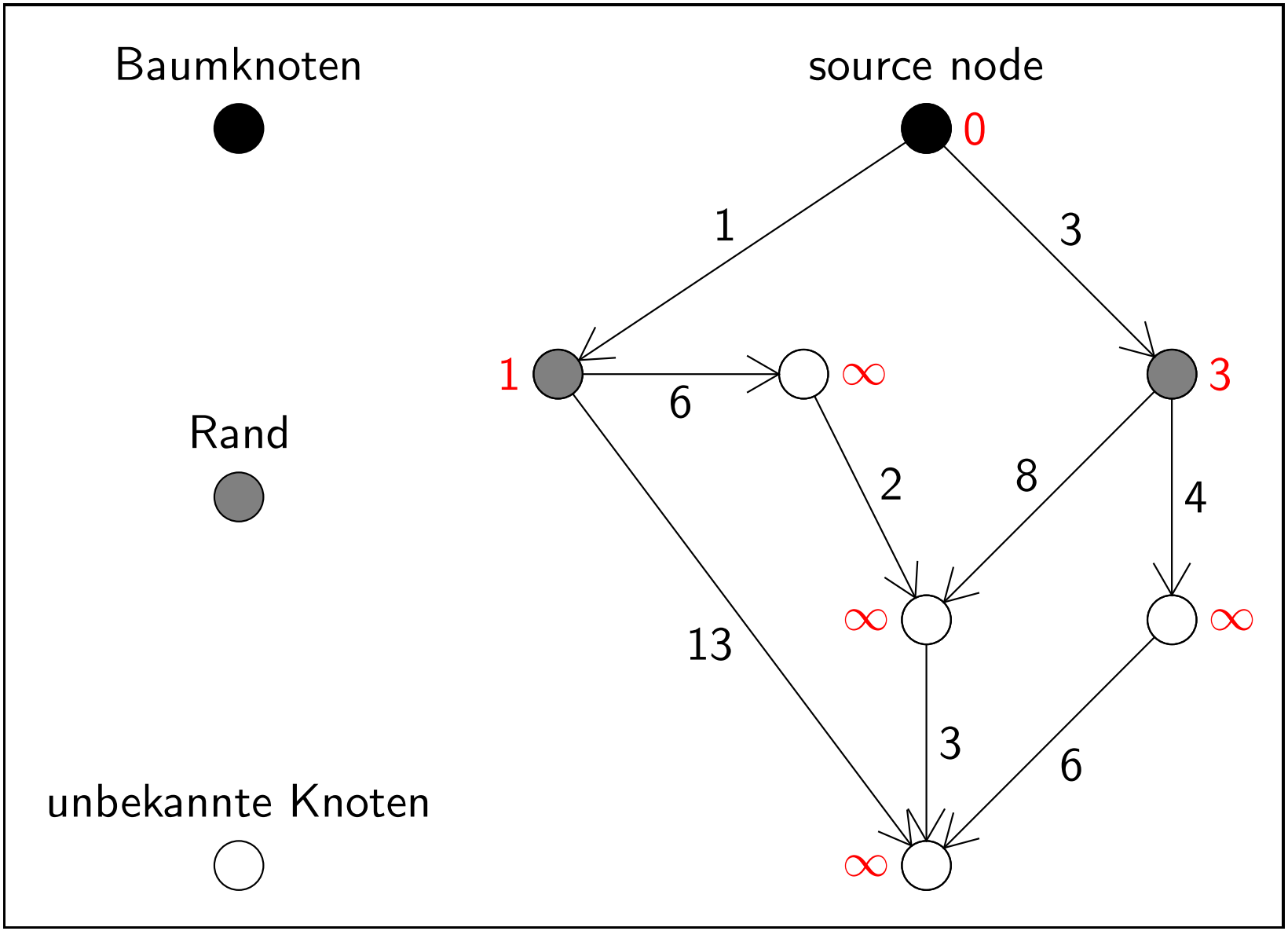
Für $u, v \in V$ ist $d(u, v)$ das Minimum der Gewichte aller Pfade von u nach v ($d(u, v) = \infty$, falls kein Pfad von u nach v existiert, und $d(u, u) = 0$).

Ziel: Gegeben $G = (V, E, \gamma)$ und Startknoten $u \in V$, berechne für jedes $v \in V$ einen Pfad $u = v_0, v_1, v_2, \ldots, v_{n-1}, v_n = v$ mit minimalem Gewicht $d(u, v)$.

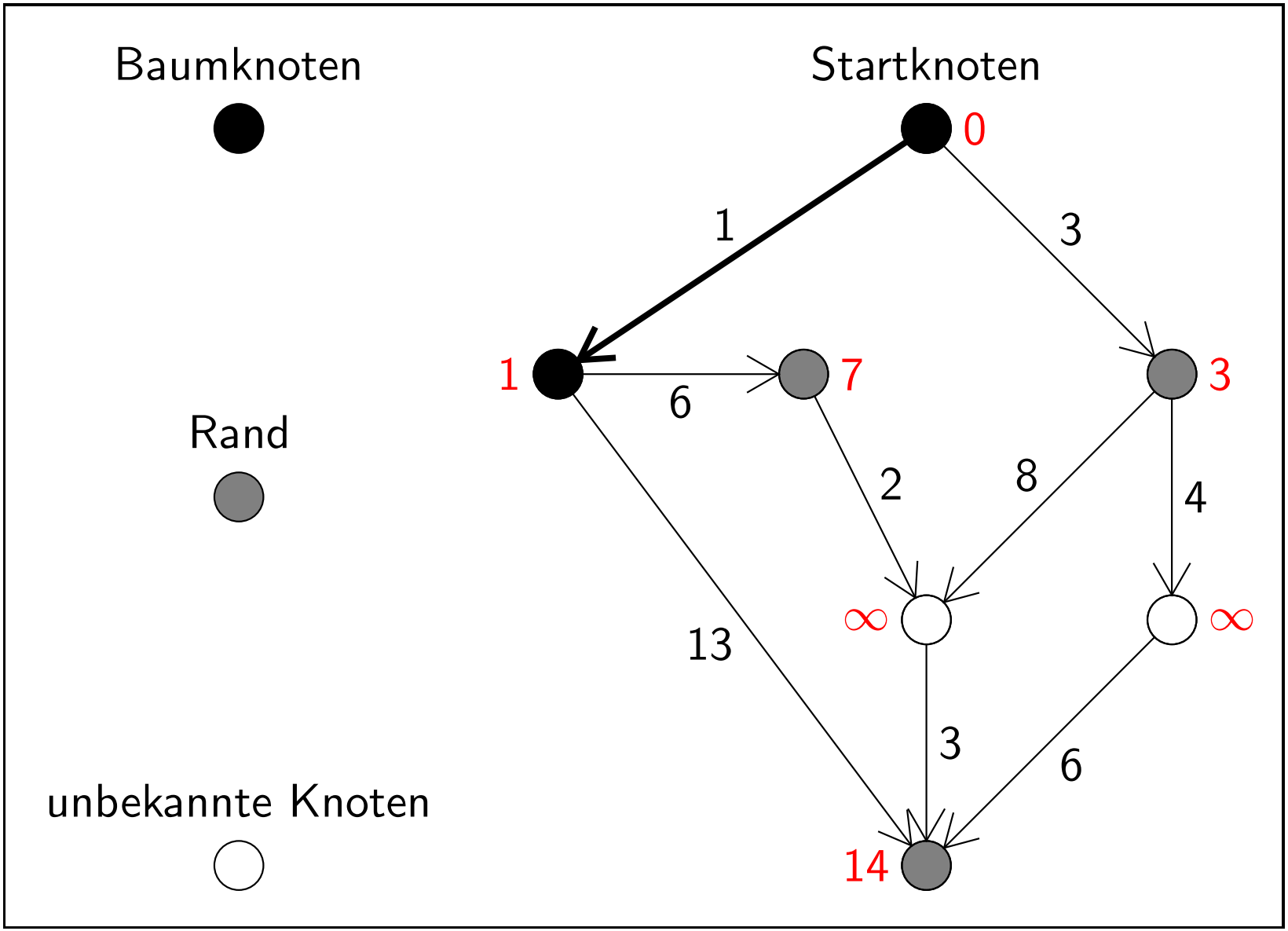
Dijkstras Algorithmus

$B := \emptyset$ (Baumknoten); $R := \{u\}$ (Rand); $U := V \setminus \{u\}$ (unbekannt);
$p(u) :=$ **nil**; $D(u) := 0$;
while $R \neq \emptyset$ **do**
  $x :=$ **nil**; $\alpha := \infty$;
  forall $y \in R$ **do**
    if $D(y) < \alpha$ **then**
      $x := y$; $\alpha := D(y)$
    endif
  endfor
  $B := B \cup \{x\}$; $R := R \setminus \{x\}$
  forall $(x, y) \in E$ **do**
    if $y \in U$ **then**
      $D(y) := D(x) + \gamma(x, y)$; $p(y) := x$; $U := U \setminus \{y\}$; $R := R \cup \{y\}$
    elsif $y \in R$ **and** $D(x) + \gamma(x, y) < D(y)$ **then**
      $D(y) := D(x) + \gamma(x, y)$; $p(y) := x$
    endif
  endfor
endwhile

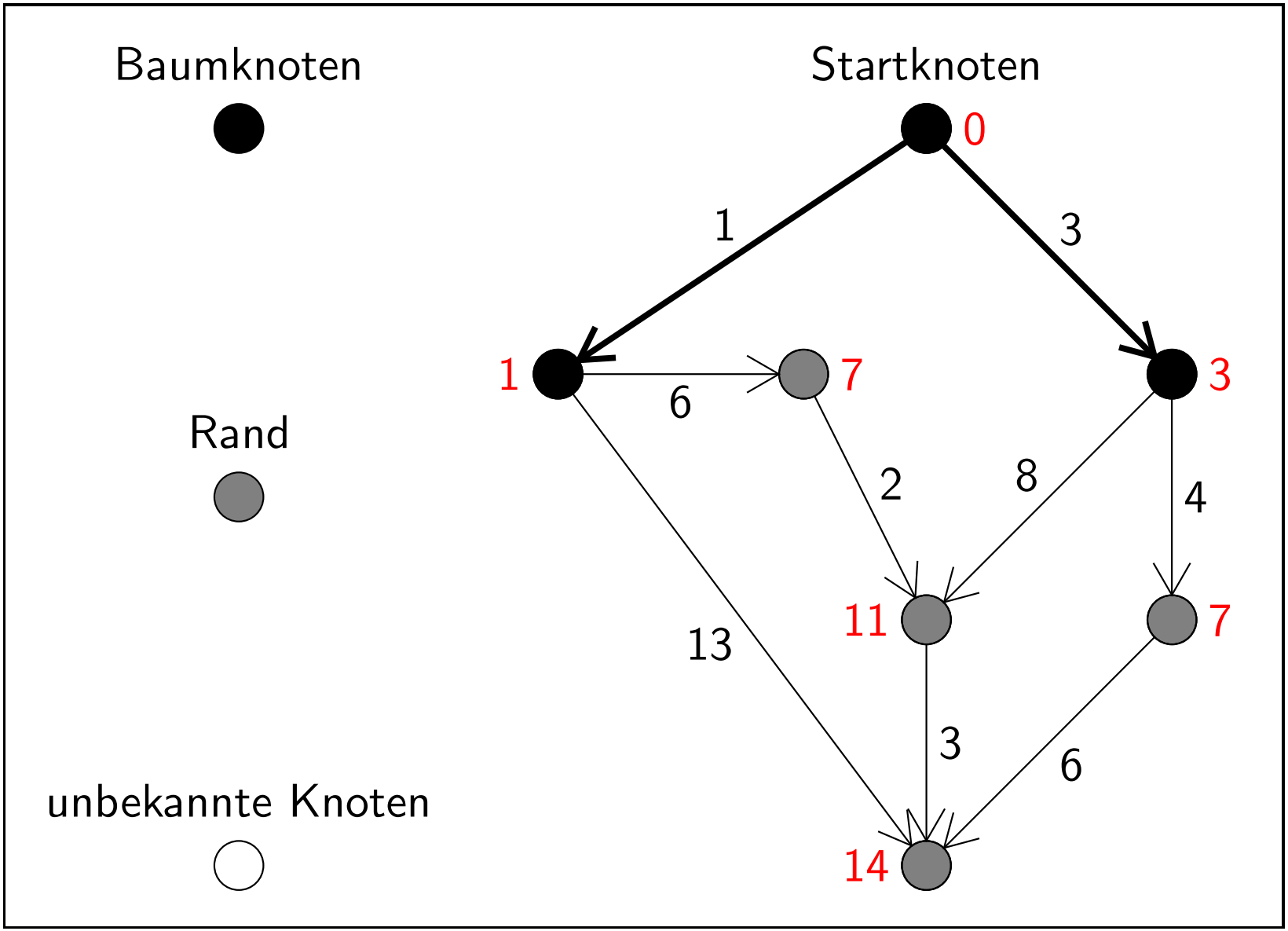
Beispiel zu Dijkstras Algorithmus



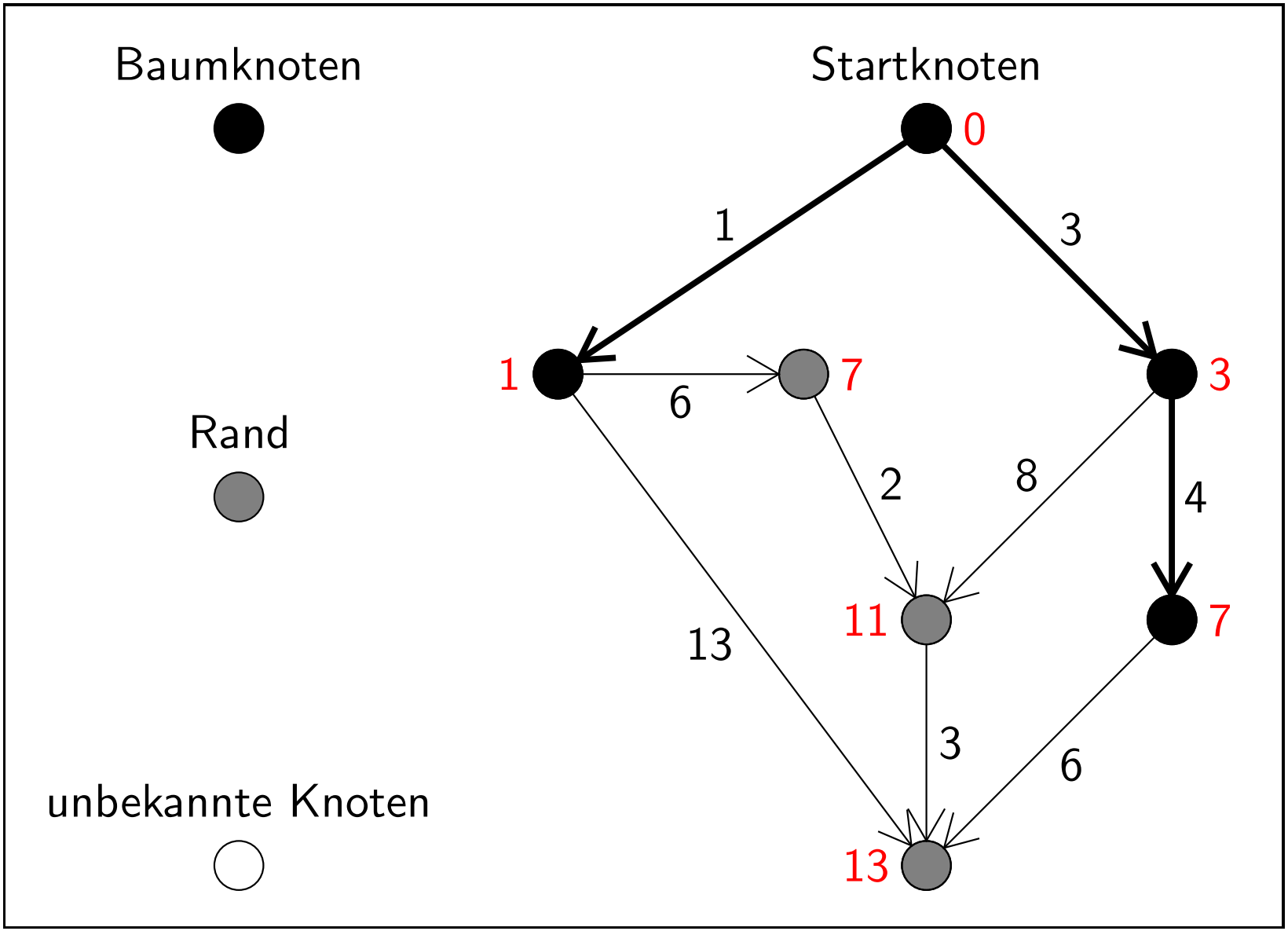
Beispiel zu Dijkstras Algorithmus



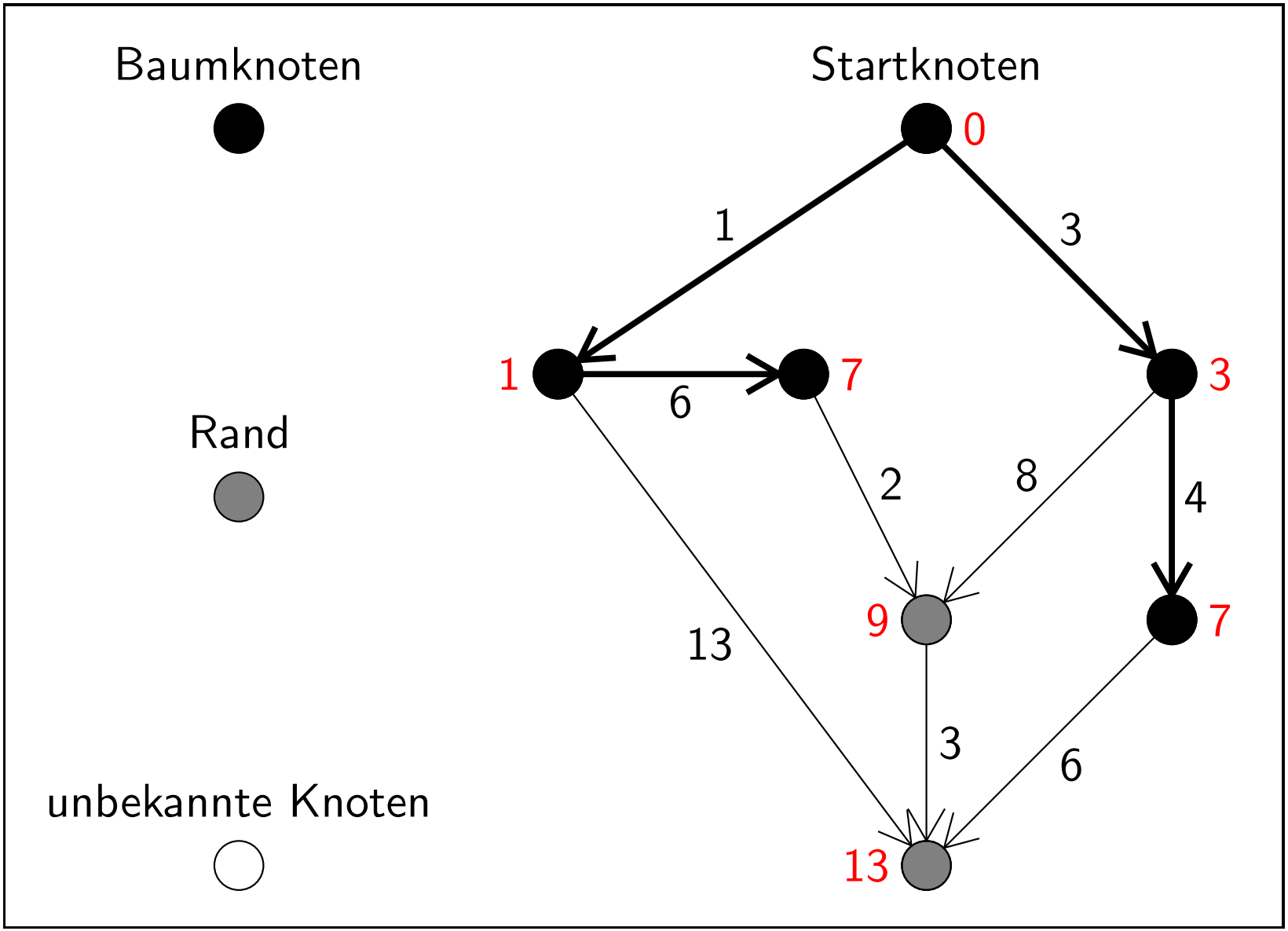
Beispiel zu Dijkstras Algorithmus



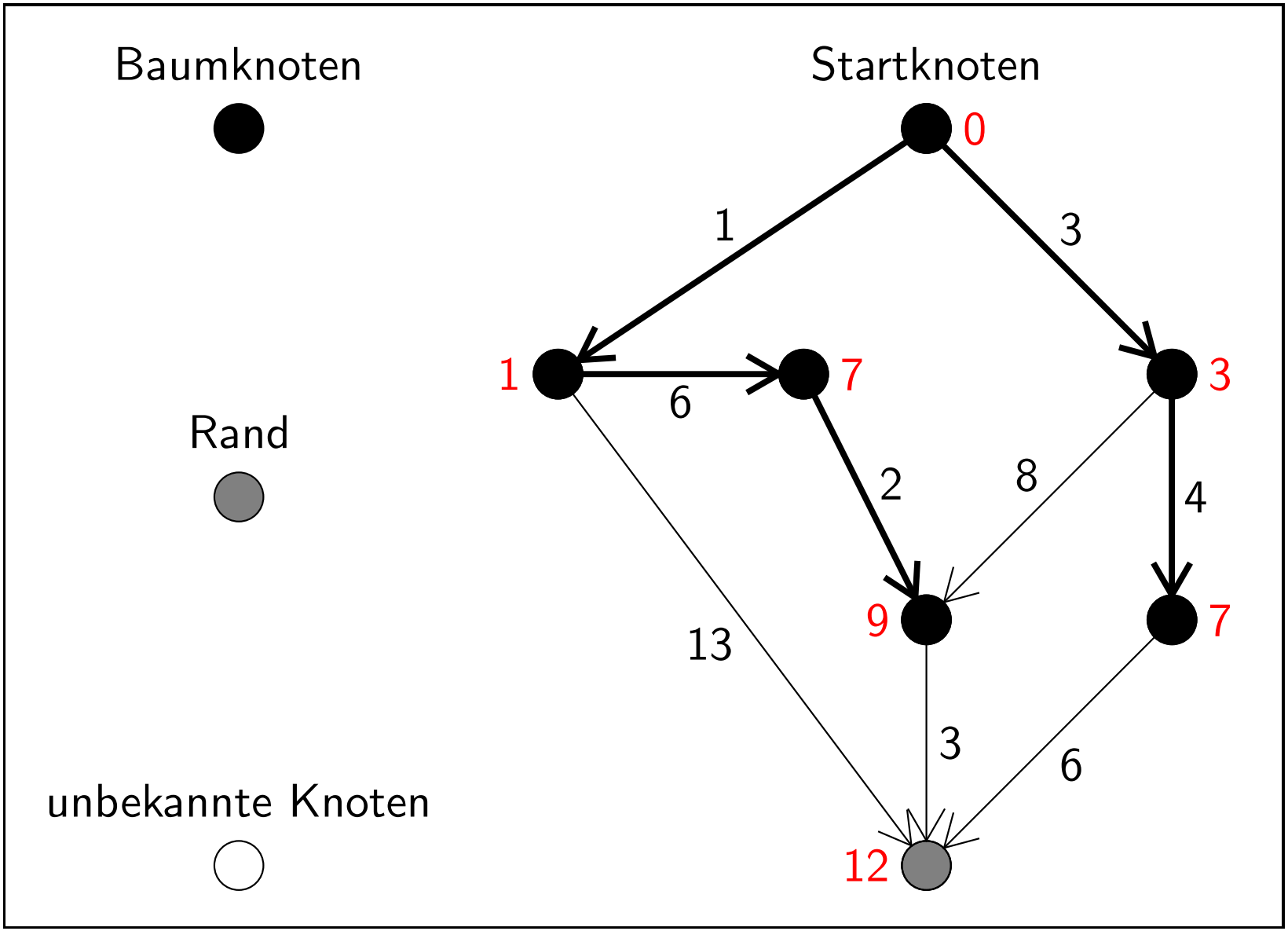
Beispiel zu Dijkstras Algorithmus



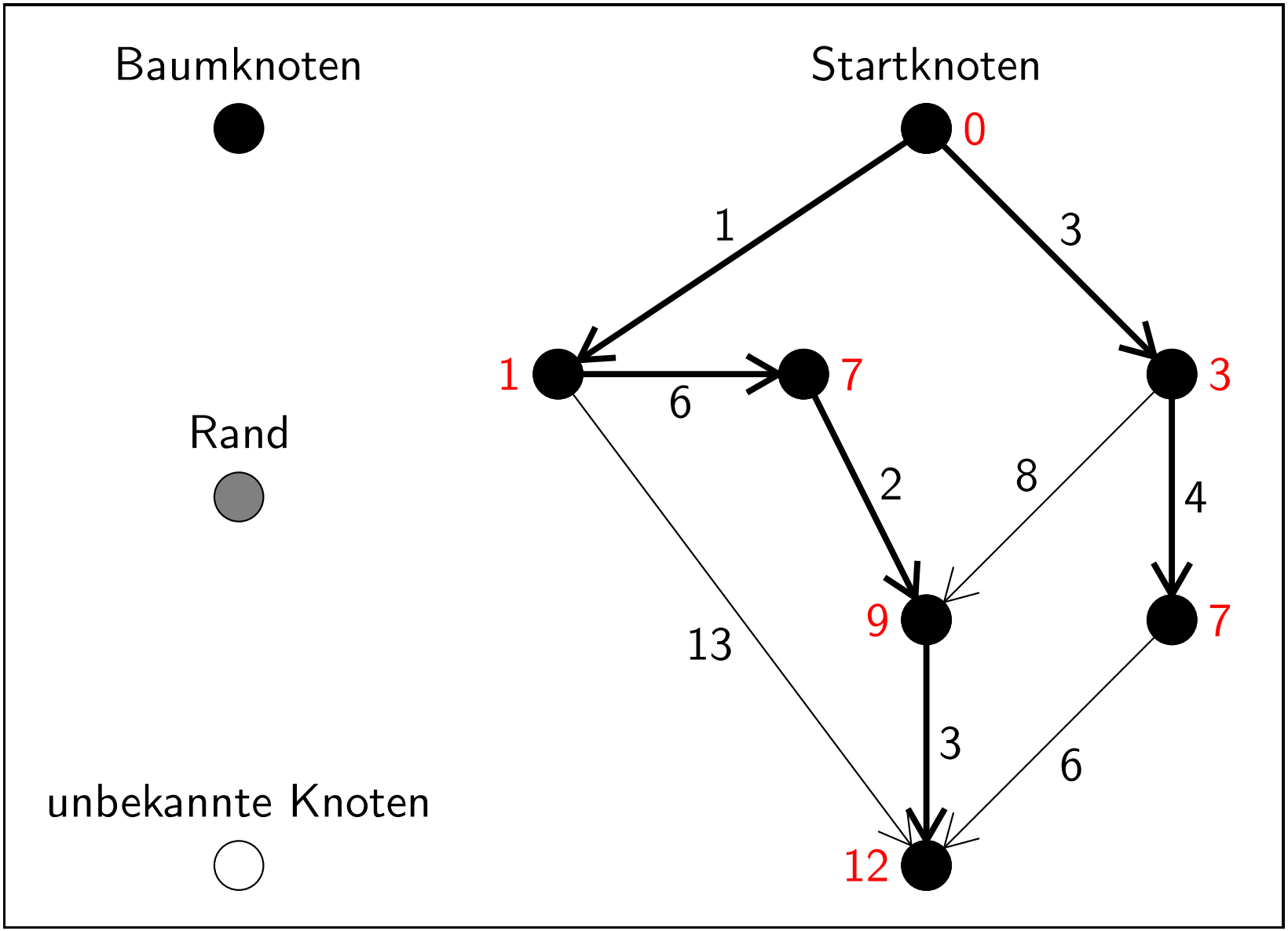
Beispiel zu Dijkstras Algorithmus



Beispiel zu Dijkstras Algorithmus



Beispiel zu Dijkstras Algorithmus



Korrektheit von Dijkstras Algorithmus

Theorem 17 (Korrektheit von Dijkstras Algorithmus)

Der Dijkstra–Algorithmus berechnet alle kürzesten Pfade von einer Quelle aus.

Beweis: Wir zeigen, dass die folgenden Invarianten durch den Riumpf der **while**-Schleife erhalten bleiben:

1. Die Mengen B, R, und U bilden eine Partition der Knotenmenge V.
2. $R = \{y \mid \exists x \in B : (x, y) \in E\} \setminus B$
3. Für alle $x \in B$ gilt $D(x) = d(u, x)$
4. Für alle $y \in R$ gilt $D(y) = \min\{D(x) + \gamma(x, y) \mid x \in B, (x, y) \in E\}$

Betrachte eine Ausführung des Rumpfes der **while**-Schleife, wobei der Knoten x von R nach B verschoben wird.

(1)–(4) gelten vor der Ausführung der **while**-Schleife.

Es ist offensichtlich, dass (1) und (2) erhalten bleiben.

Korrektheit von Dijkstras Algorithmus

Zu (3): Wegen (3) und (4) existiert ein Knoten $z \in B$ mit

$$D(x) = D(z) + \gamma(z, x) = d(u, z) + \gamma(z, x).$$

Also existiert ein Pfad von u nach x mit Gewicht $D(x)$

Angenommen, es gibt einen Pfad von u nach x mit einem Gewicht $< D(x)$.

Sei $w \in R$ der erste Knoten auf diesem Pfad, der nicht mehr zu B gehört (existiert, da $x \notin B$) und sei $v \in B$ der Vorgänger von w auf dem Pfad (existiert, da $u \in B$).

Da der gesamte Pfad ein Gewicht $< D(x)$ hat, muss gelten:

$$\begin{aligned} D(w) &= \min\{D(w') + \gamma(w', w) \mid w' \in B, (w', w) \in E\} \\ &\leq D(v) + \gamma(v, w) < D(x), \end{aligned}$$

was der Auswahl von $x \in R$ widerspricht.

Also gilt $d(u, x) = D(x)$.

Korrektheit von Dijkstras Algorithmus

Zu (4): Seien B', R', U', D' die Werte der Programmvariablen B, R, U, D nach Ausführung der der **while**-Schleife.

Beachte: $B' = B \cup \{x\}$, $D(z) = D'(z)$ für alle $z \in B$ und $D(x) = D'(x)$.

Sei $y \in R'$.

1. Fall: $y \in R \setminus \{x\}$ und $(x, y) \in E$. Dann gilt:

$$\begin{aligned} D'(y) &= \min\{D(y), D(x) + \gamma(x, y)\} \\ &= \min\{\min\{D(z) + \gamma(z, y) \mid z \in B, (z, y) \in E\}, D(x) + \gamma(x, y)\} \\ &= \min\{\min\{D'(z) + \gamma(z, y) \mid z \in B, (z, y) \in E\}, D'(x) + \gamma(x, y)\} \\ &= \min\{D'(z) + \gamma(z, y) \mid z \in B', (z, y) \in E\} \end{aligned}$$

2.Fall $y \in R \setminus \{x\}$ und $(x, y) \notin E$. Dann gilt:

$$\begin{aligned} D'(y) &= D(y) \\ &= \min\{D(z) + \gamma(z, y) \mid z \in B, (z, y) \in E\} \\ &= \min\{D'(z) + \gamma(z, y) \mid z \in B', (z, y) \in E\}. \end{aligned}$$

Korrektheit von Dijkstras Algorithmus

3.Fall: $y \notin R$. Dann gilt $(x, y) \in E$, aber es gibt keine Kante $(z, y) \in E$ mit $z \in B$ (wegen (2)).

Es gilt daher:

$$\begin{aligned} D'(y) &= D(x) + \gamma(x, y) \\ &= D'(x) + \gamma(y, x) \\ &= \min\{D'(z) + \gamma(z, y) \mid z \in B', (z, y) \in E\}. \end{aligned}$$

□

Bemerkung: Dijkstras Algorithmus liefert nicht notwendigerweise ein korrektes Ergebnis, falls negative Kantengewichte erlaubt werden.

Dijkstra mit abstrakten Datentyp für Rand

Um die Laufzeit von Dijkstras Algorithmus zu analysieren, ist es nützlich ihn mit einem abstrakten Datentyp für den Rand R zu formulieren.

Folgende Operationen werden für den Rand R benötigt:

insert	Füge ein neues Element in R ein.
decrease-key	Verringere den Schlüsselwert eines Elements von R (und erhalte die Eigenschaften des Datentyps R).
delete-min	Suche ein Element mit minimalem Schlüsselwert und entferne dieses aus R (und erhalte die Eigenschaften des Datentyps R).

Dijkstra mit abstrakten Datentyp für Rank

$B := \emptyset$; $R := \{u\}$; $U := V \setminus \{u\}$; $p(u) :=$ **nil**; $D(u) := 0$;
while $(R \neq \emptyset)$ **do**
  $x :=$ delete-min(R);
  $B := B \cup \{x\}$;
  forall $(x, y) \in E$ **do**
    if $y \in U$ **then**
      $U := U \setminus \{y\}$; $p(y) := x$; $D(y) := D(x) + \gamma(x, y)$;
      insert$(R, y, D(y))$;
    elsif $y \in R$ **and** $D(x) + \gamma(x, y) < D(y)$ **then**
      $p(y) := x$; $D(y) := D(x) + \gamma(x, y)$;
      decrease-key$(R, y, D(y))$;
    endif
  endfor
endwhile

Laufzeit von Dijkstras Algorithmus

Anzahl der Operationen (n = Anzahl der Knoten, e = Anzahl der Kanten):

insert	n
decrease-key	e
delete-min	n

Die Laufzeit hängt von der für den Rand verwendeten Datenstruktur ab:

1. Arrays der Größe n:
   einzelnes insert/decrease-key: $\mathcal{O}(1)$
   einzelnes delete-min: $\mathcal{O}(n)$
   resultierende Laufzeit: $\mathcal{O}(n + e + n^2) = \mathcal{O}(n^2)$
2. Heap (balancierter Binärbaum der Tiefe $\mathcal{O}(\log(n))$:
   einzelnes insert/decrease-key/delete-min: $\mathcal{O}(\log(n))$
   resultierende Laufzeit: $\mathcal{O}(n \log(n) + e \log(n)) = \mathcal{O}(e \log(n))$.

   Falls $\mathcal{O}(e) \subseteq o(n^2/\log n)$ gilt, ist ein Heap also besser als ein Array.
   Für planare Graphen gilt etwa $e \leq 3n - 6$ für $n \geq 3$

Fibonacci-Heaps (Fredman & Tarjan 1984)

Fibonacci-Heaps schlagen sowohl Arrays als auch Heaps: $\mathcal{O}(e + n \log n)$

Ein Fibonacci-Heap H ist eine List gewurzelter Bäume, also ein Wald.

V ist die Menge der Knoten.

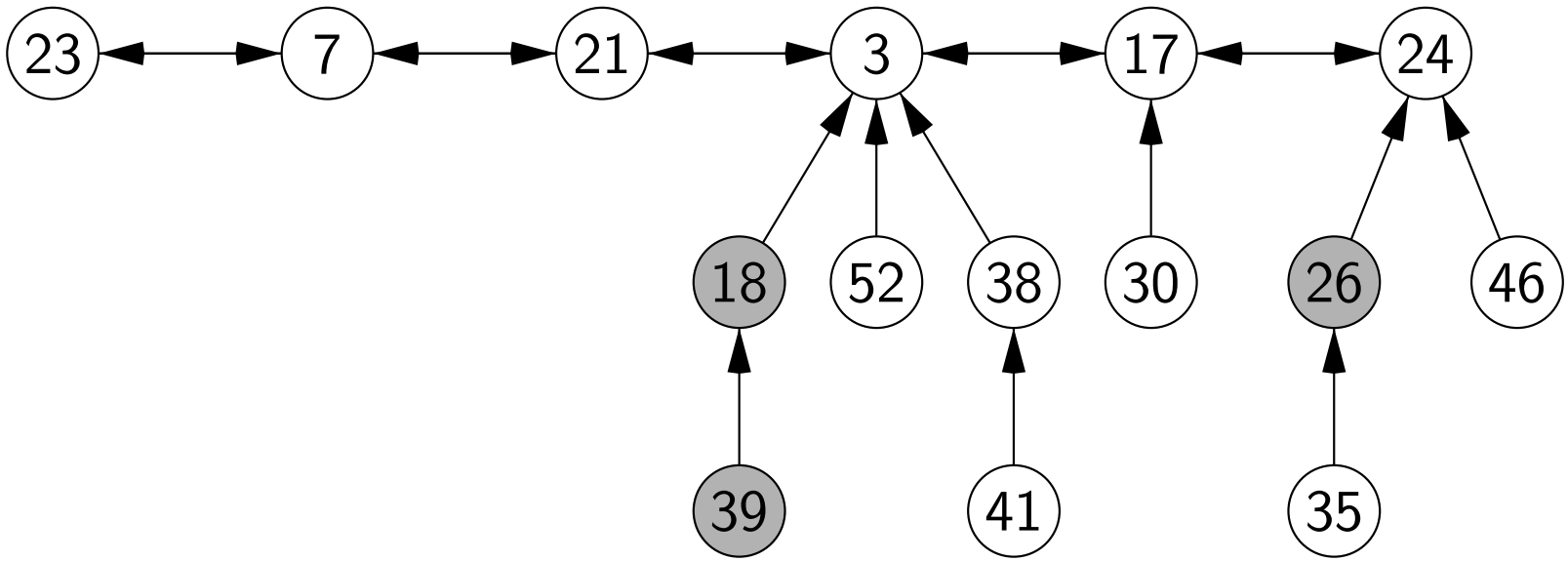
Jeder Knoten $v \in V$ ist mit einem Schlüssel $key(v) \in \mathbb{N}$ markiert.

Heap-Bedingung: $\forall x \in V : \ y$ ist Kind von $x \ \Rightarrow key(x) \leq key(y)$

Einige der Knoten aus V sind markiert. Die Wurzel eins Baums ist niemals markiert.

Beispiel für Fibonacci-Heap

(Schlüssel stehen in den Kreisen, markierte Knoten sind grau)



Fibonacci-Heaps

- Die Elternknoten-Kind Beziehung wird durch Zeiger realisiert, da die Bäume nicht notwendigerweise balanciert sind.
- D.h., Zeigermanipulationen ersetzen die Indexberechnungen bei Standard-Heaps.
- Operationen:
  1. **merge**
  2. **insert**
  3. **delete-min**
  4. **decrease-key**

Implementierung von **merge** und **insert**

- **merge**: Konkatenation zweier Listen — konstante Zeit
- **insert**: Spezialfall von **merge** — konstante Zeit
- **merge** und **insert** erzeugen schließlich lange Listen einelementiger Bäume.
- Jede solche Liste ist ein Fibonacci-Heap.

Implementierung von **delete-min**

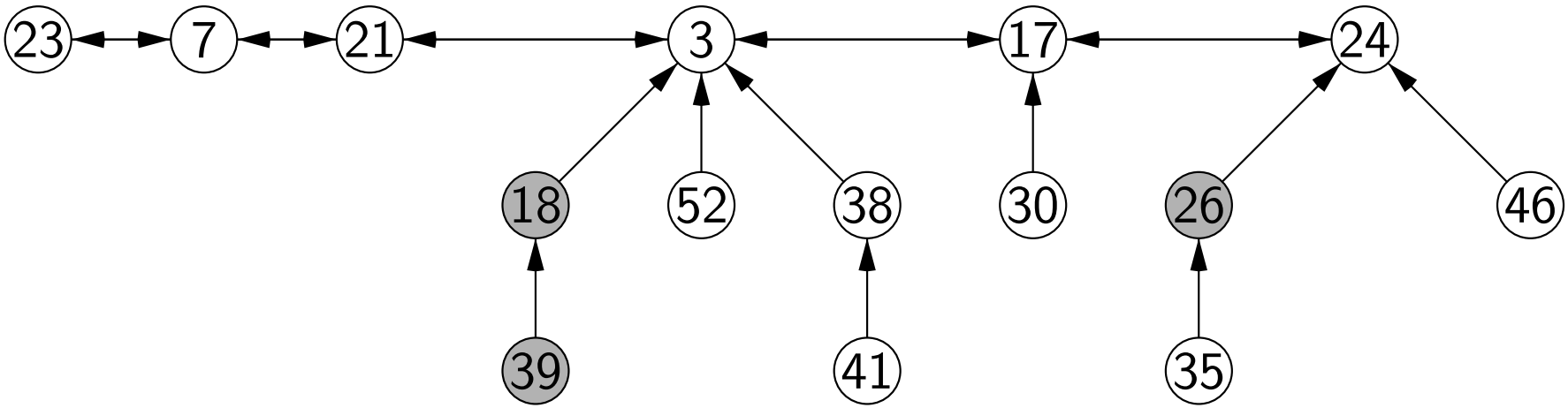
- Sei H ein Fibonacci-Heap bestehend aus T Bäumen und n Knoten.
- Für einen Knoten $x \in V$ sei rank(x) die Anzahl der Kinder von x.
- Für einen Baum B in H sei rank(B) der Rang der Wurzel von B.
- Sei $r_{\max}(n)$ der maximale Rang der in einem Fibonacci-Heap mit n Knoten auftreten kann.
- Natürlich gilt $r_{\max}(n) \leq n$. Später werden wir zeigen, dass $r_{\max}(n) \in \mathcal{O}(\log n)$ gilt.

Implementierung von **delete-min**

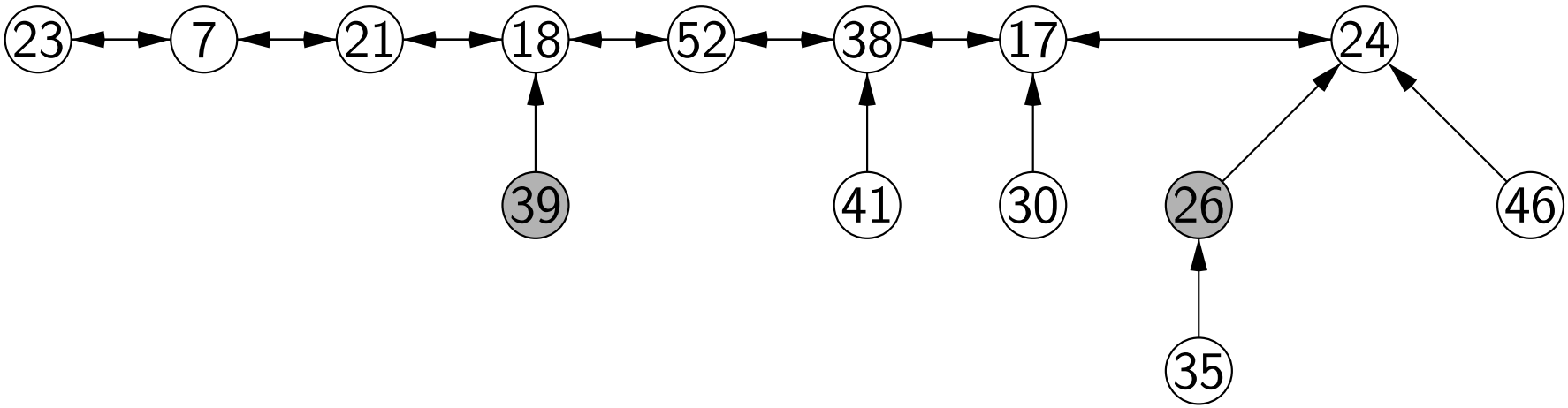
1. Suche die Wurzel x mit minimalen Schlüssel. Zeit: $\mathcal{O}(T)$
2. Entferne x und ersetze den in x gewurzelten Baum durch seine $\text{rank}(x)$ vielen Teilbäume. Entferne eventuelle Markierungen von den neuen Wurzeln. Zeit: $\mathcal{O}(\text{rank}(x)) \subseteq \mathcal{O}(r_{\max}(n))$.
3. Definiere ein Array $L[0, \ldots, r_{\max}(n)]$, wobei $L[i]$ die Liste aller Bäume von Rang i ist. Zeit: $\mathcal{O}(T + r_{\max}(n))$.
4. **for** $i := 0$ **to** $r_{\max}(n) - 1$ **do**
   while $|L[i]| \geq 2$ **do**
   entferne zwei Bäume aus $L[i]$
   mache die Wurzel mit dem größeren Schlüssel zu einem Kind der anderen Wurzel
   füge den so erhaltenen Baum in $L[i+1]$ ein
   endwhile endfor

   Zeit: $\mathcal{O}(T + r_{\max}(n))$

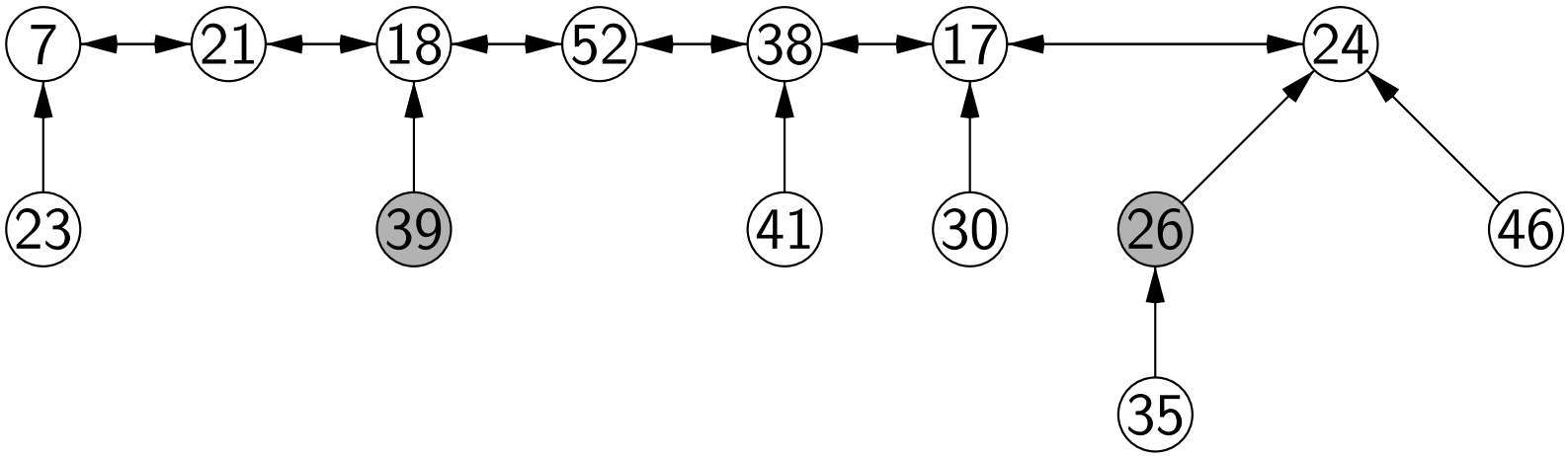
Beispiel für **delete-min**



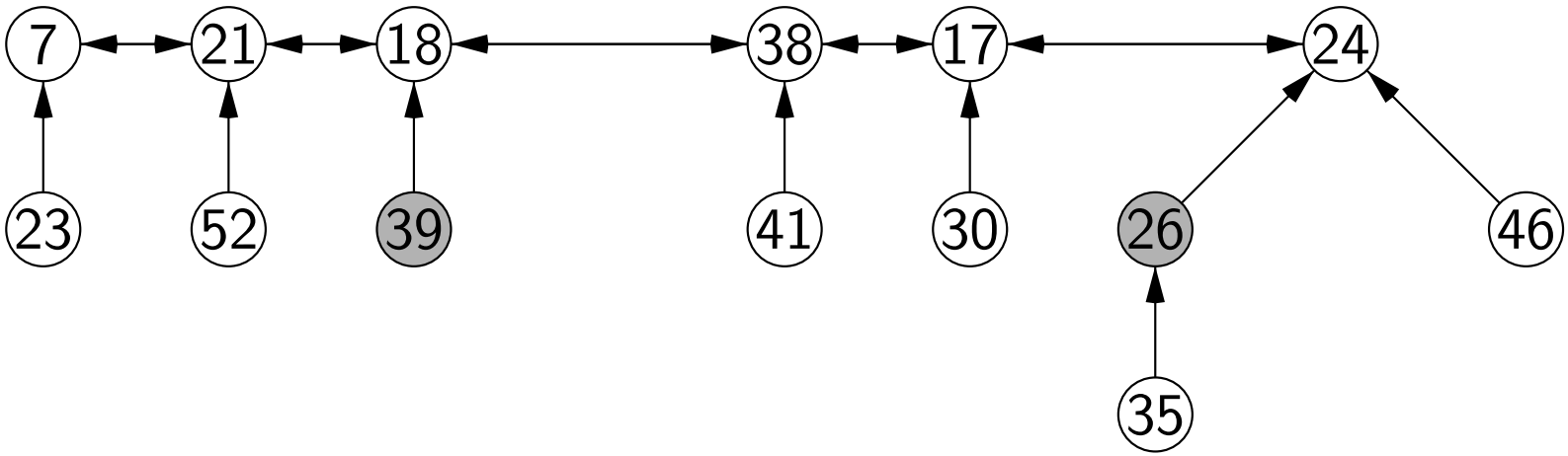
Beispiel für **delete-min**



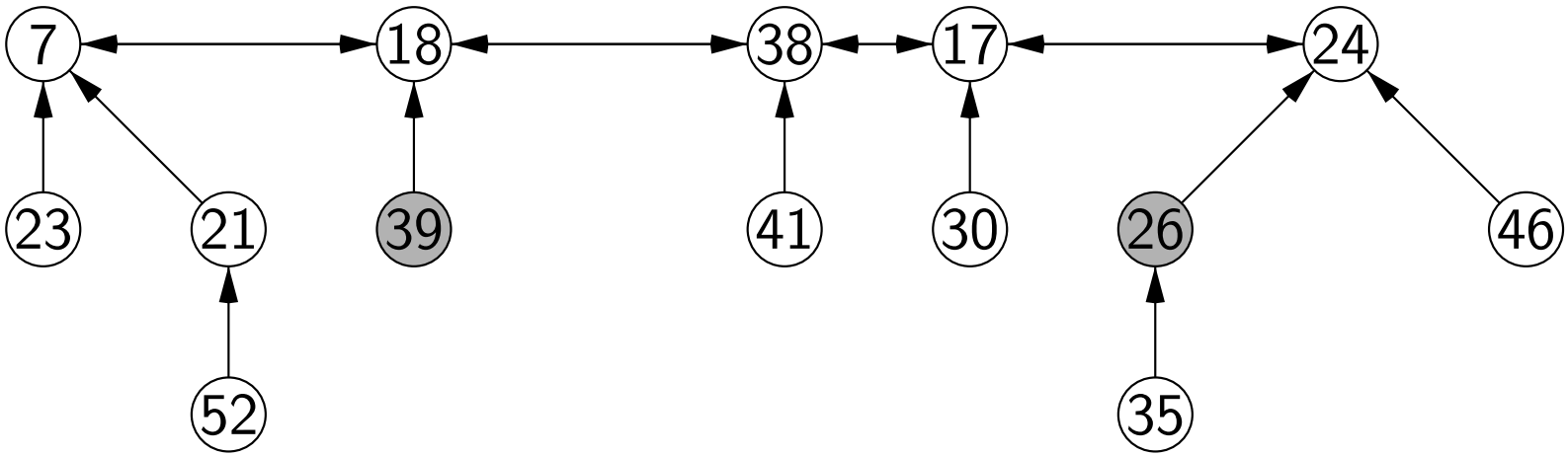
Beispiel für **delete-min**



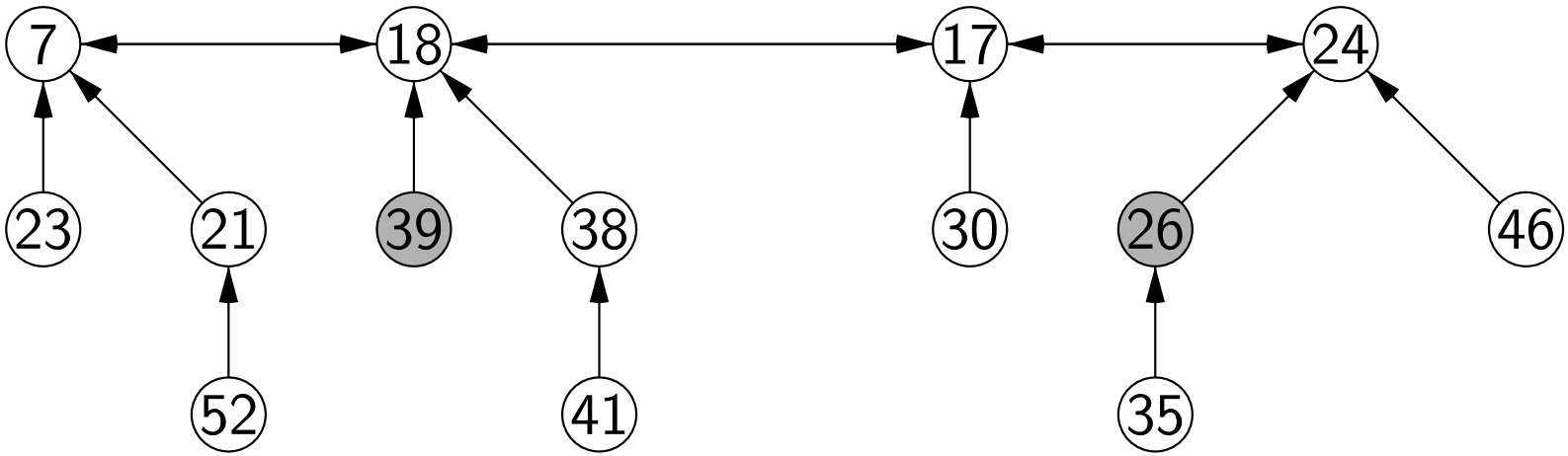
Beispiel für **delete-min**



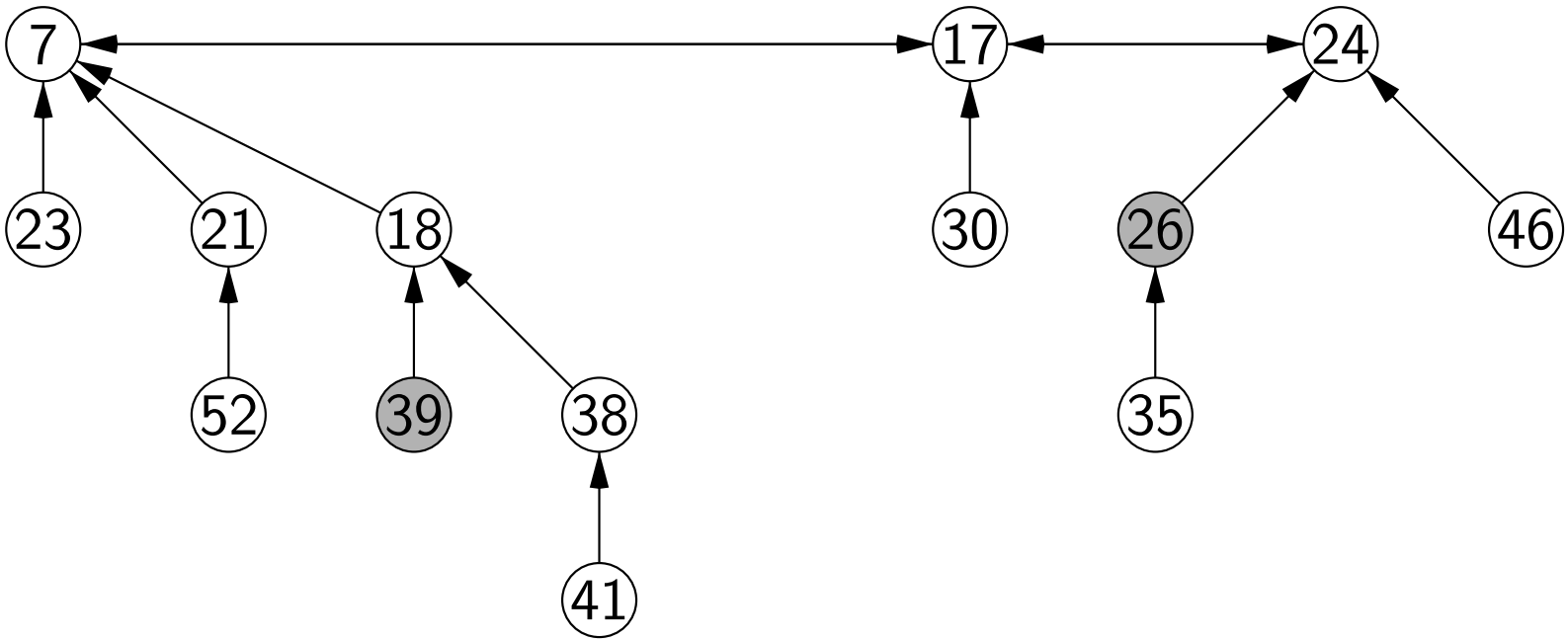
Beispiel für **delete-min**



Beispiel für **delete-min**



Beispiel für **delete-min**



Bemerkungen zu **delete-min**

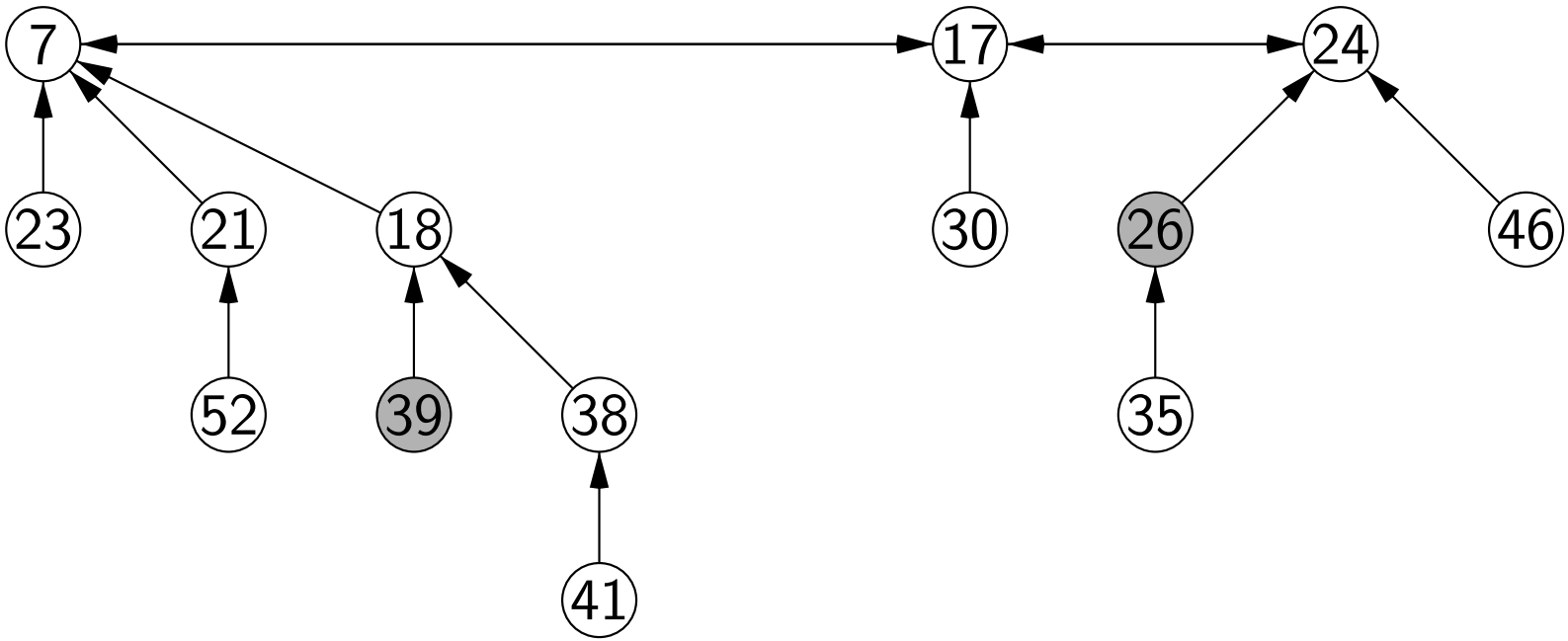
- **delete-min** benötigt Zeit $\mathcal{O}(T + r_{\max}(n))$, wobei T die Anzahl der Bäume vor der Operation ist.
- Nach Ausführung von **delete-min** existiert für jedes $i \leq r_{\max}(n)$ höchstens ein Baum von Rang i.
- Insbesondere ist die Anzahl der Bäume durch $r_{\max}(n)$ beschränkt.

Implementierung von **decrease-key**

Sei x der Knoten, für den der Schlüssel verringert werden soll.

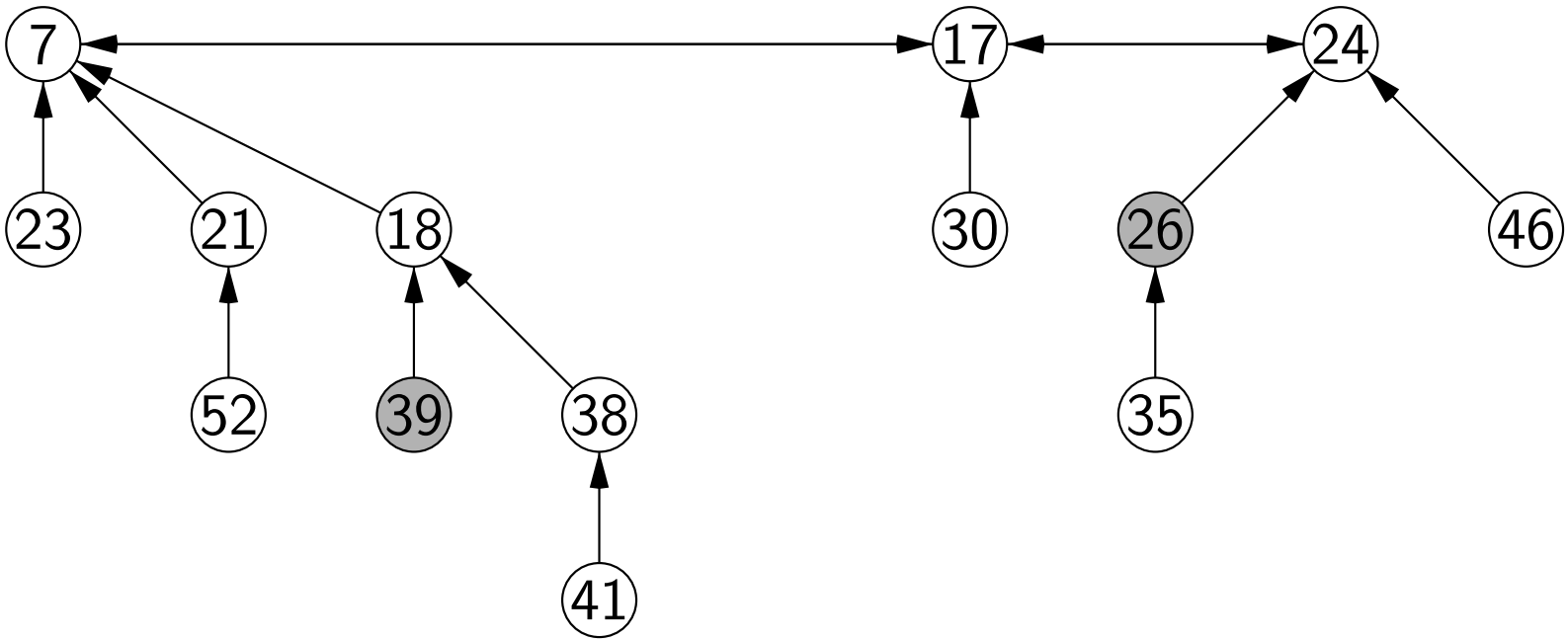
1. Falls x eine Wurzel ist, können wir $key(x)$ ohne sonstige Veränderungen reduzieren. Sei nun x keine Wurzel und sei $x = y_0, y_1, \ldots, y_m$ der Pfad von x zur Wurzel y_m ($m \geq 1$). Sei y_k ($1 \leq k \leq m$) der erste nicht markierte Knoten auf diesem Pfad, der nicht x ist. (beachte: y_m ist nicht markiert).
2. Für alle $0 \leq i < k$ schneiden wir y_i von seinem Elternknoten y_{i+1} ab und entfernen die Markierung von y_i ($y_0 = x$ kann markiert sein). y_i ($0 \leq i < k$) ist dann die (unmarkierte) Wurzel eines neuen Baums.
3. Falls y_k keine Wurzel ist, markieren wir y_k (und speichern auf diese Weise, dass y_k ein Kind verloren hat).

Beispiel zu **decrease-key**



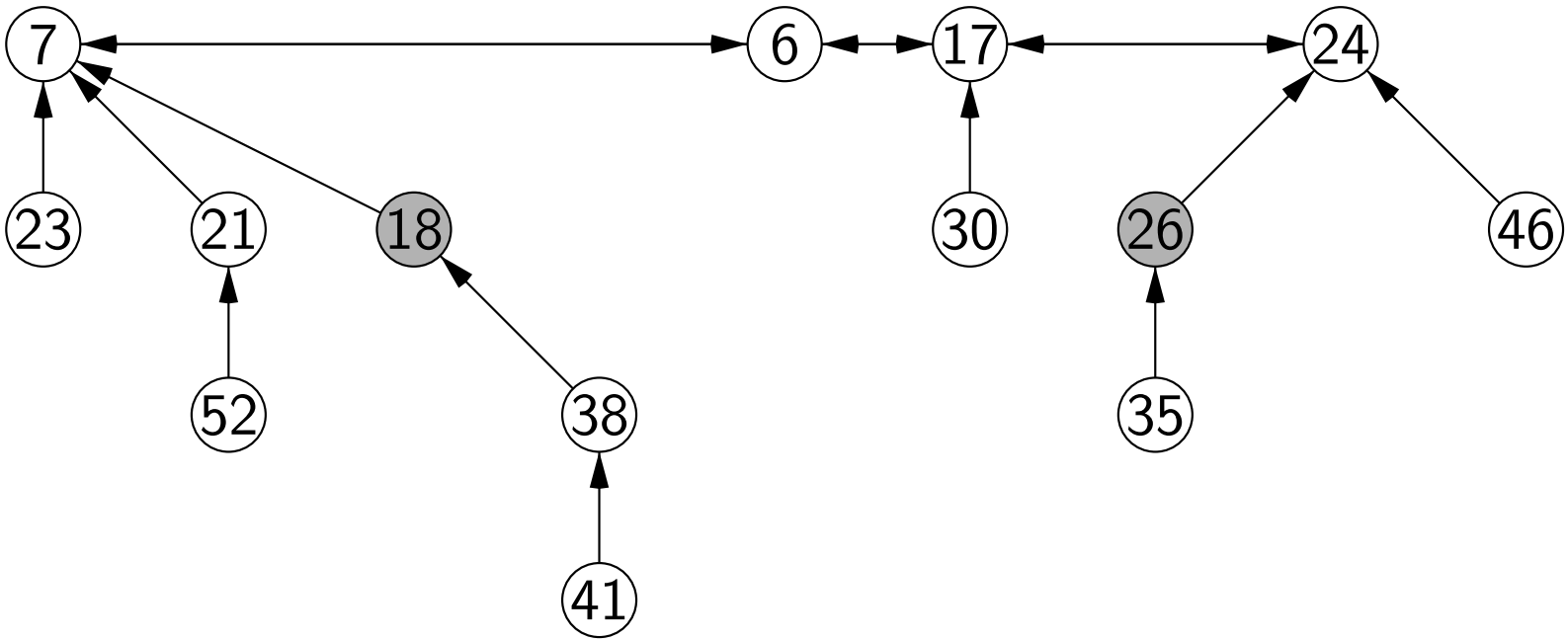
Beispiel zu **decrease-key**

decrease-key(Knoten mit Schlüssel $39, 6$)



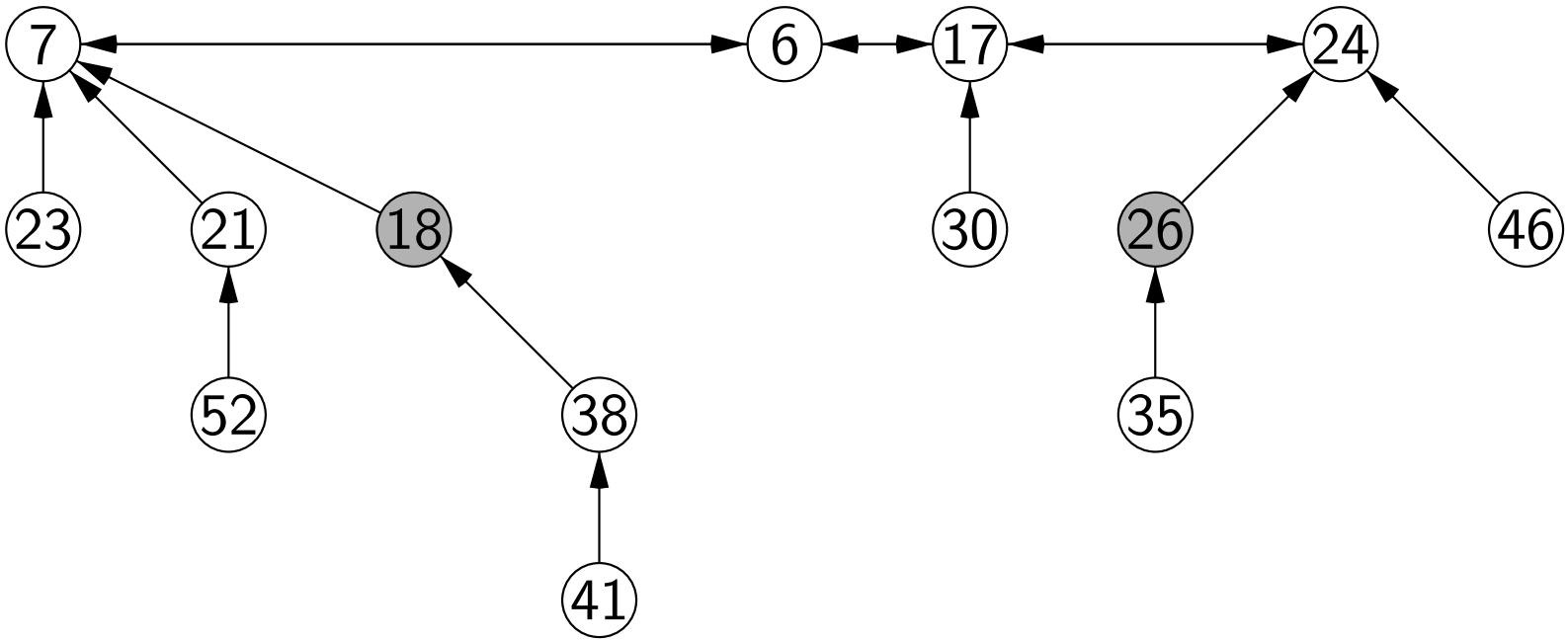
Beispiel zu **decrease-key**

decrease-key(Knoten mit Schlüssel 39, 6)



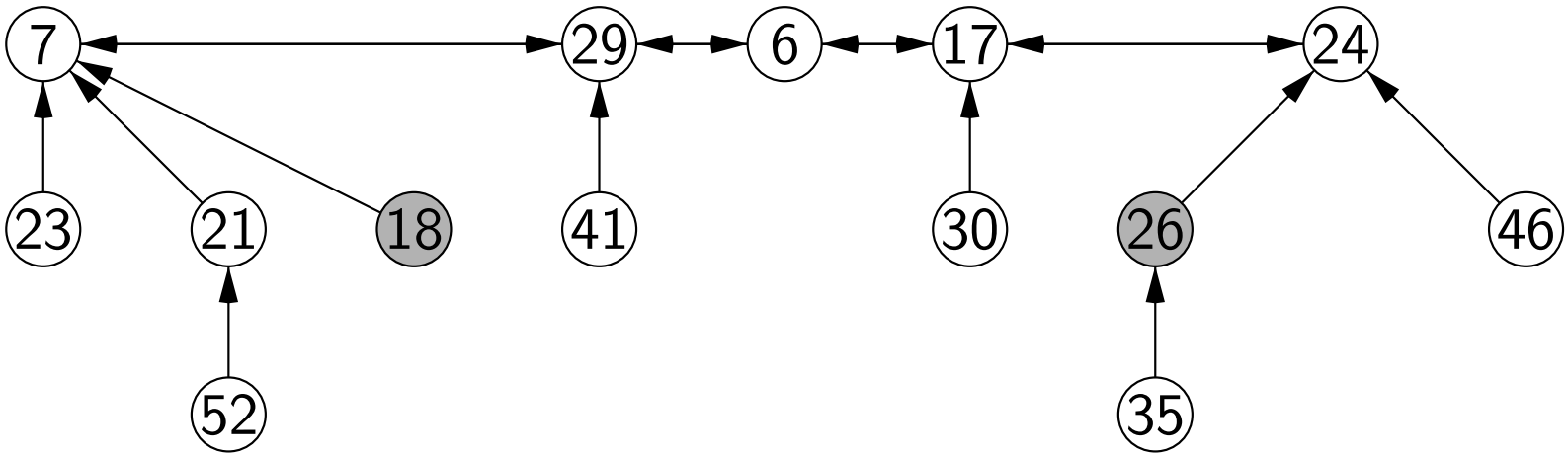
Beispiel zu **decrease-key**

decrease-key(Knoten mit Schlüssel $38, 29$)



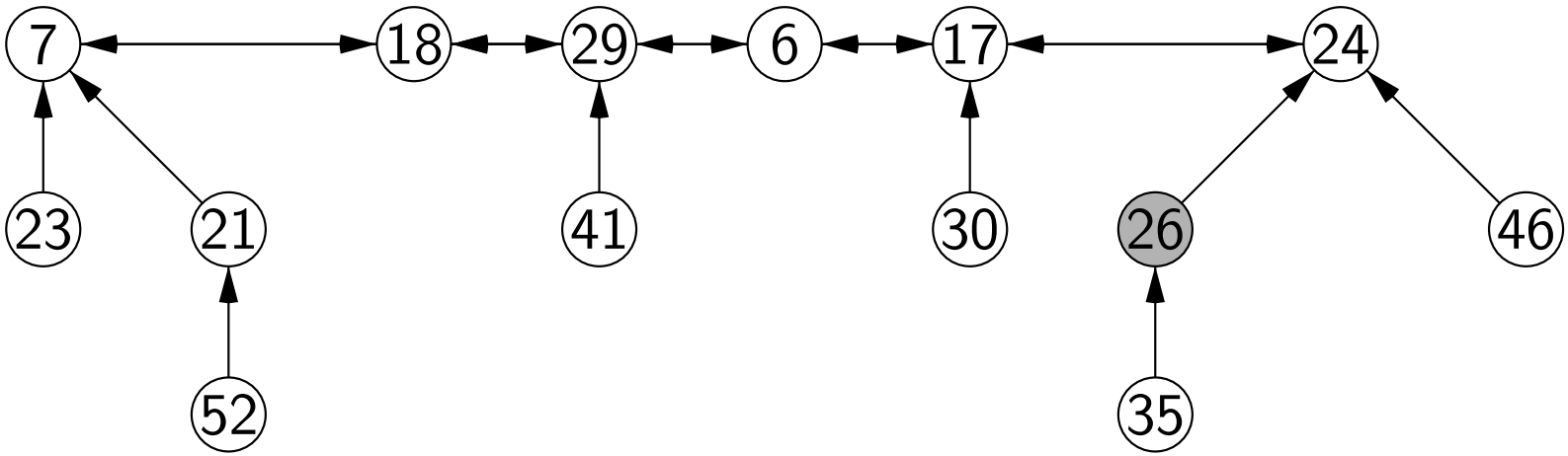
Beispiel zu **decrease-key**

decrease-key(Knoten mit Schlüssel $38, 29$)



Beispiel zu **decrease-key**

decrease-key(Knoten mit Schlüssel 38, 29)



Bemerkungen zu **decrease-key**

- Zeit: $\mathcal{O}(k) + \mathcal{O}(1)$
- **decrease-key** reduziert die Anzahl der markierten Knoten um mindestens $k - 2$ ($k \geq 1$).
- **decrease-key** erhöht die Anzahl der Bäume um k.

Definition von Fibonacci-Heaps

Definition 18

Ein Fibonacci-Heap ist eine Liste von gewurzelten Bäumen wie vorher beschrieben, die aus der leeren Liste mittels beliebig vieler Anwendungen der Operationen **merge**, **insert**, **delete-min**, und **decrease-key** erhalten werden kann.

Lemma 19 (Fibonacci-Heap Lemma)

Sei x ein Knoten in einem Fibonacci-Heap mit $rank(x) = k$

1. *Falls $c_1, \ldots, c_k$ die Kinder von x sind, und c_i vor c_{i+1} ein Kind von x wurde, dann gilt $rank(c_i) \geq i - 2$.*
2. *Der in x gewurzelte Teilbaum enthält mindestens F_{k+1} viele Knoten. Hierbei ist F_{k+1} die $(k+1)$-te Fibonacci-Zahl ($F_0 = F_1 = 1, F_{k+1} = F_k + F_{k-1}$ für $k \geq 1$).*

Beweis des Fibonacci-Heap Lemmas

Teil 1:

Zu dem Zeitpunkt, als c_i ein Kind von x wurde, waren die Knoten $c_1, \ldots, c_{i-1}$ bereits Kinder von x, d.h. der Rang von x war zu diesem Zeitpunkt mindestens $i - 1$.

Da nur Bäume mit gleichen Rang zu einem Baum verschmolzen werden (in **delete-min**), war der Rang von c_i zu diesem Zeitpunkt mindestens $i - 1$.

In der Zwischenzeit konnte Knoten c_i höchstens ein Kind verloren haben: Wenn c_i durch **decrease-key** ein Kind verliert, dann wird c_i markiert, und beim Verlust eines weiteren Kindes würde c_i von seinem Elternknoten x abgetrennt werden.

Daher gilt $\text{rank}(c_i) \geq i - 2$.

Beweis des Fibonacci-Heap Lemmas

Teil 2:

Sei B_k die minimale Anzahl von Knoten in einem Teilbaum von Rang k in einem Fibonacci-Heap. Wir zeigen $B_k \geq F_{k+1}$

Für $k = 0$ und $k = 1$ ist dies wahr: $B_0 \geq 1 = F_1$, $B_1 \geq 2 = F_2$.

Sei nun x ein Knoten mit $\text{rank}(x) = k + 1$ und B_{k+1} vielen Knoten in dem in x gewurzelten Teilbaum.

Beachte: $j \geq i \Rightarrow B_j \geq B_i$.

Seien $c_1, \ldots, c_{k+1}$ die Kinder von x sortiert nach abnehmenden Alter.

Aus Teil 1 des Lemmas folgt:

$$B_{k+1} \geq 2 + \sum_{i=2}^{k+1} B_{i-2} \geq 2 + \sum_{i=2}^{k+1} F_{i-1} = 2 + \sum_{i=1}^{k} F_i = F_{k+2}$$

Wachstum der Fibonacci-Zahlen

Theorem 20

Für $k \geq 0$ gilt:

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

Die Fibonacci-Zahlen wachsen also exponeniell.

Konsequenz: $r_{\max}(n) \in \mathcal{O}(\log n)$.

Zusammenfassung der Laufzeiten

- **merge**, **insert**: konstante Zeit
- **delete-min**: $\mathcal{O}(T + r_{\max}(n)) \subseteq \mathcal{O}(T + \log n)$, wobei T die aktuelle Zahl der Bäume ist.
- **decrease-key**: $\mathcal{O}(1) + \mathcal{O}(k)$ $(k \geq 1)$, wobei mindestens $k - 2$ Markierungen aus dem Fibonacci-Heap entfernt werden.

Amortisierte Zeiten

Definition 21

Für einen Fibonacci-Heap H definieren wir sein Potential $pot(H)$ als $pot(H) := T + 2M$, wobei T die Anzahl der Bäume ist und M die Anzahl der markierten Knoten ist.
Für eine Operation op sei $\Delta_{pot}(op)$ die Differenz aus dem Potential nach und vor der Ausführung der Operation.

$$\Delta_{pot}(op) = pot(\text{Heap nach } op) - pot(\text{Heap bevor } op).$$

Die amortisierte Zeit der Operation op ist

$$t_{amort}(op) = t(op) + \Delta_{pot}(op).$$

Amortisierte Zeiten

Das Potential hat folgende Eigenschaften:

- $pot(H) \geq 0$
- $pot(H) \in \mathcal{O}(|H|)$
- $pot(nil) = 0$

Sei nun $op_1, op_2, op_3, \ldots, op_m$ eine Folge von m Operationen, die auf einen zu Begin leeren Fibonacci-Heap angewendet werden.
Dann gilt:

$$\sum_{i=1}^{m} t(op_i) \leq \sum_{i=1}^{m} t_{amort}(op_i)\,.$$

Bemerkung: Die Differenz zwischen den beiden Summen ist das Potential des erzeugten Fibonacci-Heaps.
Daher genügt es, $t_{amort}(op)$ abzuschätzen.

Amortisierte Zeiten

Konvention:
Indem wir alle Terme in den folgenden Berechnungen mit einer geeigneten Konstante multiplizieren, können wir annehmen, dass **merge** und **insert** nur einen Zeitschritt benötigen, dass **delete-min** $T + \log n$ Zeitschritte benötigt, und dass **decrease-key** $k + 1$ Zeitschritte benötigt. Dies erlaubt es uns die $\mathcal{O}$-Notation zu vermeiden.

Amortisierte Zeiten

- $t_{amort}(\textbf{merge}) = t(\textbf{merge}) = 1$, denn das Potential der Konkatenation der beiden Teillisten ist die Summe der Potentiale der beiden Teillisten.
- $t_{amort}(\textbf{insert}) = t(\textbf{insert}) + \Delta_{pot}(op) = 1 + 1 = 2$.
- Für **delete-min** gilt $t(\textbf{delete-min}) \leq T + \log n$, wobei T die Anzahl der Bäume vor der Anwendung von **delete-min** ist.
  Nach **delete-min** ist die Anzahl der Bäume durch $r_{\max}(n)$. beschränkt.
  Die Anzahl der markierten Knoten kann nur abnehmen.
  Also gilt $\Delta_{pot}(op) \leq r_{\max}(n) - T$ und
  $t_{amort}(\textbf{delete-min}) \leq T + \log n - T + r_{\max}(n) \in \mathcal{O}(\log n)$.

Amortisierte Zeiten

- Für **decrease-key** gilt $t(\textbf{decrease-key}) \leq k + 1$ $(k \geq 1)$, wobei mindestens $k - 2$ Markierungen entfernt werden. Außerdem werden k neue Bäume zum Fibonacci-Heap hinzugefügt.
  Also gilt:

$$\begin{aligned} \Delta_{pot}(op) &= \Delta(T) + 2\Delta(M) \\ &\leq k + 2 \cdot (2 - k) \\ &= 4 - k, \end{aligned}$$

  und damit $t_{amort}(\textbf{decrease-key}) \leq k + 1 + 4 - k = 5 \in \mathcal{O}(1)$.

Amortisierte Zeiten

Theorem 22

Für einen Fibonacci-Heap gelten die folgenden amortisierten Zeitschranken:

$$t_{amort}(\textbf{merge}) \in \mathcal{O}(1)$$
$$t_{amort}(\textbf{insert}) \in \mathcal{O}(1)$$
$$t_{amort}(\textbf{delete-min}) \in \mathcal{O}(\log n)$$
$$t_{amort}(\textbf{decrease-key}) \in \mathcal{O}(1)$$

Fibonacci-Heaps für Dijkstra

Zurück zu Dijkstras Algorithmus:

Für Dijkstras Algorithm sei V der Rand, und $key(v)$ sei die aktuelle Abschätzung für $d(u, v)$.

Sei n die Anzahl der Knoten und e die Anzahl der Kanten des Eingabegraphens.

Dann werden höchstens n **insert**-, e **decrease-key**- und n **delete-min**-Operationen ausgeführt.

Fibonacci-Heaps für Dijkstra

$$
\begin{aligned}
t_{\text{Dijkstra}} \quad &\leq \quad n \cdot t_{amort}(\textbf{insert}) \\
&\quad + \quad e \cdot t_{amort}(\textbf{decrease-key}) \\
&\quad + \quad n \cdot t_{amort}(\textbf{delete-min}) \\
&\in \quad \mathcal{O}(n + e + n \log n) \\
&= \quad \mathcal{O}(e + n \log n)
\end{aligned}
$$

Zur Erinnerung:

- Mit Arrays haben wir $t_{\text{Dijkstra}} \in \mathcal{O}(n^2)$ erhalten.
- Mit Standard-Heaps haben wir $t_{\text{Dijkstra}} \in \mathcal{O}(e \log(n))$ erhalten.

Partitionen

Erinnerung: Im Algorithmus von Kruskal müssen wir eine Partition der Knotenmenge verwalten.

Definitionen

Eine Partition $\mathcal{P}$ einer Menge M ist eine Teilmenge $\mathcal{P} \subseteq 2^M \setminus \{\emptyset\}$, so dass gilt:

$$M = \bigcup_{A \in \mathcal{P}} A \text{ und}$$

$$\forall A, B \in \mathcal{P} : A \cap B \neq \emptyset \Rightarrow A = B$$

Union-Find-Datenstrukturen

Sei $\mathcal{P}$ eine Partition von M. Für alle $a \in M$ sei $[a]$ die eindeutige Menge $A \in \mathcal{P}$ mit $a \in A$.

Eine Union-Find-Datenstruktur unterstützt die folgenden Operationen:

- **find**(a) für $a \in M$ liefert ein kanonisches Element von $[a]$. Wir setzen voraus, dass zu jedem Zeitpunkt **find**$(a) =$ **find**(b) gilt, genau dann, wenn $[a] = [b]$.
- **union**(a, b) für $a, b \in M$ ersetzt die Partition $\mathcal{P}$ durch

$$\mathcal{P} \setminus \{[a], [b]\} \cup \{[a] \cup [b]\}.$$

Union-Find mit Arrays

Eine einfache Implementierung, die Arrays benutzt:

a	1	2	3	4	5	6	7	8	9	10	11	12
$\mathrm{find}(a)$	1	2	3	3	1	2	1	3	1	2	1	3

$\mathrm{union}(2, 4)$ liefert:

a	1	2	3	4	5	6	7	8	9	10	11	12
$\mathrm{find}(a)$	1	2	2	2	1	2	1	2	1	2	1	2

Union-Find mit Arrays

Für ein Array mit n Elementen erhalten wir die folgenden Laufzeiten:

- **find:** $\mathcal{O}(1)$.
- **union:** $\mathcal{O}(n)$.

In vielen Anwendungsfällen (z.B. Kruskal) liegt die Anzahl der union-Aufrufe in $\Theta(n)$.

Dies ergibt eine Gesamtlaufzeit von $\mathcal{O}(n^2)$ für Arrays.

Union-Find mit Bäumen

Union-find mit Bäumen: Sei M eine endliche Menge und $\mathcal{P}$ eine Partition von M.

Eine Menge $A \in \mathcal{P}$ wird als ein Wurzelbaum mit Knotenmenge A repräsentiert.

Für $a \in M$ liefert **find**(a) die Wurzel des eindeutigen Baums, der a enthält.

Für jedes Element $a \in M$ werden die folgenden Eigenschaften festgehalten:

$\text{pred}(a) \in M \cup \{\text{nil}\}$:	Der Vaterknoten von a oder nil, falls a eine Wurzel ist.
$\text{count}(a) \in \mathbb{N}$:	Die Anzahl der Knoten des Teilbaums mit Wurzel a, falls a eine Wurzel ist.

Union-Find mit Bäumen

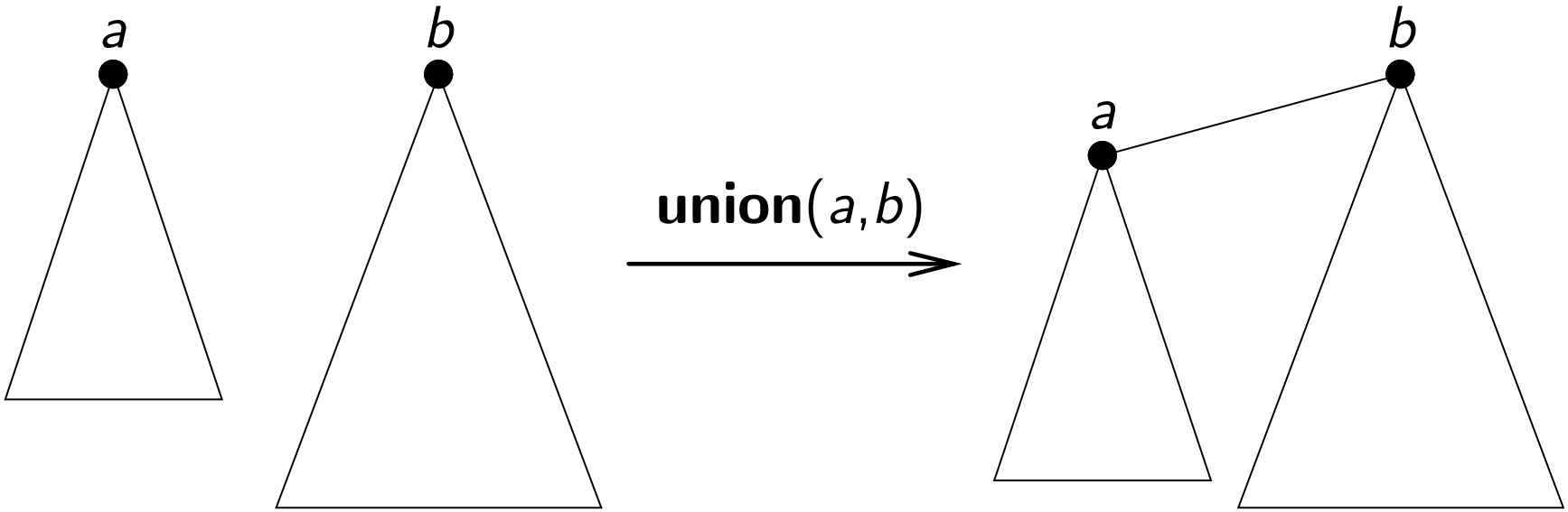
Zu Anfang: **pred**$(a) :=$ nil und **count**$(a) := 1$ für alle $a \in M$.

Implementierung von **union**: **union**(a, b) wird nur auf zwei Knoten $a, b \in M$ angewandt, die Wurzeln sind:

union(a, b)
if count$(a) >$ count(b) **then**
    pred$(b) := a$; count$(a) :=$ count$(a) +$ count(b);
else
    pred$(a) := b$; count$(b) :=$ count$(a) +$ count(b);
endfi

Der kleinere Baum wird also ein Teilbaum des größeren.

Union-Find mit Bäumen



Union-Find mit Bäumen

Wir werden sehen, dass für eine Menge M mit n Elementen die Höhe jedes Baums durch $\log(n)$ beschränkt ist.

D.h. ein Find-Aufruf benötigt $\mathcal{O}(\log(n))$ Zeit.

Ein Union-Aufruf benötigt $\mathcal{O}(1)$ Zeit.

Pfad-Kompression ist eine weitere Idee zum Beschleunigen der Find-Aufrufe:

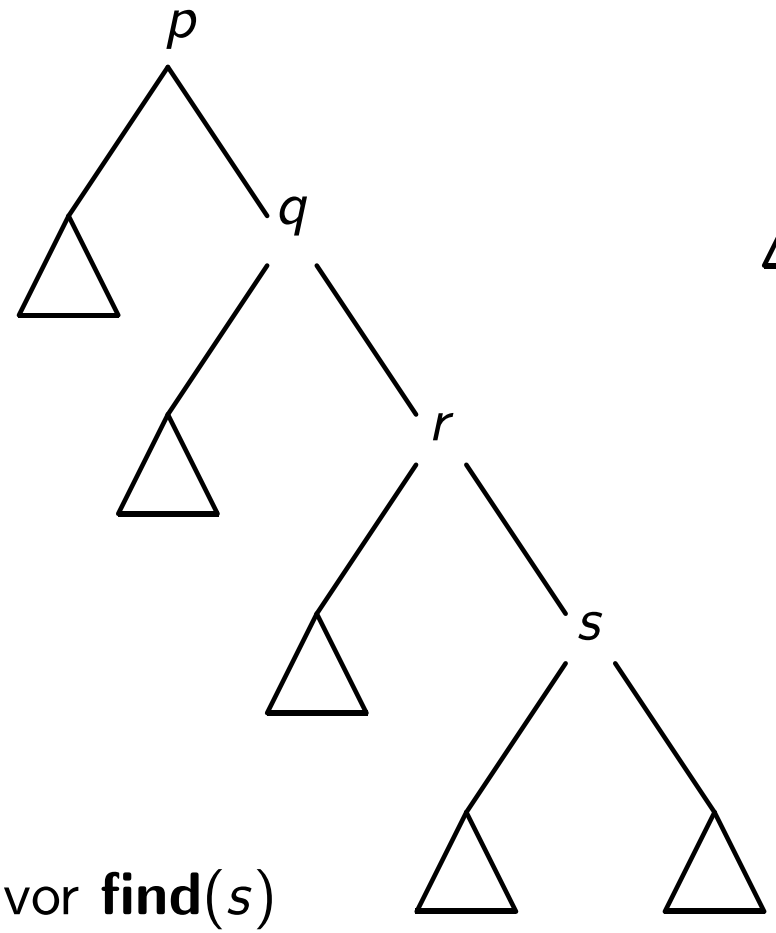
Nach dem Aufruf von **find**(a) setzt man den Vorgänger-Pointer jedes Knoten entlang des Pfads von a zum Wurzelknoten $r =$ **find**(a) auf r.

Bemerkung: Dazu muss der Pfad von a nach r zweimal durchlaufen werden.

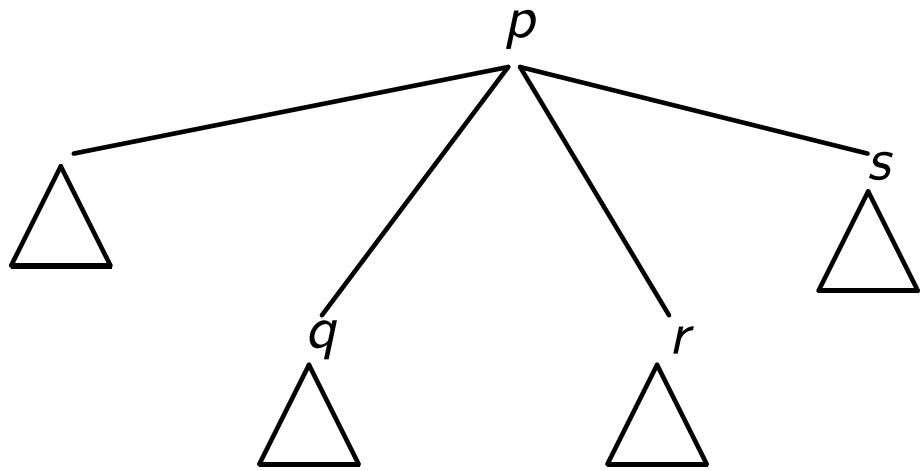
Pfad-Kompression

```
function find(a: node): node;
var b, root: nodes;
begin
   b := a;
   while pred(b) ≠ nil do
      b := pred(b);
   endwhile
   root := b;
   while pred(a) ≠ nil do
      b :=pred(a);
      pred(a) := root;
      a := b
   endwhile
   return root;
endfunction
```

Pfad-Kompression



Baum vor **find**(s)



Baum nach **find**(s)

Äquivalenz von DEAs

Weitere Anwendungen von Union-Find:
Äquivalenz von deterministischen endlichen Automaten

Definition

Ein deterministischer endlicher Automat (DEA) über dem Alphabet Σ ist ein 5-Tupel $A = (Q, \Sigma, \delta, q_0, F)$ mit:

- Q ist die Menge der Zustände,
- $q_0 \in Q$ ist der Startzustand,
- $\delta : Q \times \Sigma \longrightarrow Q$ ist die Übergangsfunktion,
- $F \subseteq Q$ ist die Menge der Endzustände.

Äquivalenz von DEAs

Für $q \in Q$ und $w \in \Sigma^*$ wird der Zustand $\delta(q, w)$ mit qw bezeichnet.

$L(q) := \{w \in \Sigma^* \mid qw \in F\}$

$L(A) := L(q_0)$

Äquivalenzproblem:

Eingabe: DEAs $A = (Q, \Sigma, \delta, q_0, F)$ und $A' = (Q', \Sigma, \delta', q_0', F')$.
Frage: $L(A) = L(A')$?

Äquivalenz von DEAs

O.E.d.A. wird angemommen, dass $Q \cap Q' = \emptyset$. Sei $\widetilde{Q} = Q \cup Q'$.

Sei $R \subseteq \widetilde{Q} \times \widetilde{Q}$ die kleinste (bezüglich Teilmengenbeziehung) Äquivalenzrelation, so dass

1. $(q_0, q_0') \in R$,
2. $(q, q') \in R$, $a \in \Sigma, q \in Q, q' \in Q' \implies (qa, q'a) \in R$

Lemma 23

$L(A) = L(A')$ genau dann, wenn

$$R \cap \left[(F \times (Q' \setminus F')) \cup ((Q \setminus F) \times F')\right] = \emptyset.$$

Beweis: Übung

Äquivalenz von DEAs

function equivalence-dfa(A, A': dfa) : **boolean**
begin
  $\mathcal{L} := \{(q_0, q_0')\}$;
  while $\mathcal{L} \neq \emptyset$ **do**
    Wähle ein Paar $(q, q') \in \mathcal{L}$; $\mathcal{L} := \mathcal{L} \setminus \{(q, q')\}$;
    $r :=$ **find**(q); $r' :=$ **find**(q')
    if $r \neq r'$ **then**
      if $(q, q') \in [(F \times (Q' \setminus F')) \cup ((Q \setminus F) \times F')]$ **then**
        return false
      else
        union(r, r');
        forall $a \in \Sigma$ **do**
          $\mathcal{L} := \mathcal{L} \cup \{(qa, q'a)\}$ **endfor endif endif endwhile**
  return true endfunction

Äquivalenz von DEAs

Die maximal mögliche Anzahl von Union-Aufrufen ist $|Q| + |Q'| =: n = |\widetilde{Q}|$.

Also werden höchstens $|\Sigma| \cdot n$ Elemente zu $\mathcal{L}$ hinzugefügt.

In jedem Durchlauf der While-Schleife wird ein Element aus $\mathcal{L}$ entfernt.

Der Algorithmus terminiert also nach höchstens $|\Sigma| \cdot n$ Durchläufen der While-Schleife.

Zusammenfassung: $\leq n$ Union-Aufrufe, $\leq 2|\Sigma| \cdot n$ Find-Aufrufe

Ackermann-Funktion

Analyse von Tarjan (1983) mit der Ackermann-Funktion.

Wir benutzen die folgende Definition für die Ackermann-Funktion:

$$A_0(x) = x + 1,$$
$$A_{k+1}(x) = A_k^x(x) \quad (A_k \text{ wird } x \text{ mal auf } x \text{ angewendet})$$

Sei $\alpha(n) = \min\{k \mid A_k(2) > n\}$ (inverse Ackermann-Funktion).

Es gilt $A_2(x) = x2^x$, $A_3(2) = 2^{11} = 2048$, also
$A_4(2) = A_3^2(2) = A_3(2048) = A_2^{2048}(2048)$.

Diese Zahl ist größer als ein Turm von $11 + 2048$ Potenzen von 2.

Also gilt, dass $\alpha(n) \leq 4$ für alle praktischen Werte von n.

Rang eines Knotens

Wir betrachten eine Folge von m Union/Find-Aufrufen $\mathrm{op}_1 \mathrm{op}_2 \cdots \mathrm{op}_m$ auf einer Menge mit n Knoten.

Definition 24

"Zeit t" wird nach dem Ausführen von $\mathrm{op}_1 \cdots \mathrm{op}_t$ $(0 \leq t \leq m)$ erreicht.

Für einen Knoten a sei $T_t(a)$ der Baum mit der Wurzel a zum Zeitpunkt t, falls Find-Aufrufe ohne Pfad-Kompression durchgeführt werden.

Sei $\mathrm{rank}(a) = 2 + \mathrm{height}(T_m(a))$ (2+ die Länge des längsten Pfades von a zu einem Blatt in $T_m(a)$).

Lemma 25 (Monotonie von Rängen)

Falls ein Knoten a irgendwann zum Vorgänger eines anderen Knoten b wird (mit oder ohne Pfad-Kompression), dann gilt $\mathrm{rank}(a) > \mathrm{rank}(b)$.

Bäume sind balanciert

Lemma 26 (Bäume sind balanciert)

$|T_t(a)| \geq 2^{height(T_t(a))}$

Beweis: Induktion über t:

$t = 0$: Alle Bäume haben Höhe 0 und Größe 1.

$t > 0$: Falls $\text{height}(T_t(a)) = \text{height}(T_{t-1}(a))$:

$$|T_t(a)| \geq |T_{t-1}(a)| \geq 2^{\text{height}(T_{t-1}(a))} = 2^{\text{height}(T_t(a))}.$$

Sei nun $\text{height}(T_t(a)) > \text{height}(T_{t-1}(a))$.

Dann muss es eine Wurzel b geben mit $\text{op}_t = \textbf{union}(a, b)$, d.h. b wird neues Kind von a.

Bäume sind balanciert

Darüber hinaus gilt $|T_{t-1}(a)| \geq |T_{t-1}(b)|$ (kleinere Unterbäume werden in größere eingefügt) und

$$\text{height}(T_t(a)) = 1 + \text{height}(T_t(b)) = 1 + \text{height}(T_{t-1}(b)).$$

Mit der Induktionsvoraussetzung für b erhalten wir:

$$\begin{aligned} |T_t(a)| &= |T_{t-1}(a)| + |T_{t-1}(b)| \\ &\geq 2 \cdot |T_{t-1}(b)| \\ &\geq 2^{1+\text{height}(T_{t-1}(b))} \\ &= 2^{\text{height}(T_t(a))} \end{aligned}$$

Ränge sind klein

Lemma 27 (Ränge sind klein)

$rank(a) \leq 2 + \lfloor \log(n) \rfloor$

Beweis: Aus Lemma 26 erhalten wir für jeden Knoten a:

$$n \geq |T_m(a)| \geq 2^{\text{height}(T_m(a))} = 2^{\text{rank}(a)-2}$$

Also $\text{rank}(a) \leq 2 + \lfloor \log(n) \rfloor$. □

Wenige Knoten mit großem Rang

Lemma 28 (Wenige Knoten mit großem Rang)

Für alle $r \geq 2$ gilt $|\{a \mid rank(a) = r\}| \leq \frac{n}{2^{r-2}}$.

Beweis: Falls $\text{rank}(a) = \text{rank}(b)$, dann müssen $T_m(a)$ und $T_m(b)$ disjunkt sein (Lemma 25).

Also gilt:

$$\begin{aligned} n &\geq \sum_{\text{rank}(a)=r} |T_m(a)| \\ &\geq \sum_{\text{rank}(a)=r} 2^{r-2} \quad \text{(Lemma 26)} \\ &= 2^{r-2} \cdot |\{a \mid \text{rank}(a) = r\}|, \end{aligned}$$

womit das Lemma bewiesen ist. □

Die Funktion δ_t

Sei a ein Knoten, der zum Zeitpunkt t keine Wurzel ist.

$\mathrm{par}_t(a)$ ist der Vaterknoten von a zum Zeitpunkt t (mit Pfad-Kompression).

$\delta_t(a) := \max\{k \in \mathbb{N} \mid \mathrm{rank}(\mathrm{par}_t(a)) \geq A_k(\mathrm{rank}(a))\}$.

Lemma 25 impliziert $\mathrm{rank}(\mathrm{par}_t(a)) \geq \mathrm{rank}(a) + 1 = A_0(\mathrm{rank}(a))$, also ist $\delta_t(a)$ definiert.

Lemma 29 (δ_t ist klein)

Falls $n \geq 5$, dann $\delta_t(a) < \alpha(n)$ (inverse Ackermann-Funktion).

Beweis: Falls $n \geq 5$ und $\delta_t(a) = k$, dann

$$\begin{aligned} n &> \lfloor \log(n) \rfloor + 2 \\ &\geq \mathrm{rank}(\mathrm{par}_t(a)) \quad \text{(Lemma 27)} \\ &\geq A_k(\mathrm{rank}(a)) \geq A_k(2) \end{aligned}$$

Gold und Eisen

Laufzeitanalyse:

Jeder Union-Aufruf benötigt Zeit $\mathcal{O}(1)$, also ist die Gesamtzeit für alle Union-Aufrufe $\mathcal{O}(m)$.

Für die Find-Aufrufe wird folgende Strategie benutzt:

Jedes Mal, wenn ein Knoten a während eines Find-Aufrufs besucht wird, erhält er entweder ein Stück Gold oder ein Stück Eisen.

Am Ende werden die Gold- und Eisenstücke gezählt.

Falls ein Knoten a während eines Find-Aufrufs zum Zeitpunkt t besucht wird, erhält er

- ein Stück Eisen, falls ein Vorgänger b von a (auf dem Pfad zum Wurzelknoten) mit $\delta_t(a) = \delta_t(b)$ existiert.
- ein Stück Gold sonst.

Begrenzen des Golds

- Wegen Lemma 29 gibt es höchstens $\alpha(n)$ verschiedene Werte für $\delta_t(a)$.
- D.h. beim Hochlaufen eines Pfades von einem Knoten a zum Wurzelknoten des Baums, in dem a liegt, werden höchstens $\alpha(n)$ Goldstücke vergeben.
- D.h. es werden höchstens $2\alpha(n)$ Goldstücke während eines Find-Aufrufs vergeben.
- Am Ende sind höchstens $2m\alpha(n)$ Goldstücke vergeben.

Begrenzen des Eisens

Annahme: a erhält Eisen zum Zeitpunkt t.

a muss also einen Vorgänger b haben, sodass $\delta_t(a) = \delta_t(b) = k$, d.h.

$$\text{rank}(\text{par}_t(a)) \geq A_k(\text{rank}(a))$$
$$\text{rank}(\text{par}_t(b)) \geq A_k(\text{rank}(b))$$

Also existiert ein $i \geq 1$ mit $\text{rank}(\text{par}_t(a)) \geq A_k^i(\text{rank}(a))$.

Sei r die Wurzel des Baums, der a und b enthält.

$$\begin{aligned}\text{rank}(r) &\geq \text{rank}(\text{par}_t(b)) \\ &\geq A_k(\text{rank}(b)) \\ &\geq A_k(\text{rank}(\text{par}_t(a))) \\ &\geq A_k(A_k^i(\text{rank}(a))) = A_k^{i+1}(\text{rank}(a))\end{aligned}$$

Begrenzen des Eisens

Da $r = \text{par}_{t+1}(a)$, gilt zum Zeitpunkt $t+1$:

$$\text{rank}(\text{par}_{t+1}(a)) \geq A_k^{i+1}(\text{rank}(a)).$$

Wenn a also $\text{rank}(a)$ viele Eisenstücke erhalten hat, dann gilt (zu einem Zeitpunkt s)

$$\text{rank}(\text{par}_s(a)) \geq A_k^{\text{rank}(a)}(\text{rank}(a)) = A_{k+1}(\text{rank}(a))$$
$$\delta_s(a) \geq k+1$$

Da $\delta_s(a)$ durch $\alpha(n)$ beschränkt ist (Lemma 29), kann a höchstens $\text{rank}(a)\alpha(n)$ Eisenstücke erhalten.

Da es höchstens $\frac{n}{2^{r-2}}$ Knoten mit Rang r gibt (Lemma 28), ist die maximale Anzahl an Eisenstücken am Ende beschränkt durch

$$\sum_{r=0}^{\infty} r\alpha(n)\frac{n}{2^{r-2}} = n\alpha(n)\sum_{r=0}^{\infty}\frac{r}{2^{r-2}} = 8n\alpha(n)$$

Laufzeit von Union-Find

Theorem 30

Eine Folge von m Union/Find-Aufrufen auf n Elementen kann in Zeit $\mathcal{O}((n+m)\alpha(n))$ ausgeführt werden.

Beweis:

Unions: Zeit $\mathcal{O}(m)$

Finds: $\mathcal{O}(m\alpha(n))$ Goldstücke, $\mathcal{O}(n\alpha(n))$ Eisenstücke, also $\mathcal{O}((m+n)\alpha(n))$ □

Idee der dynamischen Programmierung

Bestimme tabellarisch alle Teillösungen eines Problems, bis schließlich die Gesamtlösung erreicht ist.

Die Teillösungen werden dabei mit Hilfe der bereits existierenden Einträge berechnet.

Dynamischen Programmieren ist eng verwandt mit der Problemlösung durch Backtracking.

Die zusätzliche Idee ist, Rekursion durch Iteration zu ersetzen und durch tabellarisches Festhalten von Teillösungen Mehrfachberechnungen zu vermeiden.

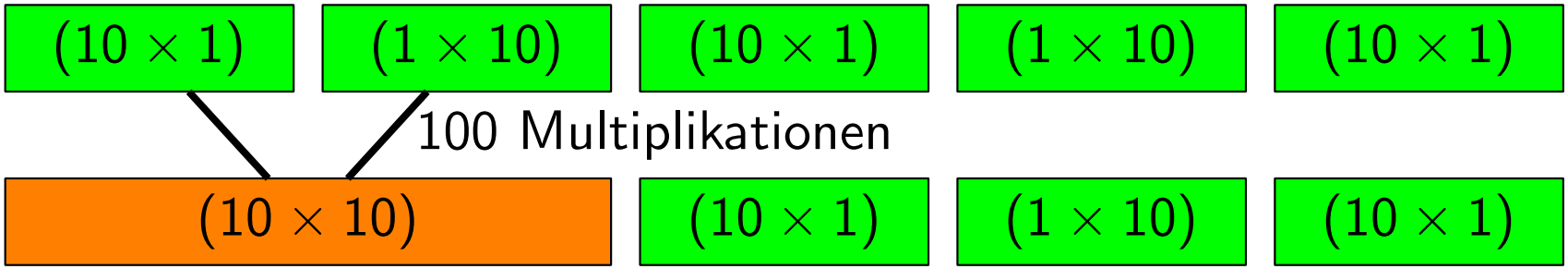
Beispiel: Multiplikation einer Matrizenfolge

Multiplikation von **links**

(10×1) (1×10) (10×1) (1×10) (10×1)

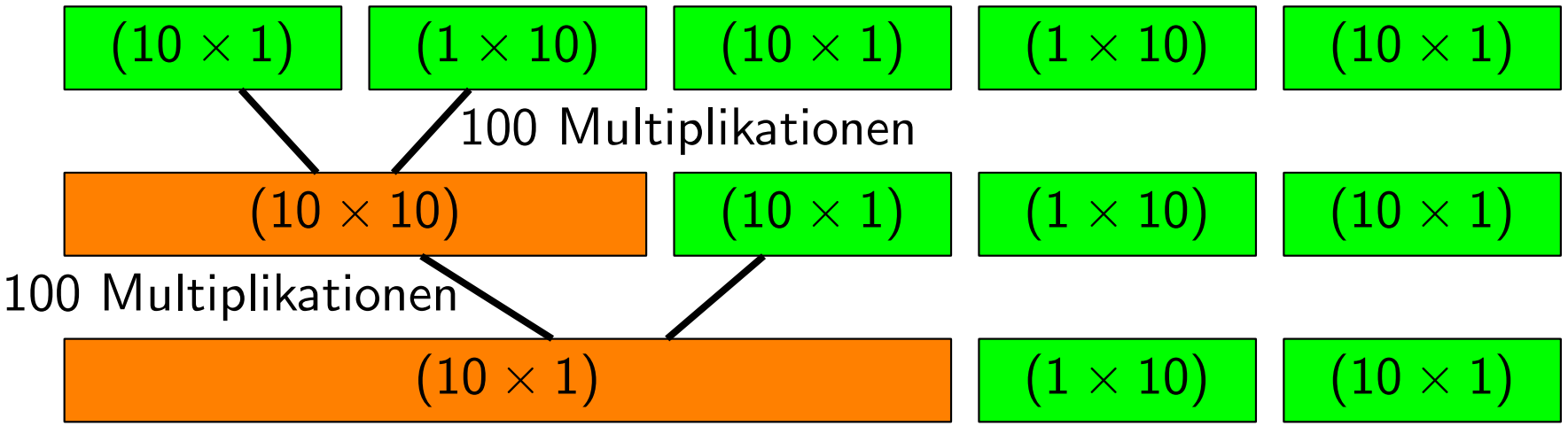
Beispiel: Multiplikation einer Matrizenfolge

Multiplikation von **links**

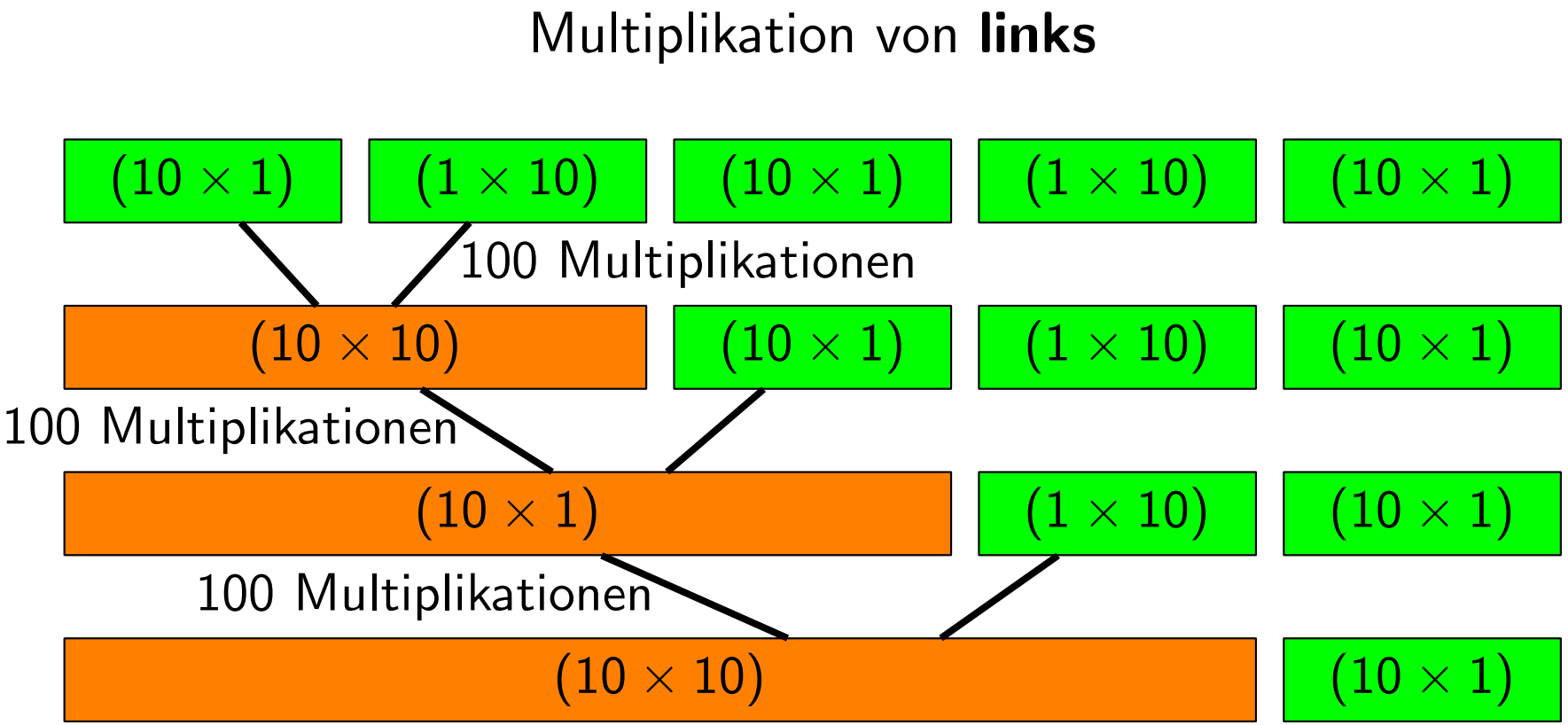


Beispiel: Multiplikation einer Matrizenfolge

Multiplikation von **links**

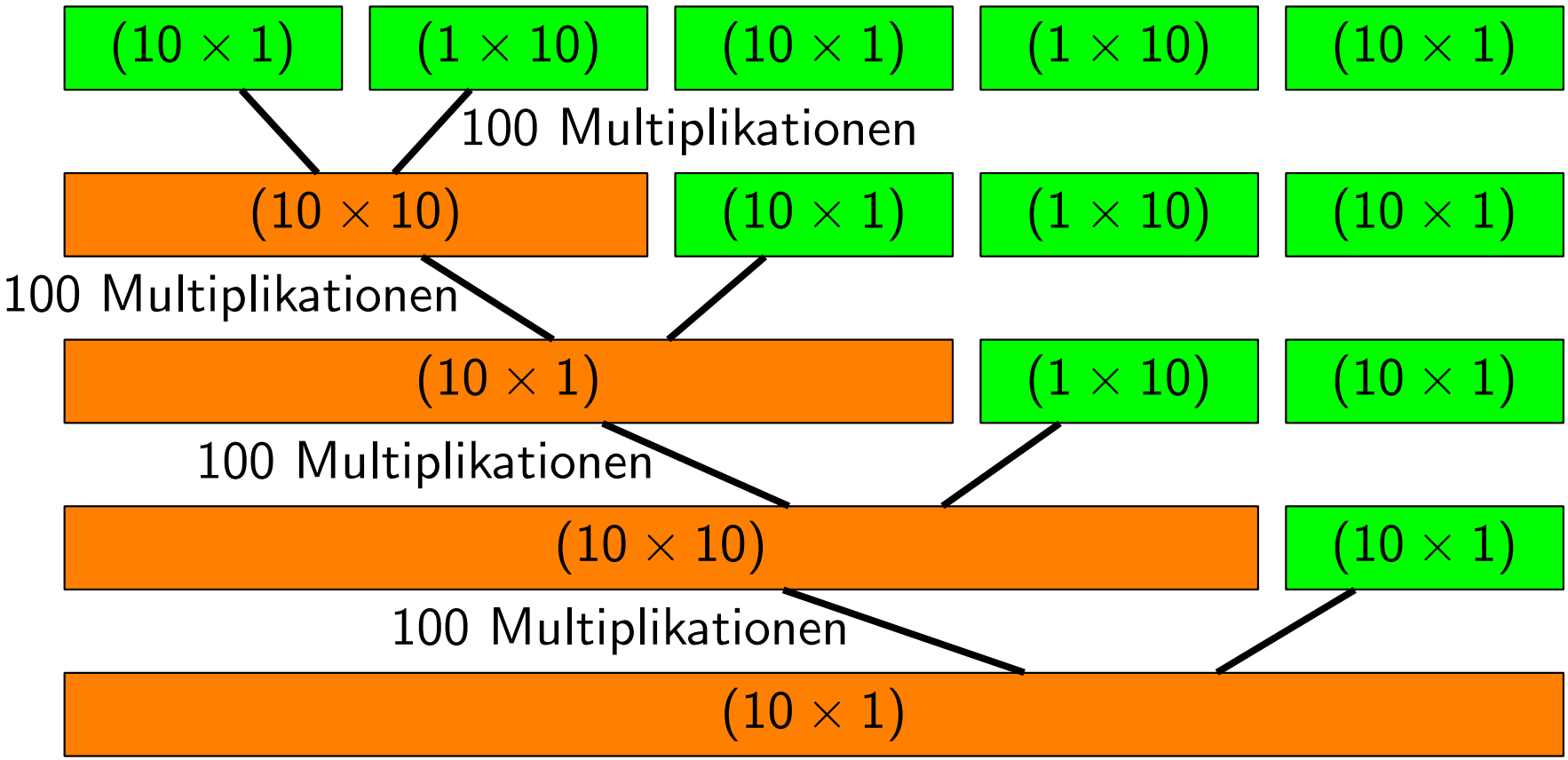


Beispiel: Multiplikation einer Matrizenfolge



Beispiel: Multiplikation einer Matrizenfolge

Multiplikation von **links**



Insgesamt: **400** Multiplikationen

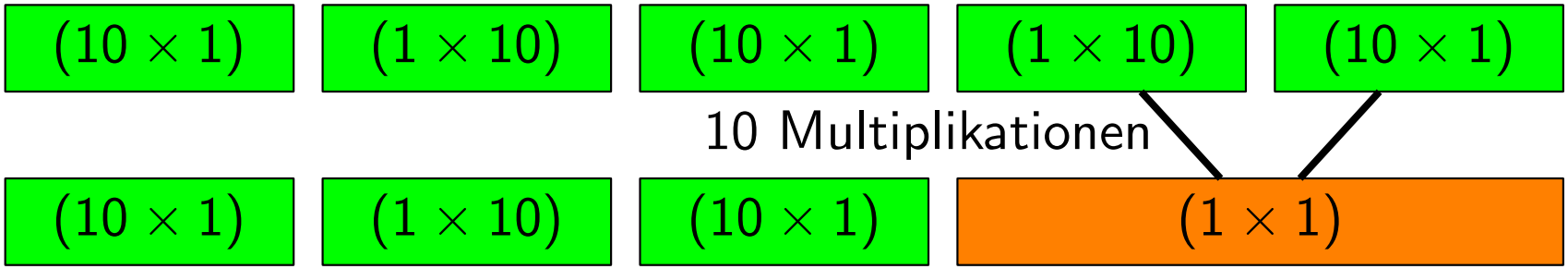
Beispiel: Multiplikation einer Matrizenfolge

Multiplikation von **rechts**

(10×1) (1×10) (10×1) (1×10) (10×1)

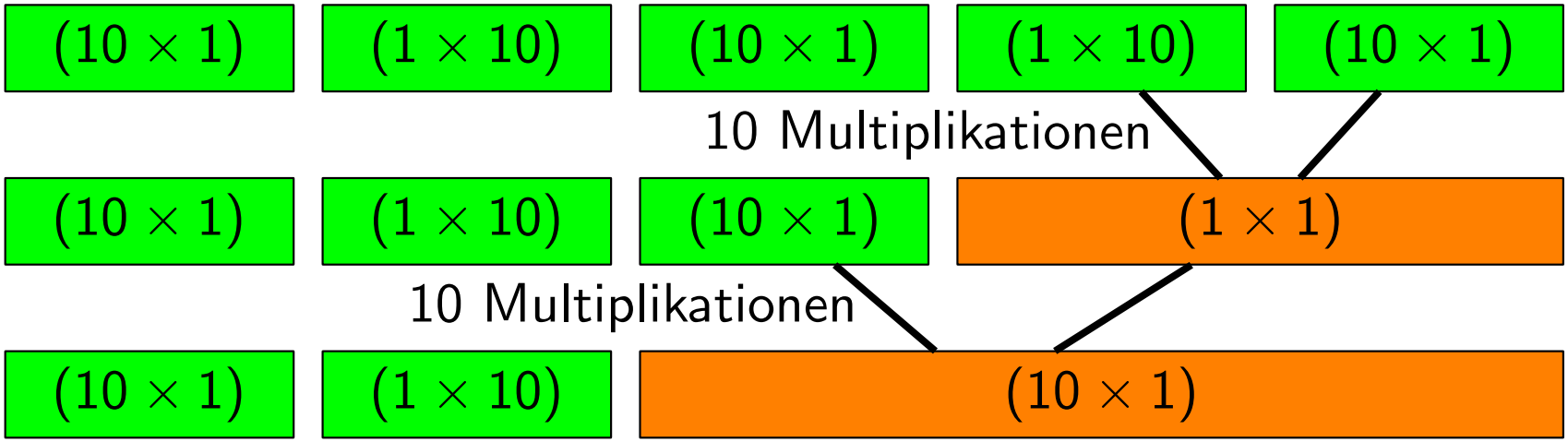
Beispiel: Multiplikation einer Matrizenfolge

Multiplikation von **rechts**



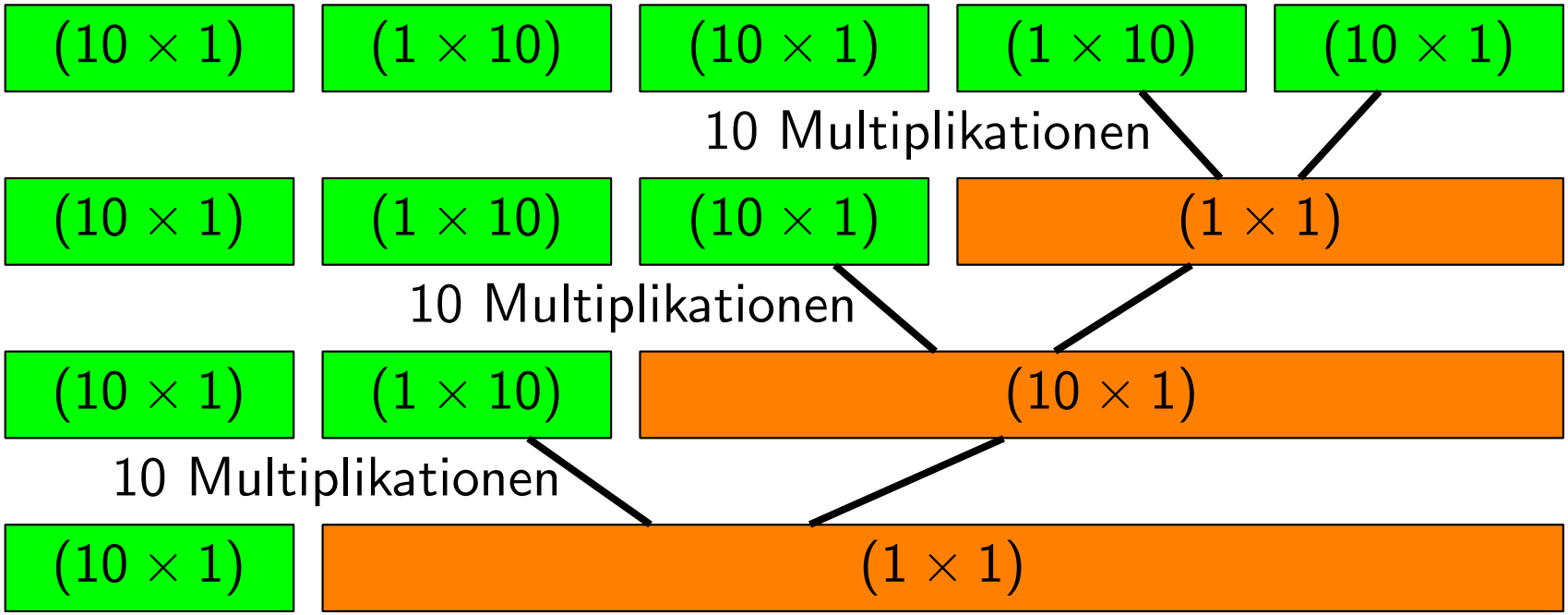
Beispiel: Multiplikation einer Matrizenfolge

Multiplikation von **rechts**



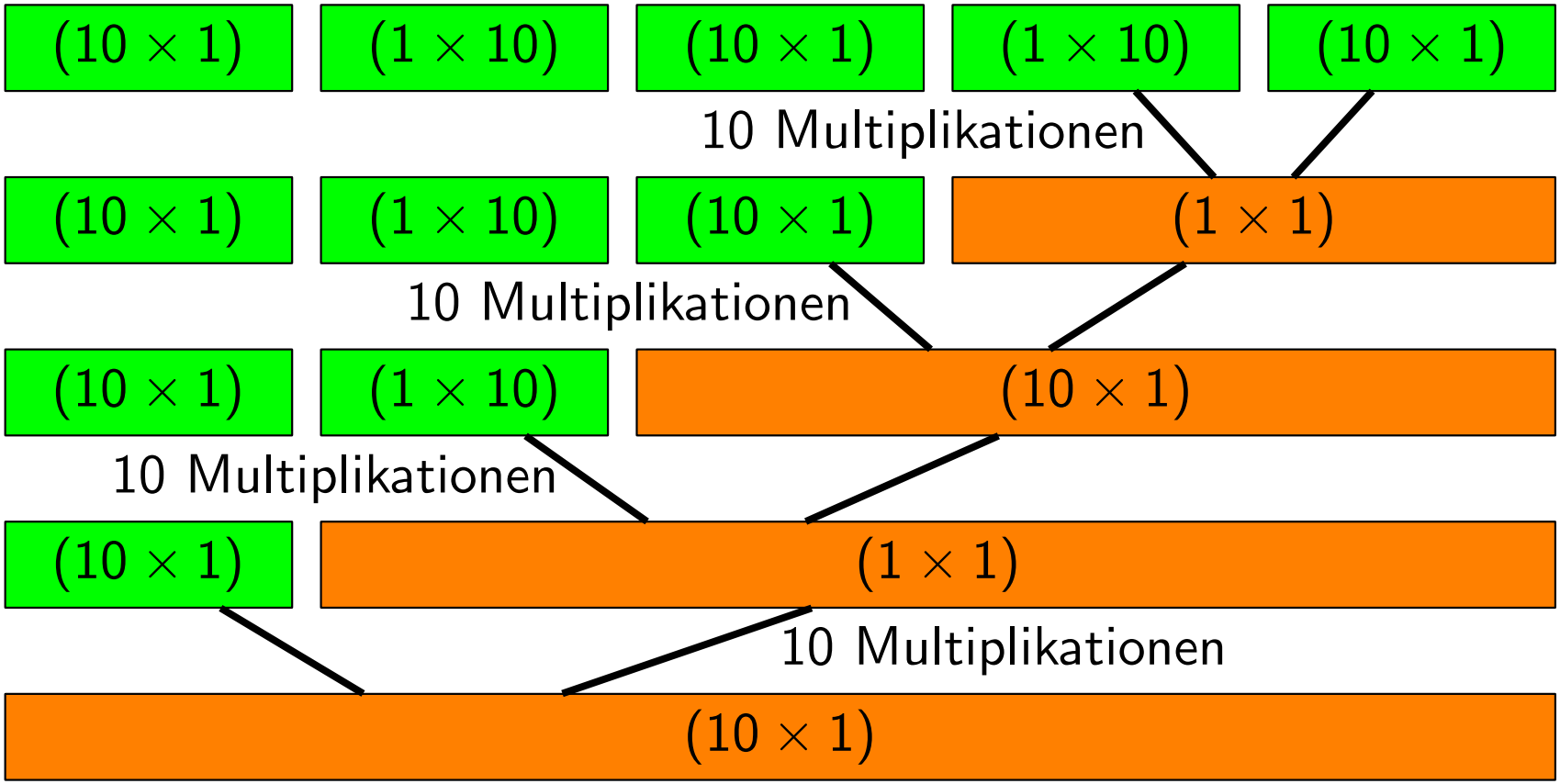
Beispiel: Multiplikation einer Matrizenfolge

Multiplikation von **rechts**



Beispiel: Multiplikation einer Matrizenfolge

Multiplikation von **rechts**



Insgesamt: **40** Multiplikationen

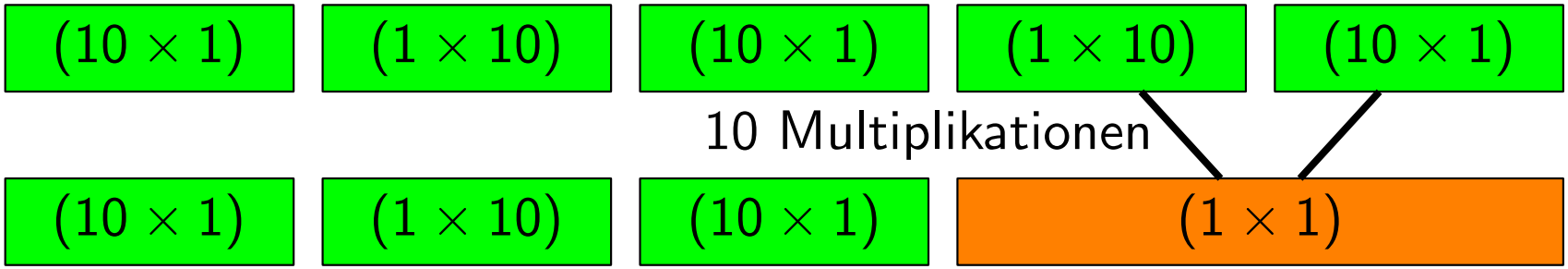
Beispiel: Multiplikation einer Matrizenfolge

Multiplikation in **optimaler Reihenfolge**

(10×1) (1×10) (10×1) (1×10) (10×1)

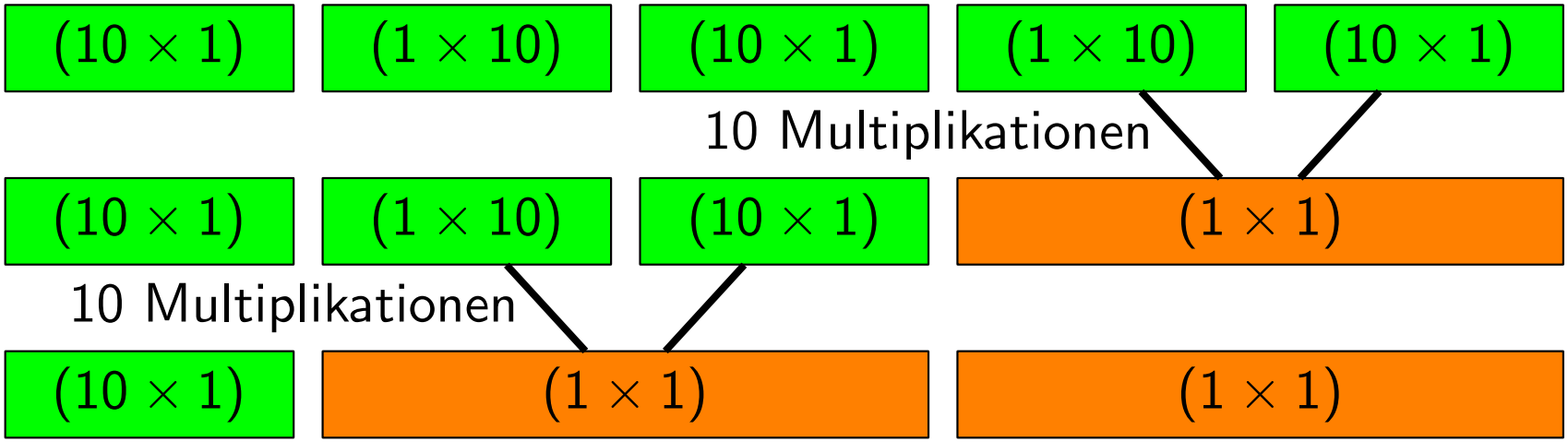
Beispiel: Multiplikation einer Matrizenfolge

Multiplikation in **optimaler Reihenfolge**



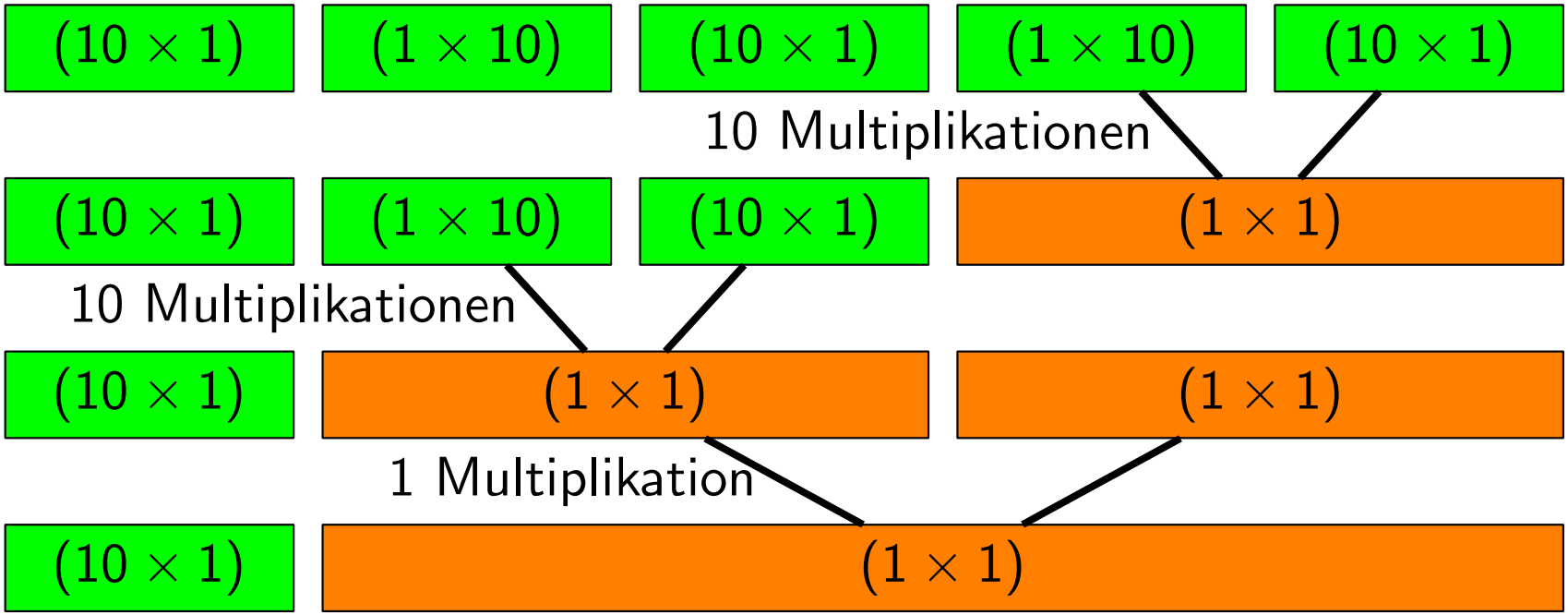
Beispiel: Multiplikation einer Matrizenfolge

Multiplikation in **optimaler Reihenfolge**



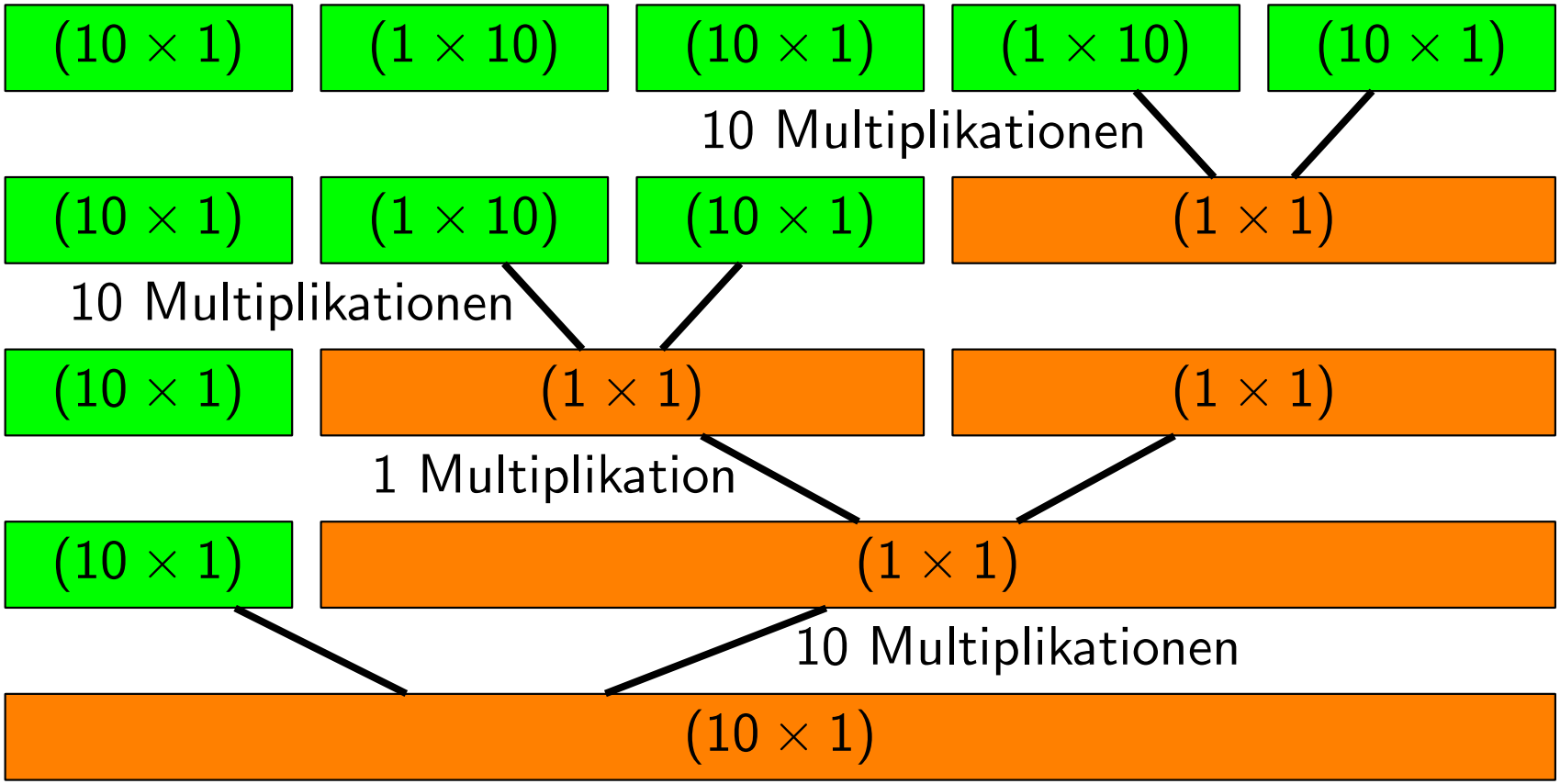
Beispiel: Multiplikation einer Matrizenfolge

Multiplikation in **optimaler Reihenfolge**



Beispiel: Multiplikation einer Matrizenfolge

Multiplikation in **optimaler Reihenfolge**



Insgesamt: **31** Multiplikationen

Multiplikation einer Matrizenfolge

$A_{(n,m)}$ sei eine Matrix A mit n Zeilen und m Spalten.

Annahme: Berechnung von $A_{(n,m)} := B_{(n,q)} \cdot C_{(q,m)}$ benötigt $n \cdot q \cdot m$ skalare Multiplikationen.

Eingabe: Matrizenfolge $M^1_{(n_0,n_1)}, M^2_{(n_1,n_2)}, M^3_{(n_2,n_3)}, \ldots, M^N_{(n_{N-1},n_N)}$.

$\text{cost}(M^1, \ldots, M^N) :=$ minimale Zahl der skalaren Multiplikationen, um $M^1 \cdots M^N$ zu berechnen.

Dynamisches Programmierung liefert:

$$\text{cost}(M^i, \ldots, M^j) = \min_k \{\text{cost}(M^i, \ldots, M^k) + \text{cost}(M^{k+1}, \ldots, M^j) + n_{i-1} \cdot n_k \cdot n_j\}$$

Multiplikation einer Matrizenfolge

```
for i := 1 to N do
    cost[i, i] := 0;
    for j := i + 1 to N do
        cost[i, j] := ∞;
    endfor
endfor
for d := 1 to N − 1 do
    for i := 1 to N − d do
        j := i + d;
        for k := i to j − 1 do
            t := cost[i, k] + cost[k + 1, j] + n[i − 1] · n[k] · n[j];
            if t < cost[i, j] then
                cost[i, j] := t;
                best[i, j] := k;
            endif
        endfor
    endfor
endfor
return best
```

Optimale Suchbäume

Erzeugung von optimalen Suchbäumen:
Die direkte Methode $\Theta(n^3)$.

Der Algorithmus von Donald E. Knuth hat einen Aufwand von $\Theta(n^2)$.
(Teleskop-Summe)

Interessant ist hier, wie man durch eine genaue Analyse des Problems den kubischen Algorithmus in einen quadratischen verwandeln kann.

Der Algorithmus ist nicht komplizierter, sondern der Gewinn liegt im Auslassen überflüssiger Schritte.

Optimale Suchbäume

Sei ein linear geordnetes Feld gegeben mit $v_1 < v_2 < \cdots < v_n$.

Sei $V = \{v_1, \ldots, v_n\}$.

Für jeden Knoten $v \in V$ haben wir eine Zugriffshäufigkeit (oder Gewicht) $\gamma(v)$ gegeben.

Der Wert $\gamma(v)$ kann sowohl die relativen als auch die absoluten Häufigkeiten bezeichnen.

Ein binärer Suchbaum für $v_1 < v_2 < \cdots < v_n$ ist ein Binärbaum mit Knotenmenge $\{v_1, v_2, \ldots, v_n\}$ mit:
Für jeden Knoten v mit linkem (bzw. rechten) Unterbaum L (bzw. R) und $u \in L$ (bzw. $w \in R$) gilt: $u < v$ ($v < w$).

Optimale Suchbäume

Jeder Knoten v hat ein Level $\ell(v)$:
$\ell(v) := 1+$ Abstand des Knotens v zur Wurzel.

Das Auffinden eines Knotens auf Level ℓ erfordert ℓ Vergleiche.

Problem: Finde einen binären Suchbaum B, der die gewichtete innere Pfadlänge

$$P(B) := \sum_{v \in V} \ell(v) \cdot \gamma(v)$$

minimiert.

Die innere Pfadlänge bestimmt die durchschnittlichen Kosten einer Sequenz von Find-Operationen.

Dynamische Programmierung möglich, da die Unterbäume eines optimalen Baums auch optimal sind.

Optimale Suchbäume

Bezeichnungen:

- Knotenmenge $= \{1, \ldots, n\}$, d.h. die Zahl i entspricht dem Knoten v_i.
- $\ell_i := \ell(i)$ und $\gamma_i := \gamma(i)$
- $P[i, j]$: gewichtete innere Pfadlänge eines optimalen Suchbaumes der Knoten $\{i, \ldots, j\}$.
- $R[i, j]$: Wurzel eines optimalen Suchbaumes für $\{i, \ldots, j\}$.
- Später: $r[i, j]$ kleinstmögliche Wurzel, $R[i, j]$ größtmögliche Wurzel.
- $\Gamma[i, j] := \sum_{k=i}^{j} \gamma_k$: Gewicht der Knotenmenge $\{i, \ldots, j\}$.

Optimale Suchbäume in Zeit $\mathcal{O}(n^3)$

Im dynamischen Ansatz sind nun Werte $R[i,j]$ gesucht, die einen optimalen Suchbaum B mit Kosten $P[i,j]$ realisieren.

Für einen binären Suchbaum B, wobei B_L (B_R) der linke (rechte) Unterbaum der Wurzel ist, gilt:

$$P(B) := P(B_L) + P(B_R) + \Gamma(B)$$

Wir realisieren diesen Ansatz zunächst in einem kubischen Algorithmus. Hier nur die Idee:

Algorithmus: Berechnung eines optimalen Suchbaums

- $P[i,j] = \Gamma[i,j] + \min\{P[i,k-1] + P[k+1,j] \mid k \in \{i,\ldots,j\}\}$
- $R[i,j] = k$, für das $P[i,k-1] + P[k+1,j]$ das Minimum annimmt.

Monotonie der Wurzel

Knuths Verbesserung ($n^3 \rightsquigarrow n^2$) beruht auf folgendem Satz:

Theorem 31 (Monotonie der Wurzel)

Sei $r[i,j]$ (bzw. $R[i,j]$) die kleinste (bzw. größte) Wurzel eines optimalen Suchbaumes für die Knoten $\{i,\ldots,j\}$. Dann gilt für $n \geq 2$:

$$\begin{aligned} r[1,n-1] &\leq r[1,n] \\ R[1,n-1] &\leq R[1,n] \end{aligned}$$

Wegen Links-Rechts-Symmetrie gilt analog:

Theorem 32 (Monotonie der Wurzel)

Für $n \geq 2$ gilt:

$$\begin{aligned} r[1,n] &\leq r[2,n] \\ R[1,n] &\leq R[2,n] \end{aligned}$$

Monotonie der Wurzel

Beweis: Wir beweisen den Satz durch Induktion über die Zahl der Knoten, d. h. wir können ihn für kleineres n bereits als bewiesen annehmen.

Zunächst zeigen wir folgendes Lemma:

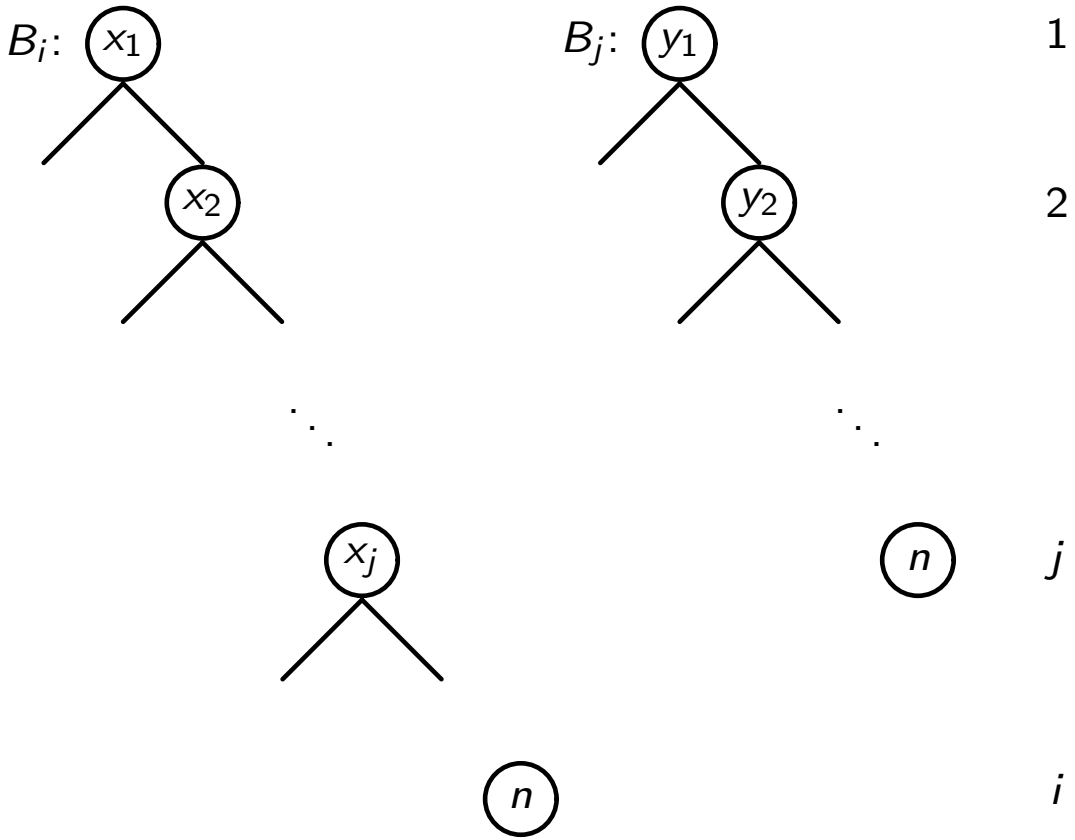
Lemma 33 (große Wurzel & minimaler Level für n)

Sei B_j ein optimaler Suchbaum für $\{1, \ldots, n\}$ mit minimalem Level j von Knoten n. Sei y_1 die Wurzel von B_j. Sei B_i ein optimaler Suchbaum für $\{1, \ldots, n\}$ mit Wurzel $x_1 > y_1$. Dann existiert ein optimaler Suchbaum B' für $\{1, \ldots, n\}$ mit Wurzel x_1 und Knoten n auf Level j.

Wir werden sehen, wie diese Verbindung der Eigenschaften minimales Level für n und große Wurzel für den Beweis des Satzes von Nutzen ist.

Monotonie der Wurzel

Beweis des Lemmas: Wir betrachten die rechten Äste der Bäume B_i und B_j, wobei die Knoten von B_i mit x_k und die Knoten von B_j mit y_k bezeichnet sind (siehe folgende Abbildung).



Monotonie der Wurzel

Bei festem x_1 maximieren wir x_2, dann maximieren wir x_3 usw, d.h. $x_{k+1} = R[x_k + 1, n]$.

Der neue Baum wird weiterhin mit B_i bezeichnet.

Falls in B_i und B_j der Knoten n auf gleichem Level j liegt, sind wir fertig.

Andernfalls können wir annehmen, dass der Knoten n in B_i auf Level i liegt und $i > j$ gilt, weil j minimal gewählt wurde.

Sei k maximal mit $x_k > y_k$.

Dann gilt $1 \leq k < j$.

Wegen $y_k + 1 \leq x_k + 1$ und Induktion (Theorem 32) folgt

$$y_{k+1} \leq R[y_k + 1, n] \leq R[x_k + 1, n] = x_{k+1}.$$

Also folgt $y_{k+1} = x_{k+1}$, da k maximal gewählt wurde.

Monotonie der Wurzel

Ausserdem muss $k+1 < j$ gelten, denn sonst würde $n = y_j = y_{k+1} = x_{k+1}$ gelten, d.h. n würde in B_i und B_j auf dem gleichen Level liegen.

Also existiert das rechte Kind von x_{k+1} bzw. y_{k+1}.

Sei nun R_i (R_j) der Unterbaum von B_i (B_j) unterhalb des rechten Kinds von x_{k+1} (y_{k+1}).

Wegen $x_{k+1} = y_{k+1}$ haben R_i und R_j dieselbe Knotenmenge $\{x_{k+1}+1, \ldots, n\}$ und sind optimale Suchbäume.

Wir bilden einen Baum B' durch Ersetzen von R_i in B_i durch R_j.

Da $P(R_i) = P(R_j)$ gilt, ergibt sich auch $P(B') = P(B_i) = P(B_j)$.

Also ist B' ist optimal für $\{1, \ldots, n\}$, hat i_1 als Wurzel und den Knoten n auf Level j. □

Monotonie der Wurzel

Symmetrisch zu Lemma 33 lässt sich zeigen:

Lemma 34 (kleine Wurzel & maximaler Level für n)

Sei B_i ein optimaler Suchbaum für $\{1, \ldots, n\}$ mit maximalem Level i von Knoten n. Sei x_1 die Wurzel von B_i. Sei B_j ein optimaler Suchbaum für $\{1, \ldots, n\}$ mit Wurzel $y_1 < x_1$. Dann existiert ein optimaler Suchbaum B' für $\{1, \ldots, n\}$ mit Wurzel y_1 und Knoten n auf Level i.

Nun zurück zum Beweis von Theorem 31:

Im folgenden bezeichnen wir mit α das Gewicht des größten Knotens n, d.h. $\alpha := \gamma_n$.

Der Wert α variiert zwischen 0 und ∞.

Sei r_α (R_α) die kleinste (größte) Wurzel eines optimalen Suchbaums für die Knoten $\{1, \ldots, n\}$ unter der Bedingung $\alpha = \gamma_n$.

Monotonie der Wurzel

Behauptung 1: $r[1, n-1] \leq r_0$ bzw. $R[1, n-1] \leq R_0$.

Beachte: Ist B (B') optimaler Suchbaum für $\{1, \ldots, n\}$ ($\{1, \ldots, n-1\}$), so gilt $P(B) = P(B')$ ($\alpha = 0$ ist hier wichtig).

Für $R[1, n-1] \leq R_0$:

- Sei B optimaler Suchbaum für $\{1, \ldots, n-1\}$ mit Wurzel $R[1, n-1]$.
- Füge n ganz rechts zu B hinzu.
- Der resultierende Suchbaum für $\{1, \ldots, n\}$ ist optimal und hat Wurzel $R[1, n-1]$.

Für $r[1, n-1] \leq r_0$:

- Sei B' optimaler Suchbaum für $\{1, \ldots, n\}$ mit Wurzel r_0.
- Entferne n aus B'.
- Der resultierende Suchbaum für $\{1, \ldots, n-1\}$ ist optimal und hat Wurzel r_0.

Optimale Suchbäume

Zusammen mit Behauptung 1 folgt Theorem 31 aus:

Behauptung 2: Wenn $\alpha < \beta$, dann $r_\alpha \leq r_\beta$ und $R_\alpha \leq R_\beta$.

Wir zeigen zunächst die Behauptung $R_\alpha \leq R_\beta$.

Für $i \in \{1, \ldots, n\}$ sei B_i ein optimaler Suchbaum unter der Nebenbedingung, dass der Knoten n auf dem Level i liegt.

Beachte:

- B_i muss nicht insgesamt optimal sein: Es kann sein, dass durch Verschieben von n auf einen anderen Level $\neq i$ ein besserer Suchbaum entsteht.
- B_i ist nicht unbedingt eindeutig.

Wir wählen nun B_i so, dass die Wurzel maximal wird (und weiterhin n auf Level i liegt).

Optimale Suchbäume

Mit $P_\alpha(B_i)$ bezeichnen wir die gewichtete innere Pfadlänge in Abhängigkeit vom Gewicht α des Knotens n.

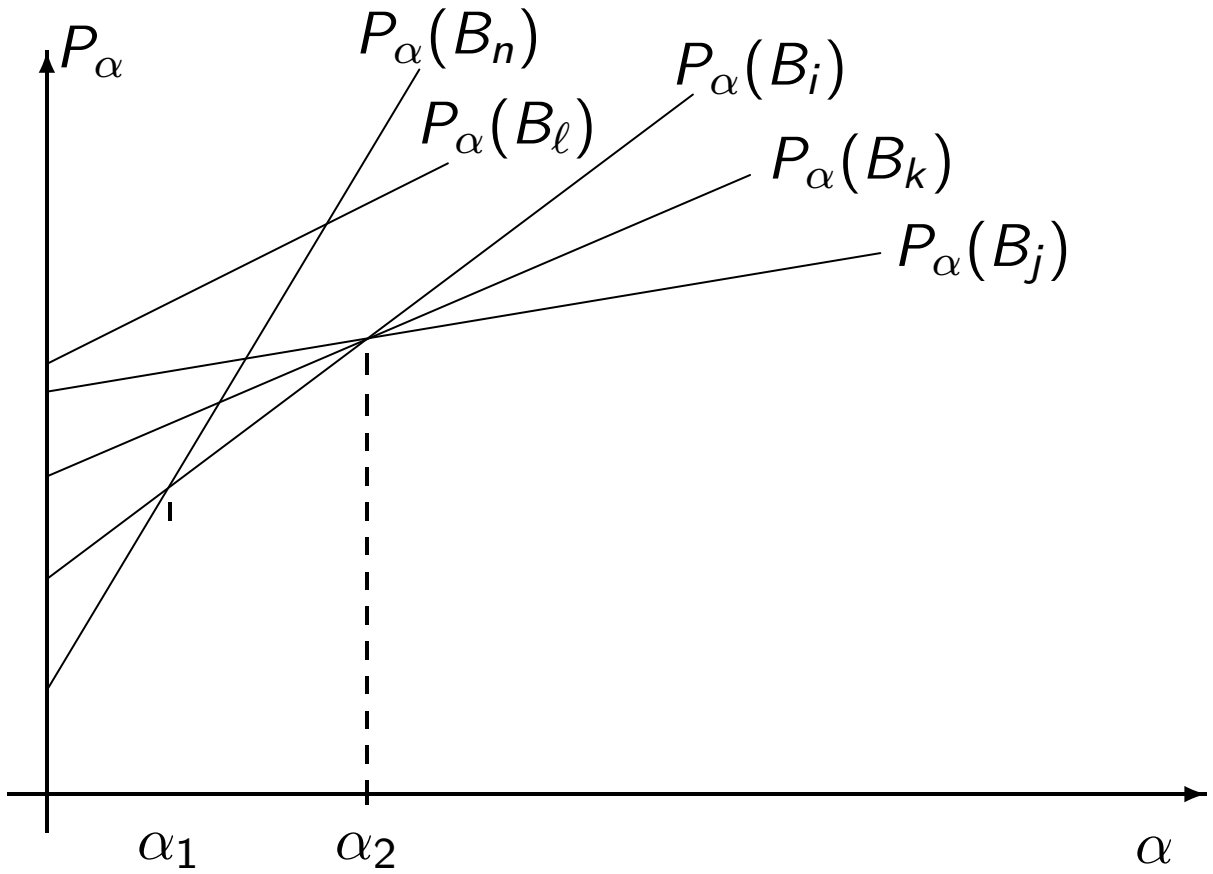
Dann gibt es eine Konstante $c(i)$ mit

$$P_\alpha(B_i) = \alpha \cdot i + c(i),$$

d.h. der Graph $P_\alpha(B_i)$ ist eine Gerade mit Steigung i.

Aufgrund der Linearität erhalten wir das folgende (vertikal gestauchte) Bild, bei der jede Steigung $1 \leq i \leq n$ genau einmal vorkommt.

Optimale Suchbäume



Optimale Suchbäume

Für jedes α ist dann

$$P(\alpha) := \min\{P_\alpha(B_i) \mid 1 \leq i \leq n\}$$

das Gewicht eines optimalen Suchbaums unter der Bedingung $\gamma_n = \alpha$.

Dann ist $P(\alpha)$ ein konkaver Linienzug mit den Ecken $\alpha_1 < \alpha_2 < \cdots < \alpha_m$. Setze $\alpha_0 = 0$ und $\alpha_{m+1} = \infty$.

Sei i_k so, dass für alle $\alpha \in [\alpha_k, \alpha_{k+1}]$ gilt: $P(\alpha) = P_\alpha(B_{i_k})$, d.h. B_{i_k} ist optimal, falls $\gamma_n \in [\alpha_k, \alpha_{k+1}]$.

Beachte: Dann gilt für alle $\alpha \in (\alpha_k, \alpha_{k+1})$: $R_\alpha =$ Wurzel von B_{i_k}.

Es genügt, die beiden folgenden Aussagen zu zeigen:

1. Wenn $\alpha_k < \alpha < \alpha_{k+1} = \beta$ dann gilt $R_\alpha \leq R_\beta$.
2. Wenn $\alpha = \alpha_k < \beta < \alpha_{k+1}$ dann gilt $R_\alpha = R_\beta$.

Optimale Suchbäume

Zu 1: Für α gilt: $R_\alpha =$ Wurzel von B_{i_k}.

Ausserdem ist B_{i_k} für $\gamma_n = \beta$ noch immer ein optimaler Suchbaum.

Also gilt $R_\beta =$ größte Wurzel eines für $\gamma_n = \beta$ optimalen Suchbaum $\geq R_\alpha$.

Zu 2: Für β gilt: $R_\beta =$ Wurzel von B_{i_k}.

Ausserdem ist B_{i_k} für $\gamma_n = \alpha$ noch immer ein optimaler Suchbaum.

Wegen Lemma 33 gibt es einen Suchbaum B mit $P_\alpha(B) = P(\alpha)$ (das Optimum bei α) und Wurzel R_α, wo ausserdem n auf dem kleinstmöglichen Level liegt.

Dieser kleinstmögliche Level ist aber i_k.

Also ist B ein optimaler Suchbaum unter der Einschränkung, dass n auf Level i_k liegt, und bei dem ausserdem die Wurzel ($= R_\alpha$) maximal ist.

Also haben B und B_{i_k} die gleiche Wurzel, d.h. $R_\alpha = R_\beta$.

Optimale Suchbäume

Analog zeigt man: Wenn $\alpha < \beta$, dann $r_\alpha \leq r_\beta$.

Man wählt hier die Wurzel von B_i minimal.

Dann gilt für alle $\alpha \in (\alpha_k, \alpha_{k+1})$: $r_\alpha =$ Wurzel von B_{i_k}.

Man zeigt nun die beiden folgenden Aussagen:

1. Wenn $\alpha_k < \alpha < \alpha_{k+1} = \beta$ dann gilt $r_\alpha = r_\beta$.
2. Wenn $\alpha = \alpha_k < \beta < \alpha_{k+1}$ dann gilt $r_\alpha \leq r_\beta$.

Für den Beweis von 1 verwedet man Lemma 34. □

Korollar

Es gilt: $r[i, j-1] \leq r[i, j] \leq r[i+1, j]$.

Optimale Suchbäume

```
cost[n, n+1] := 0;
for i := 1 to n do
    cost[i, i-1] := 0;
    cost[i, i] := γ(i);
    Γ[i, i] := γ(i);
    r[i, i] := i;
endfor
```

```
for d := 1 to n-1 do
    for i := 1 to n-d do
        j := i + d;
        left := r[i, j-1]; right := r[i+1, j];
        root := left;
        t := cost[i, left-1] + cost[left+1, j];
        for k := left+1 to right do
            if cost[i, k-1] + cost[k+1, j] < t then
                t := cost[i, k-1] + cost[k+1, j];
                root := k;
            endif
        endfor
        Γ[i, j] := Γ[i, j-1] + γ(j);
        cost[i, j] := t + Γ[i, j];
        r[i, j] := root;
    endfor
endfor
```

Optimale Suchbäume

$$\text{Laufzeit: } \sum_{d=1}^{n} \sum_{i=1}^{n-d} (1 + r[i+1, i+d] - r[i, i+d-1]) =$$
$$\sum_{d=1}^{n} (n - d + r[n-d+1, n] - r[1, d]) \in \Theta(n^2).$$

Bemerkung: Es wurde ein linear geordnetes Feld $v_1, \ldots, v_n$ von Knoten vorausgesetzt. Ein solches Feld erhält man aus einem ungeordneten Feld in Zeit $\mathcal{O}(n \log n)$ Schritten.

Damit gilt: Aus einem beliebigen Feld mit n Elementen kann ein optimaler Suchbaum in Zeit $\Theta(n^2 + n \log n) = \Theta(n^2)$ Schritten erzeugt werden.

Mittlere Höhe von Suchbäumen

Wir werden zeigen, dass ein Suchbaum auf den Knoten $\{1, \ldots, n\}$ im Mittel die Höhe $\mathcal{O}(\log n)$ hat.

Sei $\mathcal{B}_n$ die Menge aller binären Suchbäume auf den Knoten $\{1, \ldots, n\}$ ($\mathcal{B}_0$ besteht nur aus dem leeren Baum $\emptyset$ und wir setzen $\text{height}(\emptyset) = -\infty$).

Wir erzeugen ein $B \in \mathcal{B}_n$ mittels folgenden Zufallsexperiment:

- Wir wählen zufällig und gleichverteilt eine Permutation $(\pi_1, \pi_2, \ldots, \pi_n)$ von $(1, 2, \ldots, n)$ aus. Jede der $n!$ vielen Permutationen wird mit Wahrscheinlichkeit $1/n!$ ausgewählt.
- Dann wird ein Suchbaum aufgebaut, indem die Elemente $1, \ldots, n$ in der Reihnfolge $\pi_1, \pi_2, \ldots, \pi_n$ in den Suchbaum eingefügt werden.
- Beachte: Verschiedene Permutationen können den gleichen Suchbaum erzeugen.

  Beispiel: $(2, 1, 3)$ und $(2, 3, 1)$ erzeugen den gleichen Suchbaum (mit Wurzel 2).

Mittlere Höhe von Suchbäumen

Folgendes Zufallsexperiment liefert für jeden Suchbaum die gleiche Wahrscheinlichkeit wie oben:

- Falls $n \geq 2$, wähle zufällig und gleichverteilt ein Element $i \in \{1, \ldots, n\}$ aus.
  Jedes Element wird mit Wahrscheinlichkeit $1/n$ gezogen.
- Erzeuge dann rekursiv nach dem gleichen Experiment einen Suchbaum $L \in \mathcal{B}_{i-1}$ (bzw. $R \in \mathcal{B}_{n-i}$)
- Ersetze in R jeden Knoten j durch $i + j$.
- Der erzeugte Suchbaum hat i als Wurzel und L (bzw. R) als linken (bzw. rechten) Teilbaum.

Beachte: Dieses Zufallsexperiment ergibt nicht die Gleichverteilung auf Binärbäumen mit n Knoten

Mittlere Höhe von Suchbäumen

Wir definieren folgende Zufallsvariablen:

- H_n ist die Höhe eines zufällig erzeugten Suchbaums $B \in \mathcal{B}_n$.
- $X_n = 2^{H_n}$

Theorem 35

Für den Erwartungswert

$$E[H_n] = \sum_{B \in \mathcal{B}_n} Prob(B) \cdot height(B)$$

gilt: $E[H_n] \leq 3 \cdot \log_2(n)$.

Beweis: Wir zeigen zunächst, dass $E[X_n]$ durch ein Polynom $p(n)$ beschränkt ist.

Sei B ein Suchbaum mit Wurzel $1 \leq i \leq n$ und linken (bzw. rechten) Teilbaum L (R).

Dann gilt $\text{height}(B) = 1 + \max\{\text{height}(L), \text{height}(R)\}$ und daher:

Mittlere Höhe von Suchbäumen

$$
\begin{aligned}
E[X_n] &= \sum_{B \in \mathcal{B}_n} \mathrm{Prob}(B) \cdot 2^{\mathrm{height}(B)} \\
&= \sum_{i=1}^{n} \sum_{L \in \mathcal{B}_{i-1}} \sum_{R \in \mathcal{B}_{n-i}} \frac{1}{n} \mathrm{Prob}(L)\mathrm{Prob}(R) 2^{1+\max\{\mathrm{height}(L),\mathrm{height}(R)\}} \\
&= \frac{2}{n} \sum_{i=1}^{n} \sum_{L \in \mathcal{B}_{i-1}} \sum_{R \in \mathcal{B}_{n-i}} \mathrm{Prob}(L)\mathrm{Prob}(R) \max\{2^{\mathrm{height}(L)}, 2^{\mathrm{height}(R)}\} \\
&\leq \frac{2}{n} \sum_{i=1}^{n} \sum_{L \in \mathcal{B}_{i-1}} \sum_{R \in \mathcal{B}_{n-i}} \mathrm{Prob}(L)\mathrm{Prob}(R)(2^{\mathrm{height}(L)} + 2^{\mathrm{height}(R)}) \\
&= \frac{2}{n} \sum_{i=1}^{n} \left(\sum_{L \in \mathcal{B}_{i-1}} \mathrm{Prob}(L) 2^{\mathrm{height}(L)} + \sum_{R \in \mathcal{B}_{n-i}} \mathrm{Prob}(R) 2^{\mathrm{height}(R)} \right) \\
&= \frac{2}{n} \sum_{i=1}^{n} (E[X_{i-1}] + E[X_{n-i}]) = \frac{4}{n} \sum_{i=0}^{n-1} E[X_i]
\end{aligned}
$$

Mittlere Höhe von Suchbäumen

Behauptung 1: $\sum_{i=0}^{n-1} \binom{i+3}{3} = \binom{n+3}{4}$ für $n \geq 1$.

Beweis durch Induktion:

$n = 1$: $\sum_{i=0}^{0} \binom{i+3}{3} = \binom{3}{3} = 1 = \binom{1+3}{4}$.

Sei nun $n \geq 2$:

$$\sum_{i=0}^{n-1} \binom{i+3}{3} = \sum_{i=0}^{n-2} \binom{i+3}{3} + \binom{n+2}{3} = \binom{n+2}{4} + \binom{n+2}{3} = \binom{n+3}{4}.$$

Behauptung 2: $E[X_n] \leq \frac{1}{4}\binom{n+3}{3}$.

Beweis durch Induktion:

$n = 0$: $E[X_0] = 2^{-\infty} = 0 \leq \frac{1}{4}\binom{3}{3}$

$n = 1$: $E[X_1] = 2^0 = 1 = \frac{1}{4}\binom{4}{3}$

Sei nun $n \geq 2$:

Mittlere Höhe von Suchbäumen

$$
\begin{aligned}
E[X_n] &\leq \frac{4}{n}\sum_{i=0}^{n-1} E[X_i] \\
&\leq \frac{4}{n}\sum_{i=0}^{n-1} \frac{1}{4}\binom{i+3}{3} \\
&= \frac{1}{n}\sum_{i=0}^{n-1} \binom{i+3}{3} \\
&= \frac{1}{n}\binom{n+3}{4} \\
&= \frac{1}{n}\frac{(n+3)!}{4!(n-1)!} \\
&= \frac{1}{4}\frac{(n+3)!}{3!n!} \\
&= \frac{1}{4}\binom{n+3}{3}
\end{aligned}
$$

Mittlere Höhe von Suchbäumen

Ausserdem gilt $\frac{1}{4}\binom{n+3}{3} \leq n^3$ für $n \geq 1$ und damit $E[X_n] \leq n^3$.

Die Funktion $x \mapsto 2^x$ ist konvex.

Also folgt aus Jensen's Ungleichung (Folie 4):

$$2^{E[H_n]} \leq E[2^{H_n}] = E[X_n] \leq n^3.$$

Es folgt: $E[H_n] \leq 3\log_2(n)$. □

Bemerkung: Bei einer Gleichverteilung (jeder binäre Suchbaum tritt mit der gleichen Wahrscheinlichkeit auf) erhält man $\Theta(\sqrt{n})$ für die erwartete Höhe.

Bestimmung reguläre Ausdrücke

Erinnerung an GTI: Bestimmung reguläre Ausdrücke nach Kleene.

Ein nichtdeterministischer endlicher Automat (NEA) ist ein Tupel

$$A = (Q, \Sigma, \delta \subseteq Q \times \Sigma \times Q, I, F) \quad \text{(o.B.d.A. } Q = \{1, \ldots, n\}\text{)}.$$

Es sei $L^k[i,j]$ die Menge aller Wörter, die in A einen Pfad beschriften, der

- von i nach j führt und
- dabei nur Zwischenzustände aus der Menge $\{1, \ldots, k\}$ besucht (i und j müssen nicht in $\{1, \ldots, k\}$ liegen)

Gesucht: Regulärer Ausdruck für $L^n[i,j]$ für alle $i \in I$ und $j \in F$.

Es gilt:

$$\begin{aligned} L^0[i,j] &= \begin{cases} \{a \in \Sigma \mid (i,a,j) \in \delta\} & \text{falls } i \neq j \\ \{a \in \Sigma \mid (i,a,j) \in \delta\} \cup \{\varepsilon\} & \text{falls } i = j \end{cases} \\ L^k[i,j] &= L^{k-1}[i,j] + L^{k-1}[i,k] \cdot L^{k-1}[k,k]^* \cdot L^{k-1}[k,j] \end{aligned}$$

Bestimmung reguläre Ausdrücke

Algorithmus Reguläre Ausdrücke aus einem endlichen Automaten

procedure NEA2REGEXP
Eingabe : NEA $A = (Q, \Sigma, \delta \subseteq Q \times \Sigma \times Q, I, F)$
(Initialisiere: $L[i,j] := \{a \mid (i,a,j) \in \delta \vee a = \varepsilon \wedge i = j\}$)
begin
  for $k := 1$ **to** n **do**
    for $i := 1$ **to** n **do**
      for $j := 1$ **to** n **do**
        $L[i,j] := L[i,j] + L[i,k] \cdot L[k,k]^* \cdot L[k,j]$
      endfor
    endfor
  endfor
end

Transitive Hülle

Algorithmus Warshall-Algorithmus: Berechnung transitiver Hülle

```
procedure Warshall (var A : Adjazenzmatrix)
Eingabe : Graph als Adjazenzmatrix (A[i,j]) ∈ Bool_{n×n}
begin
  for k := 1 to n do
    for i := 1 to n do
      for j := 1 to n do
        if (A[i,k] = 1) and (A[k,j] = 1) then
          A[i,j] := 1
        endif
      endfor
    endfor
  endfor
end
```

Transitive Hülle?

Algorithmus Ist dieses Verfahren korrekt?

```
procedure Warshall (var A : Adjazenzmatrix)
Eingabe : Graph als Adjazenzmatrix (A[i,j]) ∈ Bool_{n×n}
begin
  for i := 1 to n do
    for j := 1 to n do
      for k := 1 to n do
        if (A[i,k] = 1) and (A[k,j] = 1) then
          A[i,j] := 1
        endif
      endfor
    endfor
  endfor
end
```

Korrektheit: Warshall

Die Korrektheit des Warshall-Algorithmus folgt aus folgenden Invarianten:

1. Nach dem k-ten Durchlauf der ersten **for**-Schleife gilt $A[i,j] = 1$, falls ein Pfad von i nach j über Knoten mit Nummern $\leq k$ existiert.

   Beachte: k steht ganz außen!

2. Gilt $A[i,j] = 1$, so existiert ein Pfad von i nach j.

Trägt man in die Adjazenz-Matrix vom Warshall-Algorithmus Kantengewichte statt Boolesche Werte ein, so entsteht der Floyd-Algorithmus zur Berechnung kürzester Wege:

Floyd-Algorithmus

Algorithmus Floyd: Alle kürzesten Wege in einem Graphen

procedure Floyd (**var** A : Adjazenzmatrix)
Eingabe : Gewichteter Graph als Adjazenzmatrix $A[i,j] \in (\mathbb{N} \cup \infty)_{n \times n}$,
wobei $A[i,j] = \infty$ bedeutet, dass es keine Kante von i nach j gibt.

```
begin
  for k := 1 to n do
    for i := 1 to n do
      for j := 1 to n do
        A[i,j] := min{A[i,j], A[i,k] + A[k,j]};
      endfor
    endfor
  endfor
end
```

Floyd-Algorithmus

Der Floyd-Algorithmus liefert ein korrektes Ergebnis auch wenn die Gewichte negativ sind, vorausgesetzt dass keine negative Schleifen vorhanden sind.

Zeitaufwand von Warshall und Floyd ist $\Theta(n^3)$.

„Verbesserung“ dadurch, dass vor der j-Schleife zuerst getestet wird, ob $A[i,k] = 1$ (bzw. ob $A[i,k] < \infty$) gilt.

Damit erreicht man den Aufwand $\mathcal{O}(n^3)$:

Floyd-Algorithmus

Algorithmus Floyd-Algorithmus in $\mathcal{O}(n^3)$

procedure Floyd (**var** A : Adjazenzmatrix)
Eingabe : Adjazenzmatrix $A[i,j] \in (\mathbb{N} \cup \infty)_{n \times n}$
begin
  for $k := 1$ **to** n **do**
    for $i := 1$ **to** n **do**
      if $A[i,k] < \infty$ **then**
        for $j := 1$ **to** n **do**
          $A[i,j] := \min\{A[i,j], A[i,k] + A[k,j]\}$;
        endfor
      endif
    endfor
  endfor
end

Floyd-Algorithmus

Algorithmus Floyd-Algorithmus mit negativen Zyklen

procedure Floyd (**var** A : Adjazenzmatrix)
Eingabe : Adjazenzmatrix $A[i,j] \in (\mathbb{Z} \cup \{\infty, -\infty\})_{n \times n}$
begin
  for $k := 1$ **to** n **do**
    for $i := 1$ **to** n **do**
      if $A[i,k] < \infty$ **then**
        for $j := 1$ **to** n **do**
          if $A[k,j] < \infty$ **then**
            if $A[k,k] < 0$ **then** $A[i,j] := -\infty$
              else $A[i,j] := \min\{A[i,j], A[i,k] + A[k,j]\}$
            endif
          endif
  endfor endif endfor endfor
end

Transitive Hülle und Matrixmultiplikation

Sei $A = (a_{i,j})_{1 \leq i,j \leq n}$ die Adjazenzmatrix eines gerichteten Graphen mit der Knotenmenge $\{1, \ldots, n\}$, d.h.

$$a_{i,j} = \begin{cases} 1 & \text{falls eine Kanten von } i \text{ nach } j \text{ existiert} \\ 0 & \text{sonst} \end{cases}$$

Der Warshall-Algorithmus berechnet den reflexiven transitiven Abschluss A^* in $\mathcal{O}(n^3)$ Schritten.

Hierbei ist $A^* = \sum_{k \geq 0} A^k$, wobei $A^0 = I_n$ die Einheitsmatrix ist, und $\vee$ als Addition boolescher Matrizen interpretiert wird.

Wir addieren also wie folgt: $0 + 0 = 0$, $0 + 1 = 1 + 0 = 1 + 1 = 1$.

Mit Induktion folgt: $A^k(i,j) = 1 \iff \exists$ Weg der Länge k von i nach j.

Hieraus folgt $A^* = \sum_{k=0}^{n-1} A^k$.

Transitive Hülle $\leq$ Matrixmultiplikation

Setze $B = I_n + A$. Dann gilt $A^* = B^m$ für alle $m \geq n - 1$.

Also reicht es, eine Matrix $\lceil \log_2(n-1) \rceil$-mal zu quadrieren, um A^* zu berechnen.

Sei $M(n)$ der Aufwand, zwei boolesche $n \times n$-Matrizen zu multiplizieren, und sei $T(n)$ der Aufwand, die reflexive transitive Hülle zu berechnen.

Dann gilt also:

$$T(n) \in \mathcal{O}(M(n) \cdot \log n).$$

Hieraus folgt für alle $\varepsilon > 0$ nach Strassen

$$T(n) \in \mathcal{O}(n^{\log_2(7)+\varepsilon}).$$

Matrixmultiplikation $\leq$ Transitive Hülle

Unter der plausiblen Annahme $T(3n) \in \mathcal{O}(T(n))$ gilt $M(n) \in \mathcal{O}(T(n))$:

Für alle Booleschen Matrizen A und B gilt:

$$\begin{pmatrix} 0 & A & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{pmatrix}^* = \begin{pmatrix} I_n & A & AB \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}.$$

Unter der ebenfalls plausiblen Annahme, dass $M(2n) \geq (2+\varepsilon)M(n)$ für ein $\varepsilon > 0$, zeigen wir $T(n) \in \mathcal{O}(M(n))$.

Dies bedeutet: Die Berechnung der transitiven Hülle ist bis auf konstante Faktoren genauso aufwendig wie die Matrixmultiplikation.

Berechnung der transitiven Hülle

Eingabe: $E \in \mathrm{Bool}(n \times n)$

- Teile E in vier Teilmatrizen A, B, C, D so, dass A und D quadratisch sind und jede Matrix ungefähr die Größe $n/2 \times n/2$ hat:

$$E = \begin{pmatrix} A & B \\ C & D \end{pmatrix}.$$

- Berechne rekursiv D^*:
  Aufwand $T(n/2)$.
- Berechne $F = A + BD^*C$:
  Aufwand $\mathcal{O}(M(n/2)) \leq \mathcal{O}(M(n))$.
- Berechne rekursiv F^*:
  Aufwand $T(n/2)$.
- Setze

$$E^* = \left(\begin{array}{c|c} F^* & F^*BD^* \\ \hline D^*CF^* & D^* + D^*CF^*BD^* \end{array}\right).$$

Berechnung der transitiven Hülle

Damit erhalten wir die Rekursionsgleichung

$$T(n) \leq 2T(n/2) + c \cdot M(n) \qquad \text{für ein } c > 0.$$

Daraus ergibt sich:

$$\begin{aligned} T(n) &\leq c \cdot \left(\sum_{i \geq 0} 2^i \cdot M(n/2^i) \right) && \text{(mit Induktion)} \\ &\leq c \cdot \sum_{i \geq 0} \left(\frac{2}{2+\varepsilon} \right)^i \cdot M(n) && \text{(da } M(n/2) \leq \tfrac{1}{2+\varepsilon} M(n)\text{)} \\ &\in \mathcal{O}(M(n)). \end{aligned}$$

Einschub: Wie testet man $A \cdot B = C$?

Der beste aktuell bekannte Algorithmus zur Multiplikation zweier $(n \times n)$-Matrizen benötigt ca. $\Theta(n^{2,372873\ldots})$ viele arithmetische Operationen (Virginia Vassilevska Williams 2014).

Vermutung

Für jedes $\epsilon > 0$ existiert ein Algorithmus, der zwei $(n \times n)$-Matrizen in Zeit $O(n^{2+\varepsilon})$ multipliziert.

Seien nun 3 $(n \times n)$-Matrizen A, B und C gegeben.

Wieviele arithmetische Operationen benötigt man, um zu testen, ob $A \cdot B = C$ gilt?

Triviale Antwort: $O(n^{2,372873\ldots})$

Aber es geht besser!

Einschub: Wie testet man $A \cdot B = C$?

Satz 36 (Korec, Wiedermann 2014)

Seien A, B, C $(n \times n)$-Matrizen mit Einträgen aus $\mathbb{Z}$. Mittels $O(n^2)$ vieler Operationen kann überprüft werden, ob $A \cdot B = C$ gilt.

Beweis: Sei

$$\begin{aligned} A &= (a_{i,j})_{1 \leq i,j \leq n}, \\ B &= (b_{i,j})_{1 \leq i,j \leq n}, \\ C &= (c_{i,j})_{1 \leq i,j \leq n} \text{ und} \\ D &= (d_{i,j})_{1 \leq i,j \leq n} = A \cdot B - C. \end{aligned}$$

Offensichtlich gilt $A \cdot B = C$ genau dann, wenn D die Nullmatrix ist.

Sei x eine reellwertige Variable und betrachte den Spaltenvektor

$$v = (1, x, x^2, \ldots, x^{n-1})^T.$$

Einschub: Wie testet man $A \cdot B = C$?

Dann ist $D \cdot v = A \cdot B \cdot v - C \cdot v$ ein Spaltenvektor, dessen i-ter Eintrag das Polynom

$$p_i(x) = d_{i,1} + d_{i,2}x + d_{i,3}x^2 + \cdots + d_{i,n}x^{n-1}$$

ist. Also gilt $A \cdot B = C$ genau dann, wenn $p_i(x)$ das Nullpolynom für alle $1 \leq i \leq n$ ist.

Wir verwenden nun den folgenden Satz:

Cauchys Schranke

Sei $p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 \in \mathbb{R}[x]$ nicht das Nullpolynom und $a_n \neq 0$. Dann gilt für jede Nullstelle α von $p(x)$:

$$|\alpha| < 1 + \frac{\max\{|a_i| \mid 0 \leq i \leq n-1\}}{|a_n|}.$$

Einschub: Wie testet man $A \cdot B = C$?

Beweis von Cauchys Schranke:

Indem wir $p(x)$ durch das Polynom $x^n + \frac{a_{n-1}}{a_n}x^{n-1} + \cdots + \frac{a_1}{a_n}x + \frac{a_0}{a_n}$ ersetzen, genügt es, Cauchys Schranke für den Fall $a_n = 1$ zu zeigen.

Sei also $p(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ und

$$h = \max\{|a_i| \mid 0 \leq i \leq n-1\}.$$

Gelte $p(\alpha) = \alpha^n + a_{n-1}\alpha^{n-1} + \cdots + a_1\alpha + a_0 = 0$, d.h.

$$\alpha^n = -a_{n-1}\alpha^{n-1} - \cdots - a_1\alpha - a_0.$$

Zu zeigen: $|\alpha| < 1 + h$.

Wenn $|\alpha| \leq 1$ gilt, gilt auch $|\alpha| < 1 + h$ (falls $h = 0$ muss $\alpha = 0$ gelten).

Gelte also $|\alpha| > 1$.

Einschub: Wie testet man $A \cdot B = C$?

Es folgt:

$$\begin{aligned} |\alpha|^n &\leq |a_{n-1}| \cdot |\alpha|^{n-1} + \cdots + |a_1| \cdot |\alpha| + |a_0| \\ &\leq h \cdot (|\alpha|^{n-1} + \cdots + |\alpha| + 1) \\ &= h \cdot \frac{|\alpha|^n - 1}{|\alpha| - 1} \end{aligned}$$

Da $|\alpha| > 1$ gilt, folgt:

$$|\alpha| - 1 \leq h \cdot \frac{|\alpha|^n - 1}{|\alpha|^n} < h$$

Einschub: Wie testet man $A \cdot B = C$?

Sei $a = \max\{|a_{i,j}| \mid 1 \leq i,j \leq n\}$, $b = \max\{|b_{i,j}| \mid 1 \leq i,j \leq n\}$ und $c = \max\{|c_{i,j}| \mid 1 \leq i,j \leq n\}$.

Damit lassen sich die Beträge der Koeffizienten der Polynome $p_i(x)$ wie folgt abschätzen:

$$|d_{i,j}| = |\sum_{k=1}^{n} a_{i,k} b_{k,j} - c_{i,j}| \leq \sum_{k=1}^{n} |a_{i,k}| \cdot |b_{k,j}| + |c_{i,j}| \leq n \cdot a \cdot b + c.$$

Sei nun $d = n \cdot a \cdot b + c$ und $r = 1 + d$.

Wegen Cauchy's Schranke gilt für alle $1 \leq i \leq n$:

$$p_i(x) = 0 \;\Leftrightarrow\; p_i(r) = 0$$

Beachte: Während wir sehr einfach die obere Schranke d für die Beträge der Einträge von D finden können, ist es nicht so einfach eine untere Schranke für diese Beträge zu finden.

Einschub: Wie testet man $A \cdot B = C$?

Aber: Da die Matrizen A, B, C Einträge aus $\mathbb{Z}$ haben, gilt $p_i(x) \in \mathbb{Z}[x]$.

Also ist der Absolutbetrag des führenden Koeffizienten von $p_i(x)$ mindestens 1, falls $p_i(x)$ nicht das Nullpolynom ist.

Wir können daher mit folgenden Algorithmus testen, ob $A \cdot B = C$ gilt:

1. Berechne $a = \max\{|a_{i,j}|\}$, $b = \max\{|b_{i,j}|\}$, $c = \max\{|c_{i,j}|\}$ und $r = 1 + n \cdot a \cdot b + c$
   ($3n^2$ viele Vergleiche, 2 Additionen, 2 Multiplikationen)
2. Berechne den Spaltenvektor $u = (1, r, r^2, \ldots, r^{n-1})^T$.
   ($n - 2$ Multiplikationen)
3. Berechne $p := B \cdot u$, $s := A \cdot p$ und $t := C \cdot u$
   ($O(n^2)$ viele arithmetische Operationen)
4. Es gilt $A \cdot B = C$ genau dann, wenn $s = t$.

Parallele Algorithmen und NC

1. Einführung in parallele Architekturen
2. Die Klasse NC und parallele Matrix-Multiplikation
3. Parallele Berechnung der Präfixsummen
4. Parallele Integer-Addition, -Multiplikation und -Division
5. Parallele Berechnung der Determinante

Einführung in parallele Architekturen

- Parallele Random-Access-Maschinen (PRAM)
  - CRCW (Concurrent Read Concurrent Write)
  - CREW (Concurrent Read Exclusive Write)
  - EREW (Exclusive Read Exclusive Write)
  - ERCW (Exclusive Read Concurrent Write)
- Vektor-Maschinen
  - SIMD oder MIMD
- Boolsche und arithmetische Schaltkreise
  - DAG mit Ein- und Ausgabeknoten
  - Knoten für grundlegende bitweise und arithmetische Operationen

Die Klasse NC

NC bezeichnet die Klasse aller Probleme, die "effizient parallelisierbar" sind.

NC steht für ,,Nick's Class" (benannt nach Nick Pippenger).

Definition(en):

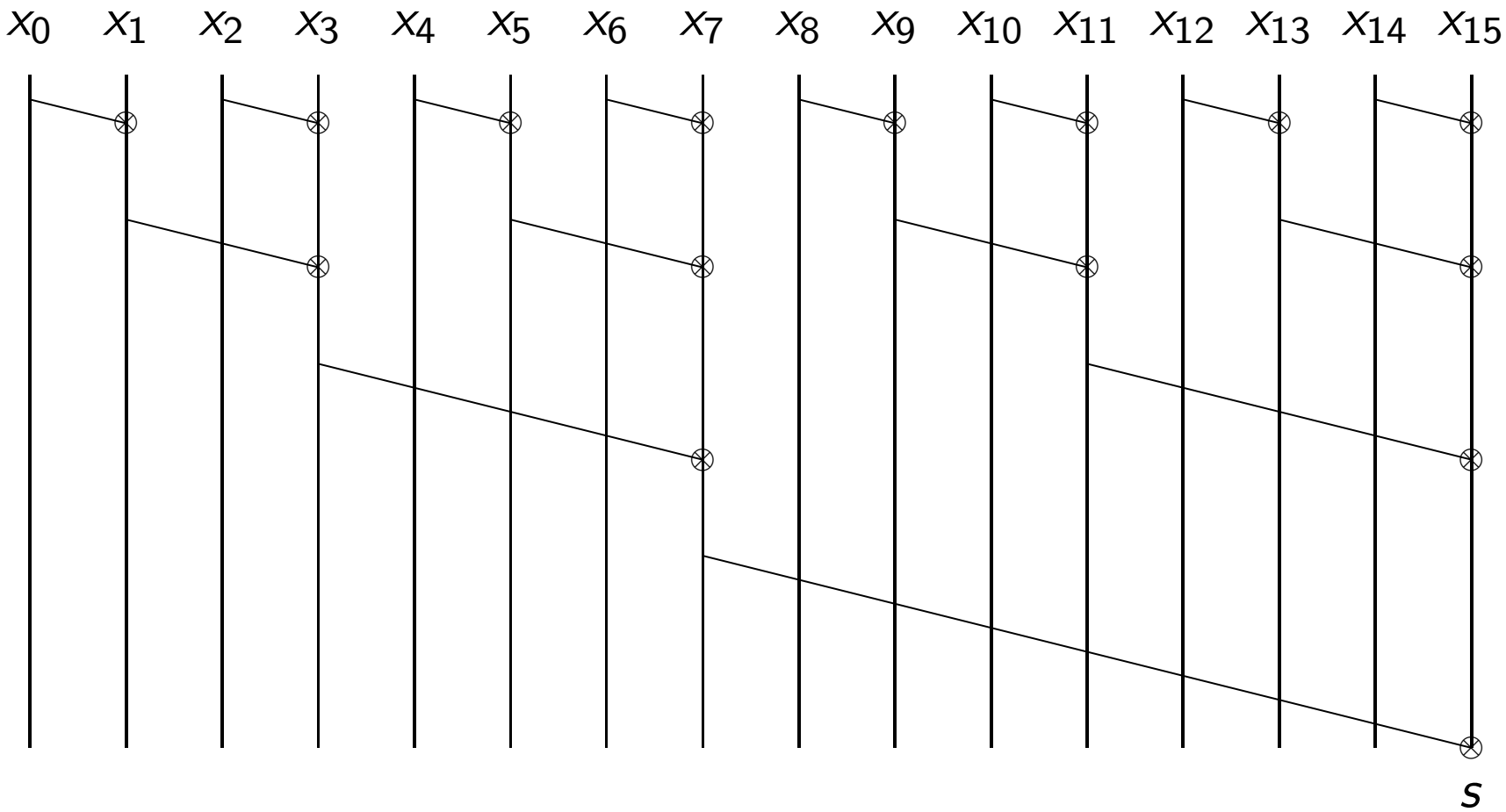
- Probleme, die mit einer PRAM mit $n^{O(1)}$ Prozessoren in Zeit $(\log n)^{O(1)}$ lösbar sind.
- Probleme, die mit einem bool'schen Schaltkreis der Tiefe $(\log n)^{O(1)}$ und Größe $n^{O(1)}$ lösbar sind.

Die Klasse ist robust gegenüber kleinen Änderungen am Maschinen-Modell.

Die Frage $NC \stackrel{?}{=} P$ ist noch offen.

Es gibt P-vollständige Probleme (z.B. das Auswerten boolescher Schaltkreise), von denen vermutet wird, dass sie nicht zu NC gehören.

Berechne die Summe $s = \sum_{i=0}^{n} x_i$



Summe $s = \sum_{i=0}^{n} x_i$ ist mit n Prozessoren in Zeit $\log n$ berechenbar.

Parallele Matrix-Multiplikaton

Theorem 37

Das Produkt von zwei $n \times n$-Matrizen ist mit n^3 Prozessoren in Zeit $1 + \log n$ berechenbar.

Beweis: Sei $A = (A_{ij})$ und $B = (B_{ij})$.

Dann gilt $(AB)_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$.

- Berechne mit n^3 Prozessoren alle n^3 Produkte.
- Teile jeder der n^2 Summen n Prozessoren zu.
- Berechne in $\log n$ Schritten alle n^2 Summen.

□

Parallele Präfixsummen

Das Problem:

- Eingabe: x_i $(0 \leq i \leq n-1)$
- Ausgabe: Alle Präfixsummen $y_i = \sum_{j=0}^{i} x_j$ für alle $0 \leq i \leq n-1$

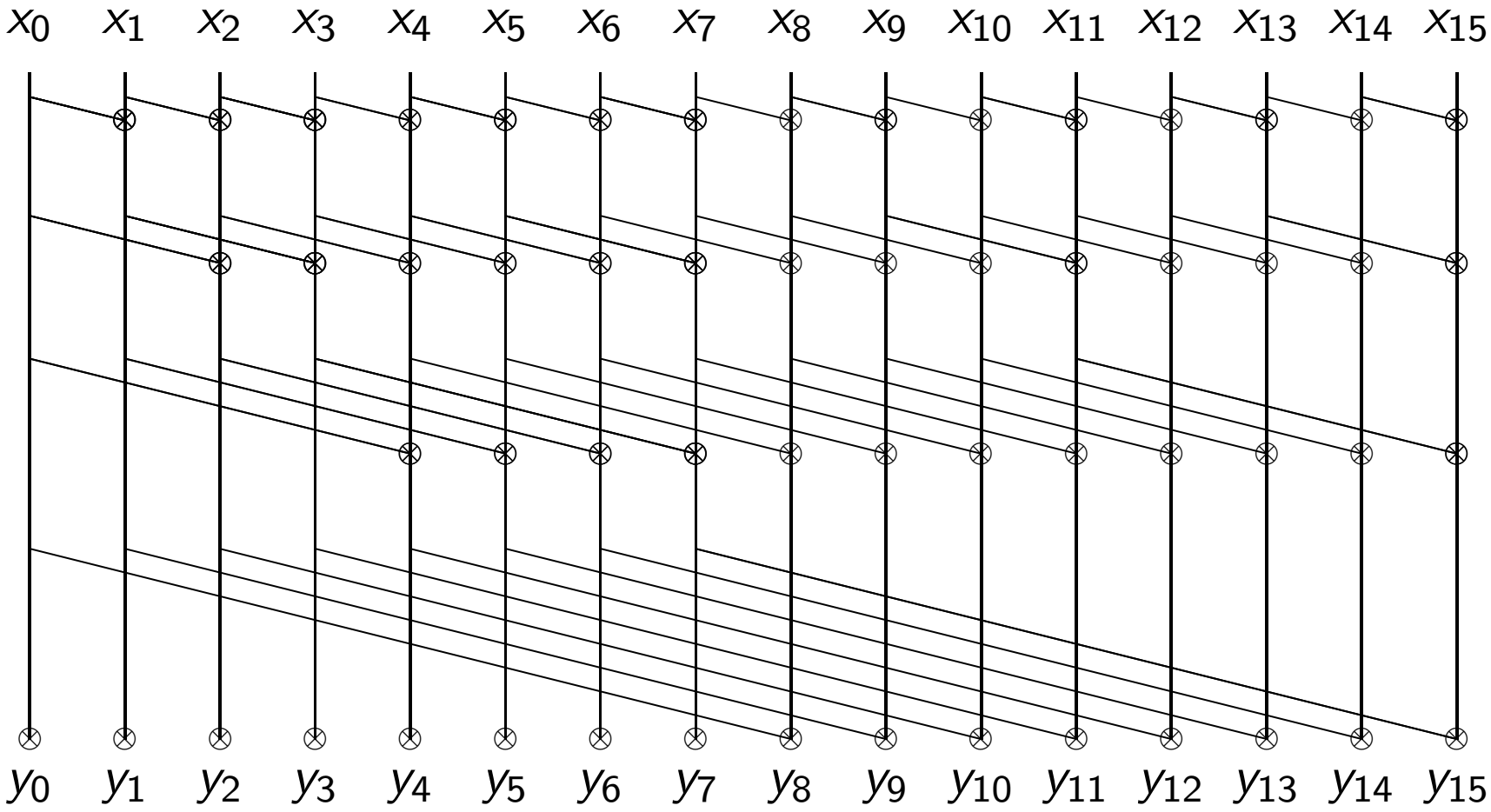
Theorem 38

Alle Präfixsummen können mit n Prozessoren in Zeit $\log n$ berechnet werden.

Beweis:

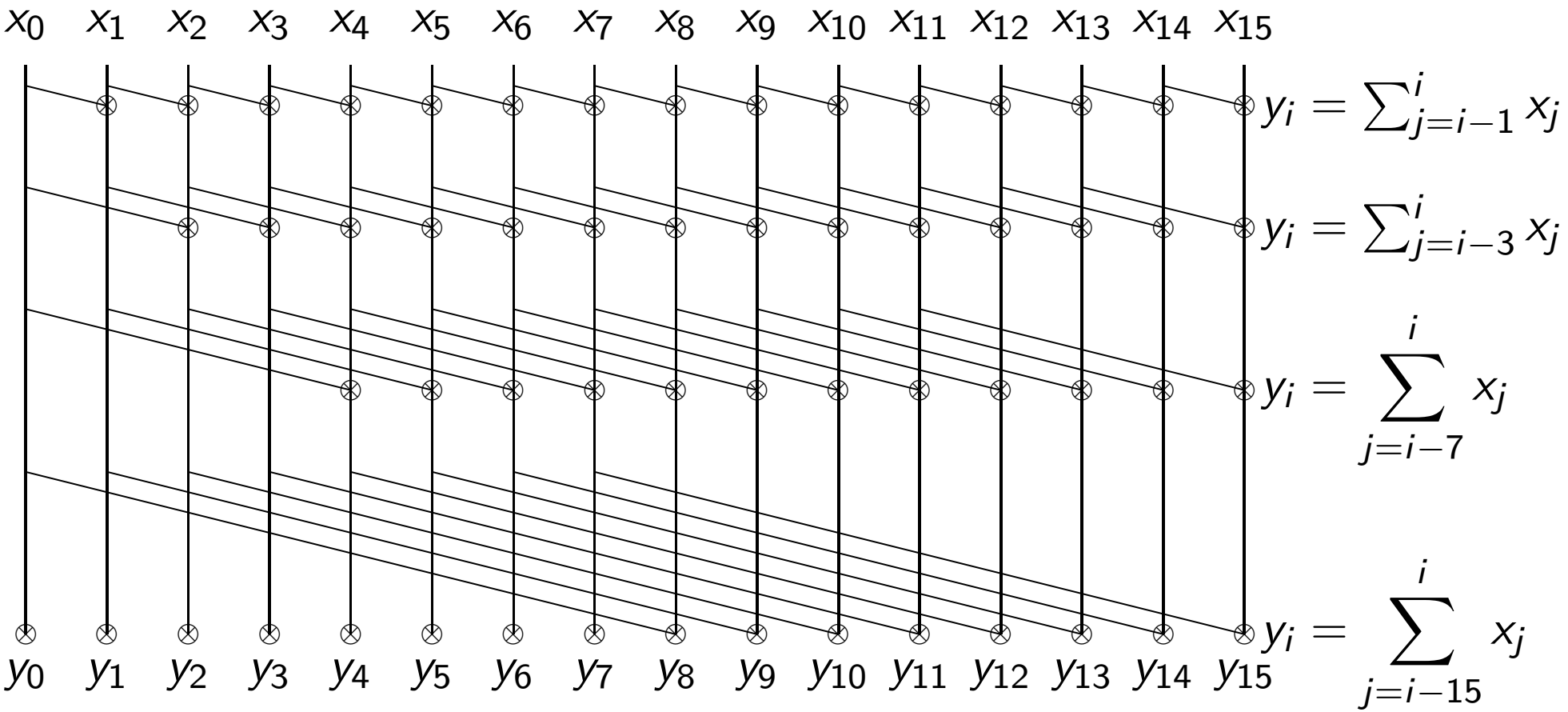
Zum Verständnis: Berechne die Summe $y_{n-1} = \sum_{j=0}^{n-1} x_j$

Parallele Präfixsummen



Parallele Präfixsummen

Sei $x_i = 0$ für alle $i < 0$.



Integer-Addition in NC

Theorem 39

Zwei n-bit-Binärzahlen können mit n Prozessoren in Zeit $\mathcal{O}(\log n)$ addiert werden.

Beweis: Seien $a_{n-1} \dots a_3 a_2 a_1 a_0$ und $b_{n-1} \dots b_3 b_2 b_1 b_0$ die Eingabezahlen (niederwertigstes Bit ist rechts).

Schritt 1: Berechne mit n Prozessoren in Zeit $\mathcal{O}(1)$ den Carry-Propagierungs-String $c_n c_{n-1} \dots c_3 c_2 c_1 c_0$:

$$c_i = \begin{cases} 0 & a_{i-1} = b_{i-1} = 0 \ \vee \ i = 0 \\ 1 & a_{i-1} = b_{i-1} = 1 \\ p & \text{else} \end{cases}$$

Integer-Addition in NC

Schritt 2: Berechne mit n Prozessoren in Zeit $\mathcal{O}(\log n)$ carry_i aus c_i mit dem parallelen Präfixsummen-Algorithmus unter Verwendung der folgenden assoziativen binären Operation:

$$\begin{aligned} 0 \cdot x &= 0 \\ 1 \cdot x &= 1 \\ p \cdot x &= x \end{aligned}$$

Beachte: Der parallele Präfixsummen-Algorithmus funktioniert für jede assoziative binäre Operation.

Schritt 3: Berechne das i-te Bit der Summe als das XOR von a_i, b_i und carry_i (wobei $a_n = b_n = 0$). □

Integer-Addition in NC

Beispiel:

$$\begin{array}{rcr} a & = & 100101011101011 \\ b & = & 110101001010001 \\ c & = & 1\mathtt{p}01010\mathtt{p}1\mathtt{ppp}0\mathtt{p}10 \\ carry & = & 1001010110000110 \\ \hline sum & = & 1011010100111100 \end{array}$$

Multiplikation von zwei Zahlen kann auf Addition von n Zahlen zurückgeführt werden.

Aus Theorem 39 folgt: Zwei n-bit-Binärzahlen können mit n^2 Prozessoren in Zeit $(\log n)^2$ multipliziert werden

Integer-Multiplikation in NC

Lemma 40

Aus drei n-bit-Binärzahlen a, b, c können mit n Prozessoren in Zeit $\mathcal{O}(1)$ zwei $(n+1)$-bit Binärzahlen d, e berechnet werden mit $a + b + c = d + e$.

Beweis:

$$
\begin{array}{r}
100111 \\
011100 \\
+111101 \\
\hline
\qquad 10 \\
\qquad 01 \\
\qquad 11 \\
\qquad 10 \\
\qquad 10 \\
+10 \\
\hline
\end{array}
\qquad
\begin{array}{r}
111101 \\
+000110 \\
\hline
\end{array}
$$

Integer-Multiplikation in NC

Theorem 41

Die Summe von n vielen n-bit Binärzahlen kann mit n^2 Prozessoren in Zeit $O(\log n)$ berechnet werden.

Beweis:

- Berechne in konstaner Zeit aus jedem Block von drei Zahlen einen neuen Block von zwei Zahlen mit einem Bit mehr.
- Wiederhole dies, bis nur noch zwei Binärzahlen übrig bleiben. Dies benötigt Zeit $O(\log n)$.
- Addiere die zwei übrig gebliebenen Zahlen in Zeit $O(\log n)$. □

Korollar

Zwei n-bit-Binärzahlen können mit n^2 Prozessoren in Zeit $\mathcal{O}(\log n)$ multipliziert werden.

Integer-Division in NC

Ziel: Bestimme zu zwei gegebenen Binärzahlen $s, t > 0$ mit $\leq n$ Bits die eindeutigen Zahlen q und r so, dass $s = qt + r$ und $0 \leq r < t$.

Wir werden dies in Zeit $\mathcal{O}((\log n)^2)$ mit $\mathcal{O}(n^4)$ Prozessoren durchführen.

Hauptwerkzeug: Newton-Verfahren zum Approximieren von Nullstellen.

Sei $f : \mathbb{R} \to \mathbb{R}$.

Rate einen Anfangswert x_0 und berechne die Folge $(x_i)_{i \geq 0}$ mit der Rekursionsgleichung

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, \text{ wobei } f' = df/dx$$

Mit Glück konvergiert die Folge $(x_i)_{i \geq 0}$ zu einer Nullstelle von f.

Integer-Division in NC

Nimm $f(x) = t - \frac{1}{x}$, also ist $\frac{1}{t}$ die eindeutige Nullstelle von f.

$f'(x) = \frac{1}{x^2}$, also wird Newtons Rekursionsgleichung zu

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{t - \frac{1}{x_i}}{\frac{1}{x_i^2}} = 2x_i - tx_i^2$$

Sei x_0 die eindeutige Zahl der Form $\frac{1}{2^j}$ $(j > 0)$ im Intervall $(\frac{1}{2t}, \frac{1}{t}]$.

Wir bestimmen x_0 in Zeit $\mathcal{O}(1)$ unter der Benutzung von n Prozessoren wie folgt: finde die eindeutige Zweierpotenz im Intervall $[t, 2t)$, drehe die Reihenfolge der Bits um und platziere einen Dezimalpunkt hinter der ersten 0.

Integer-Division in NC

Lemma 42

Die eindeutige Folge $(x_i)_{i\geq 0}$, die durch $x_{i+1} = 2x_i - tx_i^2$ erhalten wird (wobei x_0 die eindeutige Zahl der Form $\frac{1}{2^j}$ $(j > 0)$ im Interval $(\frac{1}{2t}, \frac{1}{t}]$ ist), erfüllt $0 \leq 1 - t \cdot x_i < \frac{1}{2^{(2^i)}}$.

Beweis: Induktion über i:

Nach Definition gilt $\frac{1}{2t} < x_0 \leq \frac{1}{t}$, d.h. $0 \leq 1 - tx_0 < \frac{1}{2}$.

Für $i \geq 0$ erhalten wir

$$1 - t \cdot x_{i+1} = 1 - t(2 \cdot x_i - t \cdot x_i^2) = (1 - t \cdot x_i)^2$$

Also $0 \leq 1 - t \cdot x_{i+1} < \left(\frac{1}{2^{(2^i)}}\right)^2 = \frac{1}{2^{(2^{i+1})}}$.

Integer-Division in NC

Aus dem vorangehenden Lemma erhalten wir für $k = \lceil \log(\log(s)) \rceil$:
$0 \leq 1 - t \cdot x_k < \frac{1}{s} \leq \frac{t}{s}$.

Also $0 \leq \frac{s}{t} - s \cdot x_k < 1$.

Es folgt, dass der ganzzahlige Anteil q von $\frac{s}{t}$ entweder $\lceil s \cdot x_k \rceil$ oder $\lfloor s \cdot x_k \rfloor$ ist (der richtige Wert kann durch einen Test bestimmt werden, bzw. durch eine einzige Multiplikation).

Der Rest r kann durch $r = s - qt$ bestimmt werden.

Integer-Division in NC

Schätzung der Laufzeit: Sei b_i die Anzahl der Bits von x_i.

Wegen $x_{i+1} = 2 \cdot x_i - t \cdot x_i^2$ gilt $b_{i+1} \leq 2b_i + n$.

Also $b_i \in O(2^i n)$ und x_k (sowie alle x_i mit $i < k$) hat höchstens $O(2^{\lceil \log(\log(s)) \rceil} n) \leq O(n^2)$ viele Bits.

Somit benötigt die Berechnung von $x_{i+1} = 2 \cdot x_i - t \cdot x_i^2$ aus x_i Zeit $\mathcal{O}(\log(n))$ mit $\mathcal{O}(n^4)$ Prozessoren.

Da $k \in \mathcal{O}(\log(n))$, ist $\mathcal{O}((\log n)^2)$ die Gesamtlaufzeit.

Csanskys Algorithmus zum Invertieren von Matrizen

Ziel: Ein NC-Algorithmus zum Invertieren einer $(n \times n)$-Matrix (falls die Matrix invertierbar ist).

Vereinbarung: Im Folgenden nehmen wir an, dass ein einzelner Prozessor eine einzelne arithmetische Operation in Zeit $\mathcal{O}(1)$ durchführen kann. Nach vorangehenden Überlegungen wirkt sich dies nicht auf die Klasse NC aus.

Bemerkung: Falls A eine $(n \times m)$-Matrix und B eine $(m \times p)$-Matrix sind, dann kann $A \cdot B$ mit $n \cdot m \cdot p$ Prozessoren in Zeit $\mathcal{O}(\log(m))$ berechnet werden.

1. Schritt: Invertieren unterer Dreiecksmatrizen.

Eine Matrix ist eine untere Dreiecksmatrix, falls (i) alle Einträge über der Hauptdiagonale 0 sind und (ii) auf der Hauptdiagonale keine 0 vorkommt.

Csanskys Algorithmus zum Invertieren von Matrizen

Betrachte eine $(n \times n)$ große untere Dreiecksmatrix $A = \begin{pmatrix} B & 0 \\ C & D \end{pmatrix}$.

B und D sind hier $(\frac{n}{2} \times \frac{n}{2})$ große untere Dreiecksmatrizen und C ist eine $(\frac{n}{2} \times \frac{n}{2})$-Matrix.

Dann gilt $A^{-1} = \begin{pmatrix} B^{-1} & 0 \\ -D^{-1}CB^{-1} & D^{-1} \end{pmatrix}$.

Diese Gleichung führt zu einem parallelen Algorithmus zum Berechnen von A^{-1} mit Zeitkomplexität

$$T(n) = T(\frac{n}{2}) + \mathcal{O}(\log(n))$$

unter Verwendung von n^3 Prozessoren.

Also $T(n) \in O(\log^2(n))$.

Csanskys Algorithmus zum Invertieren von Matrizen

2. Schritt: Lösen einer linearen Gleichungssystems:

Betrachte ein Gleichungssystem der Form

$$
\begin{aligned}
x_1 &= c_1 \\
x_2 &= a_{2,1}x_1 + c_2 \\
x_3 &= a_{3,1}x_1 + a_{3,2}x_2 + c_3 \\
&\vdots \\
x_n &= a_{n,1}x_1 + a_{n,2}x_2 + \cdots a_{n,n-1}x_{n-1} + c_n
\end{aligned}
$$

x_i sind Unbekannte, c_i und $a_{i,j}$ sind vorgegebene ganze Zahlen.

Sei $A = (a_{i,j})_{1 \leq i,j \leq n}$, wobei $a_{i,j} = 0$ für $i \leq j$, und sei $c = (c_1, \ldots, c_n)^T$.

Csanskys Algorithmus zum Invertieren von Matrizen

Das vorangehende Gleichungssystem ist äquivalent zu $Ax + c = x$, d.h. $(A - \mathrm{Id})x = -c$, wobei Id die $(n \times n)$-Identitätsmatrix ist.

Also $x = (\mathrm{Id} - A)^{-1}c$.

Da $\mathrm{Id} - A$ eine untere Dreiecksmatrix ist, können wir $(\mathrm{Id} - A)^{-1}$ bestimmen.

Also kann x in Zeit $\mathcal{O}(\log^2(n))$ unter Verwendung von n^3 Prozessoren berechnet werden.

Csanskys Algorithmus zum Invertieren von Matrizen

3. Schritt (der Hauptschritt): Berechnen des charakteristischen Polynoms

Das charakteristische Polynom einer $(n \times n)$-Matrix $A = (a_{i,j})_{1 \leq i,j \leq n}$ ist

$$\det(x \cdot \mathrm{Id} - A) = x^n - s_1 x^{n-1} + s_2 x^{n-2} - \cdots + (-1)^n s_n =$$
$$= \prod_{i=1}^{n} (x - \lambda_i)$$

wobei $\lambda_1, \ldots, \lambda_n$ die mit Vielfachheiten gezählten Eigenwerte von A sind.

Der Koeffizient s_1 ist die Spur von A:

$$s_1 = \mathrm{tr}(A) = \sum_{i=1}^{n} \lambda_i = \sum_{i=1}^{n} a_{i,i}$$

Csanskys Algorithmus zum Invertieren von Matrizen

Lemma 43

λ_i^m ist ein Eigenwert von A^m mit derselben Vielfachheit wie λ_i von A.

Beweis: Übung

Also $\operatorname{tr}(A^m) = \sum_{i=1}^{n} \lambda_i^m$.

Durch Einsetzen von $x = 0$ in das charakteristische Polynom erhalten wir

$$s_n = (-1)^n \det(-A) = \det(A) = \prod_{i=1}^{n} \lambda_i.$$

Für alle $1 \leq i \leq n$ erhalten wir $s_k = \sum_{1 \leq i_1 < \cdots < i_k \leq n} \lambda_{i_1} \lambda_{i_2} \cdots \lambda_{i_k}$.

Csanskys Algorithmus zum Invertieren von Matrizen

Sei $f_k^m = \sum_{\substack{1 \le i_1 < \cdots < i_k \le n \\ j \notin \{i_1, \ldots, i_k\}}} \lambda_{i_1} \lambda_{i_2} \cdots \lambda_{i_k} \lambda_j^m$

Also $f_k^0 = (n-k) s_k$, $f_0^m = \operatorname{tr}(A^m)$ und

$$\begin{aligned} s_k \cdot \operatorname{tr}(A^m) &= \left(\sum_{1 \le i_1 < \cdots < i_k \le n} \lambda_{i_1} \lambda_{i_2} \cdots \lambda_{i_k} \right) \cdot \sum_{j=1}^{m} \lambda_j^m \\ &= \sum_{\substack{1 \le i_1 < \cdots < i_k \le n \\ j \notin \{i_1, \ldots, i_k\}}} \lambda_{i_1} \lambda_{i_2} \cdots \lambda_{i_k} \lambda_j^m + \sum_{\substack{1 \le i_1 < \cdots < i_k \le n \\ j \in \{i_1, \ldots, i_k\}}} \lambda_{i_1} \lambda_{i_2} \cdots \lambda_{i_k} \lambda_j^m \\ &= f_k^m + f_{k-1}^{m+1} \end{aligned}$$

Csanskys Algorithmus zum Invertieren von Matrizen

Daraus folgt, dass

$$\begin{aligned} & s_k \cdot \mathrm{tr}(A^0) - s_{k-1} \cdot \mathrm{tr}(A^1) + s_{k-2} \cdot \mathrm{tr}(A^2) - \cdots \\ & \qquad + (-1)^{k-1} s_1 \cdot \mathrm{tr}(A^{k-1}) + (-1)^k \mathrm{tr}(A^k) \\ & = (f_k^0 + f_{k-1}^1) - (f_{k-1}^1 + f_{k-2}^2) + \cdots \\ & \qquad + (-1)^{k-1} (f_1^{k-1} + f_0^k) + (-1)^k f_0^k \\ & = f_k^0 = (n-k) s_k \end{aligned}$$

und weiter, dass (zur Erinnerung: $\mathrm{tr}(A^0) = n$)

$$\begin{aligned} s_k = \frac{1}{k} \Big(& s_{k-1} \mathrm{tr}(A^1) - s_{k-2} \mathrm{tr}(A^2) + \cdots \\ & - (-1)^{k-1} s_1 \mathrm{tr}(A^{k-1}) - (-1)^k \mathrm{tr}(A^k) \Big) . \end{aligned}$$

Csanskys Algorithmus zum Invertieren von Matrizen

Wir können also die Koeffizienten s_k des charakteristischen Polynoms wie folgt berechnen:

1. Berechne die Potenzen $A^1, A^2, \cdots, A^n$ in Zeit $\mathcal{O}(\log^2(n))$ unter Verwendung von n^4 Prozessoren.
2. Berechne $\mathrm{tr}(A^1), \ldots, \mathrm{tr}(A^n)$ in Zeit $\mathcal{O}(\log(n))$ unter Verwendung von n^2 Prozessoren.
3. Löse das obige Gleichungssystem in Zeit $\mathcal{O}(\log^2(n))$ unter Verwendung von n^3 Prozessoren.

Bemerkung: Dies ist nur möglich, wenn die Charakteristik p des zugrunde liegenden Körpers größer als n ist, da dann $\frac{1}{k}$ für alle $1 \leq k \leq n$ existiert.

Csanskys Algorithmus zum Invertieren von Matrizen

4. Schritt (der letzte Schritt): Invertieren beliebiger nicht-singulärer Matrizen

Satz von Cayley-Hamilton: Jede quadratische Matrix erfüllt ihre eigene charakteristische Gleichung, d.h.

$$A^n - s_1 \cdot A^{n-1} + s_2 \cdot A^{n-2} - \cdots + (-1)^{n-1} s_{n-1} \cdot A + (-1)^n s_n \cdot \mathrm{Id} = 0$$

Falls A^{-1} existiert ($\Longleftrightarrow s_n \neq 0$) gilt also:

$$A^{-1} = \frac{(-1)^{n-1}}{s_n} \left(A^{n-1} - s_1 A^{n-2} + s_2 A^{n-3} - \cdots + (-1)^{n-1} s_{n-1} \cdot \mathrm{Id} \right).$$

Also können wir A^{-1} in Zeit $\mathcal{O}(\log^2(n))$ unter Verwendung n^4 Prozessoren bestimmen, indem wir erst die Koeffizienten s_k berechnen und dann den obigen Ausdruck in Zeit $\mathcal{O}(\log(n))$ unter Verwendung von n^2 Prozessoren berechnen.

Randomisierte Algorithmen

Ein randomisierter Algorithmus (oder auch probabilistischer Algorithmus) verwendet Zufallsentscheidungen (Münzwürfe).

Beispiele:

- Quicksort mit zufällig gewählten Pivoelementen
- Quickselect für Medianbestimmung.

Bei randomisierten Algorithmen unterscheidet man zwischen

- Las Vegas Algorithmen: Diese liefern stets ein korrektes Ergebnis, die Rechenzeit (oder der Speicherbedarf) ist eine Zufallsgröße.

  Beispiel: Bei Quicksort mit zufällig gewählten Pivoelementen ist der Erwartungswert für die Laufzeit $O(n \log n)$.
- Monte Carlo Algorithmen: Bei diesen gibt es eine kleine Fehlerwahrscheinlichkeit.

Probabilistische Tests mit Polynomen

Sei $\mathbb{F}$ ein beliebiger Körper und seien $x_1, \ldots, x_n$ Variablen.

Mit $\mathbb{F}[x_1, \ldots, x_n]$ bezeichnen wir den Ring der Polynome mit Variablen $x_1, \ldots, x_n$ und Koeffizienten aus $\mathbb{F}$.

Seien $a_1, \ldots, a_k \in \mathbb{F}$ und

$$p(x_1, \ldots, x_n) = \sum_{i=1}^{k} a_i \prod_{j=1}^{n} x_j^{e_{i,j}} \in \mathbb{F}[x_1, \ldots, x_n].$$

Dann gilt $\deg(p) = \max\{e_{i,1} + e_{i,2} + \cdots + e_{i,n} \mid 1 \leq i \leq k\}$.

Satz 44

Sei $S \subseteq \mathbb{F}$ endlich und $p(x_1, \ldots, x_n) \in \mathbb{F}[x_1, \ldots, x_n] \setminus \{0\}$, d.h. p ist nicht das Null-Polynom. Dann hat die Gleichung $p(x_1, \ldots, x_n) = 0$ höchstens $\deg(p) \cdot |S|^{n-1}$ Lösungen in S^n.

Probabilistische Tests mit Polynomen

Beweis: Induktion über n und $\deg(p)$.

Fall 1: $n = 1$, d.h. p ist ein Polynom mit einer Variable (dies schließt den Fall $\deg(p) = 0$ mit ein, d.h. $p \in \mathbb{F} \setminus \{0\}$).

Ein Polynom mit einer Variable hat höchstens $\deg(p) = \deg(p) \cdot |S|^{n-1}$ Lösungen in S.

Fall 2: $\deg(p) = 1$, d.h. p hat die Form $a + a_1 x_1 + \cdots + a_n x_n$.
Da $p \neq 0$ und $\deg(p) \neq 0$, gibt es ein i mit $a_i \neq 0$. O.E.d.A. nehmen wir an, dass $a_1 \neq 0$.

Dann ist $p = 0$ äquivalent zu $x_1 = \frac{1}{a_1}\left(-a - \sum_{i=2}^{n} a_i x_i\right)$.

Es gibt genau $|S|^{n-1}$ Belegungen für $x_2, \ldots, x_n$ aus S. Also hat $p = 0$ höchstens $|S|^{n-1} = \deg(p) \cdot |S|^{n-1}$ Lösungen in S^n.

Probabilistische Tests mit Polynomen

Fall 3: $\deg(p) \geq 2$ und $n \geq 2$.

Fall 3.1: p ist nicht irreduzibel, d.h. $p = q \cdot r$ mit $\deg(q) < \deg(p)$ und $\deg(r) < \deg(p)$.

Weder q noch r können das Null-Polynom sein.

Nach Induktion hat $q = 0$ höchstens $\deg(q) \cdot |S|^{n-1}$ Lösungen in S^n und $r = 0$ hat höchstens $\deg(r) \cdot |S|^{n-1}$ Lösungen in S^n.

Da $(a_1, \ldots, a_n) \in S^n$ genau dann eine Lösung von $q \cdot r = 0$ ist, wenn es eine Lösung von $q = 0$ oder eine Lösung von $r = 0$ ist, folgt, dass $p = 0$ höchstens

$$\deg(q) \cdot |S|^{n-1} + \deg(r) \cdot |S|^{n-1} = (\deg(q) + \deg(r)) \cdot |S|^{n-1} = \deg(p) \cdot |S|^{n-1}$$

Lösungen in S^n hat.

Probabilistische Tests mit Polynomen

Fall 3.2: p ist irreduzibel.

Sei $\overline{x} = (x_1, \ldots, x_{n-1})$, d.h. $p = p(\overline{x}, x_n)$.
Zu jedem $s \in S$ betrache das Polynom $p(\overline{x}, s) \in \mathbb{F}[\overline{x}]$.

Behauptung: $p(\overline{x}, s)$ ist nicht das Null-Polynom.

Um die Behauptung zu beweisen, nehmen wir an, dass $p(\overline{x}, s) = 0$.

Schreibe $p(\overline{x}, x_n)$ als ein Polynom mit einer einzigen Variable x_n und Koeffizienten aus $\mathbb{F}[\overline{x}]$.

Polynom-Division durch $x_n - s$ mit Rest liefert:

$$p(\overline{x}, x_n) = q(\overline{x}, x_n)(x_n - s) + r,$$

wobei r ein Polynom vom Grad 0 in x_n ist, d.h. $r \in \mathbb{F}[\overline{x}]$.

Indem man $x_n = s$ einsetzt, erhält man $r = 0$.

Also gilt $p(\overline{x}, x_n) = q(\overline{x}, x_n)(x_n - s)$, was der Irreduzibilität von p widerspricht (hier ist wichtig, dass $\deg(p) \geq 2$).

Probabilistische Tests mit Polynomen

Da $p(\overline{x}, s)$ nicht das Null-Polynom ist, können wir die Induktionsannahme auf $p(\overline{x}, s)$ anwenden:

$p(\overline{x}, s) = 0$ hat höchstens $\deg(p(\overline{x}, s)) \cdot |S|^{n-2} \leq \deg(p) \cdot |S|^{n-2}$ Lösungen in S^{n-1}.

Da es $|S|$ verschiedene Werte für s gibt, folgt, dass $p = 0$ höchstens $|S| \cdot \deg(p) \cdot |S|^{n-2} = \deg(p) \cdot |S|^{n-1}$ Lösungen in S^n hat.

Damit ist der Satz bewiesen. □

Satz von Zippel und Schwartz

Sei $p(x_1, \ldots, x_n)$ ein Polynom vom Grad d, das nicht das Null-Polynom ist, und Koeffizienten aus dem Körper $\mathbb{F}$, und sei $S \subseteq \mathbb{F}$ endlich. Wenn wir $(s_1, \ldots, s_n) \in S^n$ zufällig und gleichverteilt auswählen, gilt $\Pr(p(s_1, \ldots, s_n) = 0) \leq d/|S|$.

Anwendung: Perfekte Matchings

Sei $G = (V, E)$ ein ungerichteter Graph.

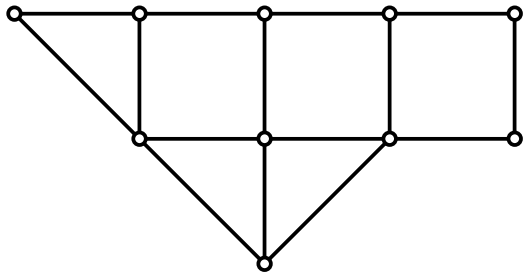
Ein Matching von G ist eine Teilmenge $M \subseteq E$ so, dass zwei verschiedene Kanten aus M keinen Knoten gemeinsam haben.

Ein Matching M von G ist ein perfektes Matching, falls $|M| = |V|/2$, d.h. jeder Knoten von G ist in genau einer Kante des Matchings enthalten.

Beachte: Ein perfektes Matching kann nur existieren, falls die Anzahl der Knoten von G gerade ist.

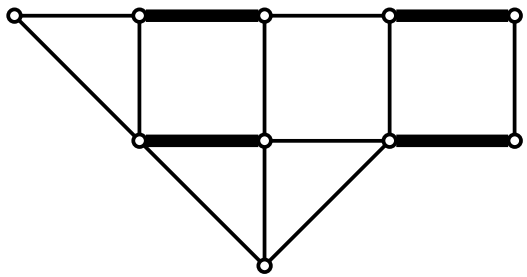
Graphentheorie: Matchings

Beispiel:



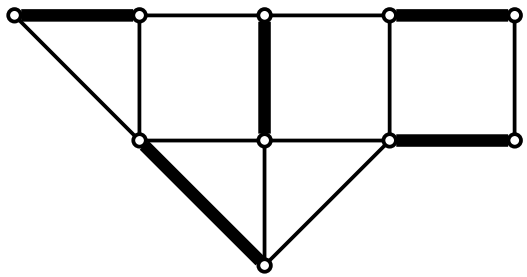
Graphentheorie: Matchings

Beispiel: ein Matching, welches jedoch nicht perfekt ist.



Graphentheorie: Matchings

Beispiel: ein perfektes Matching.



Anwendung: Perfekte Matchings

Wir suchen nach einem randomisierten NC-Algorithmus zum Testen, ob ein Graph ein perfektes Matching hat.

Dazu werden wir ein Polynom konstruieren, das genau dann nicht null ist, wenn G ein perfektes Matching hat.

Bemerkung: Es gibt einen deterministischen Algorithmus in Polynomialzeit zum Testen, ob ein Graph ein perfektes Matching hat, aber es ist nicht bekannt, ob dieses Problem in (deterministischen) NC liegt.

Anwendung: Perfekte Matchings

Sei $G = (V, E)$ ein ungerichteter Graph mit Knotenmenge $V = \{1, 2, \ldots, n\}$.

Zu jedem $\{u, v\} \in E$ mit $u < v$ sei $x_{u,v}$ eine Variable.

Die Tutte-Matrix von G ist die Matrix $T_G = (T_{u,v})_{1 \leq u,v \leq n}$ mit

$$T_{u,v} = \begin{cases} x_{u,v} & \text{falls } \{u, v\} \in E \text{ und } u < v \\ -x_{v,u} & \text{falls } \{u, v\} \in E \text{ und } u > v \\ 0 & \text{sonst} \end{cases}.$$

Anwendung: Perfekte Matchings

Wir interessieren uns für die Determinante von T_G, und verwenden hierfür die Leibniz-Formel:

Leibniz-Formel für die Determinante

Sei $A = (A_{i,j})$ eine $(n \times n)$-Matrix. Dann gilt

$$\det(A) = \sum_{\sigma \in S_n} \operatorname{sign}(\sigma) \prod_{i=1}^{n} A_{i,\sigma(i)},$$

wobei S_n die Menge aller Permutationen von $\{1, \ldots, n\}$ ist, und $\operatorname{sign}(\sigma)$ ist $+1$ (-1), falls σ das Produkt einer geraden (ungeraden) Anzahl von Transpositionen ist.

Hintergrund zu Permutationen kommt später.

Insbesondere ist $\det(T_G)$ ein Polynom mit $|E| \leq n^2$ Variablen vom Grad höchstens n.

Anwendung: Perfekte Matchings

Tuttes Satz

G hat ein perfektes Matching genau dann, wenn $\det(T_G)$ nicht das Nullpolynom ist.

Der Beweis kommt später.

Vorsicht: Wir können die Leibniz-Formel nicht dazu verwenden, das Polynom $\det(T_G)$ effizient auszurechnen, da die Summe über S_n aus $n!$ Summanden besteht.

Aber wir können probabilistisch testen, ob $\det(T_G)$ das Nullpolynom ist.

Anwendung: Perfekte Matchings

Satz 45

Es gibt einen randomisierten NC-Algorithmus (der Zeit $(\log(n))^{\mathcal{O}(1)}$ *unter Verwendung von* $n^{\mathcal{O}(1)}$ *Prozessoren benötigt), der einen ungerichteten Graphen* G *als Eingabe erhält und für den gilt:*

- *Falls* G *kein perfektes Matching hat, lehnt der Algorithmus* G *mit Wahrscheinlichkeit* 1 *ab.*
- *Falls* G *ein perfektes Matching hat, akzeptiert der Algorithmus* G *mit Wahrscheinlichkeit* $\geq 1/2$.

Bemerkung: Die Wahrscheinlichkeit $\frac{1}{2}$ kann auf $1 - \frac{1}{2^k}$ erhöht werden, indem der Algorithmus k mal wiederholt wird.

Anwendung: Perfekte Matchings

Beweis von Satz 45:

Der Algorithmus funktioniert wie folgt auf einem Graph $G = (V, E)$ mit $|V| = n$:

- Konstruiere die Tutte-Matrix T_G in Zeit $\mathcal{O}(1)$ unter Verwendung $|V|^2$ Prozessoren.
- Wähle zufällig einen Vektor $(a_1, a_2, \ldots, a_m) \in \{1, \ldots, 2n\}^m$, wobei $m = |E|$ die Anzahl der Variablen von T_G ist.
- Berechne $D = \det(T_G)(a_1, a_2, \ldots, a_m)$ in NC unter Benutzung von Csanskys Algorithmus.
- Falls $D \neq 0$ akzeptiere, ansonsten lehne ab.

Anwendung: Perfekte Matchings

Falls G kein perfektes Matching hat, folgt nach Tuttes Satz, dass $\det(T_G) = 0$. Somit lehnen wir mit Wahrscheinlichkeit 1 ab.

Falls G ein perfektes Matching hat, folgt nach Tuttes Satz, dass $\det(T_G)$ nicht das Null-Polynom ist.

Also folgt nach dem Satz von Zippel und Schwartz (mit $S = \{1, \ldots, 2n\}$), dass

$$\Pr(\text{Algorithmus akzeptiert nicht}) \leq \frac{1}{2n} \deg(\det(T_G)) \leq \frac{1}{2}$$

Anwendung: Perfekte Matchings

Es bleibt, Tuttes Satz zu beweisen:
$G = (\{1, \ldots, n\}, E)$ hat ein perfektes Matching $\Leftrightarrow \det(T_G) \neq 0$.

Hierbei ist $T_G = (T_{u,v})_{1 \leq u,v \leq n}$ mit

$$T_{u,v} = \begin{cases} x_{u,v} & \text{falls } \{u,v\} \in E \text{ und } u < v \\ -x_{v,u} & \text{falls } \{u,v\} \in E \text{ und } u > v \\ 0 & \text{sonst} \end{cases}.$$

Hintergrund zu Permutationen: Sei σ eine Permutation von $\{1, \ldots, n\}$. Dann kann σ eindeutig als Produkt von disjunkten Zyklen geschrieben werden.

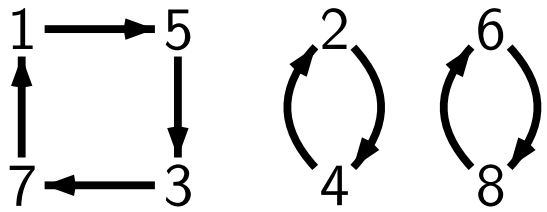
Sei $E_n = \{\sigma \in S_n \mid \sigma \text{ enthält nur Zyklen gerade Länge}\}$.

Anwendung: Perfekte Matchings

Beispiel: Sei die Permutation
$\sigma : \{1,2,3,4,5,6,7,8\} \to \{1,2,3,4,5,6,7,8\}$ wie folgt definiert:

a	1	2	3	4	5	6	7	8
$\sigma(a)$	5	4	7	2	3	8	1	6

Dann sieht σ wie folgt aus:



Wir schreiben auch

$$\sigma = (1,5,3,7)(2,4)(6,8) = (6,8)(2,4)(1,5,3,7) = \cdots .$$

Es gilt $\sigma \in E_n$, da alle Zyklen gerade Länge haben.

Anwendung: Perfekte Matchings

Eine Transposition ist eine Permutation der Form (a, b), d.h. sie besteht aus genau einem Kreis der Länge 2 und Fixpunkten (Kreisen der Länge 1).

Jede Permutation kann als ein Produkt von (nicht unbedingt disjunkten) Transpositionen geschrieben werden.

Im Beispiel der vorangehenden Folie haben wir

$$\sigma = (1,7)(3,5)(5,7)(2,4)(6,8)$$

(das Produkt muss von links nach rechts ausgewertet werden).

Das Vorzeichen einer Permutation σ ist

$$\operatorname{sign}(\sigma) = \begin{cases} +1 & \text{falls } \sigma \text{ das Produkt einer geraden Anzahl von Transpositionen ist} \\ -1 & \text{falls } \sigma \text{ das Produkt einer ungeraden Anzahl von Transpositionen ist} \end{cases}$$

Anwendung: Perfekte Matchings

Zur Erinnerung: $\det(T_G) = \sum_{\sigma \in S_n} \text{sign}(\sigma) \prod_{i=1}^{n} T_{i,\sigma(i)}$.

Lemma 46

$$\det(T_G) = \sum_{\sigma \in E_n} \text{sign}(\sigma) \prod_{i=1}^{n} T_{i,\sigma(i)}$$

Beweis: Wir zeigen

$$\sum_{\sigma \in S_n \setminus E_n} \text{sign}(\sigma) \prod_{i=1}^{n} T_{i,\sigma(i)} = 0.$$

Anwendung: Perfekte Matchings

Zunächst gilt: Für jede Permutation $\sigma \in S_n$, die einen Fixpunkt hat, d.h. $\sigma(j) = j$ für ein $1 \leq j \leq n$ (jede solche Permutation gehört zu $S_n \setminus E_n$) gilt $\prod_{i=1}^{n} T_{i,\sigma(i)} = 0$ (denn $T_{j,\sigma(j)} = 0$).

Wir betrachten nun noch alle Permutationen aus $S_n \setminus E_n$, die keinen Fixpunkt haben.

Sei $U_n = \{\sigma \in S_n \setminus E_n \mid \forall i : \sigma(i) \neq i\}$.

Sei $\sigma = \sigma_1 \sigma_2 \cdots \sigma_k \in U_n$, wobei die σ_i paarweise disjunkte Zyklen sind, und gelte $\prod_{i=1}^{n} T_{i,\sigma(i)} \neq 0$.

Also gilt $\{i, \sigma(i)\} \in E$ für alle $1 \leq i \leq n$ nach Definition der Tutte-Matrix.

O.E.d.A. hat $\sigma_1 = (a_0, a_1, \ldots, a_{\ell-1})$ ungerade Länge $\ell \geq 3$.

Anwendung: Perfekte Matchings

Sei $\tau = \sigma_1^{-1}\sigma_2 \cdots \sigma_k = (a_{\ell-1}, \ldots, a_1, a_0)\sigma_2 \cdots \sigma_k \in U_n$.

Es gilt für alle $0 \leq i \leq \ell - 1$ (wobei $i+1$ also $i+1$ mod ℓ zu lesen ist):

$$T_{a_i,\sigma(a_i)} = T_{a_i,a_{i+1}} = -T_{a_{i+1},a_i} = -T_{a_{i+1},\tau(a_{i+1})}$$

Also gilt:

$$\prod_{i=1}^{n} T_{i,\sigma(i)} = (-1)^{\ell} \prod_{i=1}^{n} T_{i,\tau(i)} = -\prod_{i=1}^{n} T_{i,\tau(i)}$$

Wir behaupten, dass $\operatorname{sign}(\sigma) = \operatorname{sign}(\tau)$, woraus dann folgt:

$$\operatorname{sign}(\sigma) \prod_{i=1}^{n} T_{i,\sigma(i)} = -\operatorname{sign}(\tau) \prod_{i=1}^{n} T_{i,\tau(i)}.$$

Anwendung: Perfekte Matchings

Es gilt $\sigma\tau = \sigma_1\sigma_2\cdots\sigma_k\sigma_1^{-1}\sigma_2\cdots\sigma_k = (\sigma_2\cdots\sigma_k)^2$.

Also $\operatorname{sign}(\sigma\tau) = +1$ (jede Permutation der Form ρ^2 ist ein Produkt einer geraden Anzahl von Transpositionen) und somit $\operatorname{sign}(\sigma) = \operatorname{sign}(\tau)$.

(Falls σ (τ) Produkt einer ungeraden Anzahl (geraden) Anzahl von Transpositionen wäre, dann wäre $\sigma\tau$ ein Produkt einer ungeraden Anzahl von Transpositionen.)

Wir können die Paarung zwischen σ und τ auf alle Permutationen in U_n erweitern, was zeigt, dass

$$\sum_{\sigma\in S_n\setminus E_n} \operatorname{sign}(\sigma)\prod_{i=1}^{n} T_{i,\sigma(i)} = 0.$$

Anwendung: Perfekte Matchings

Genauer: Definiere auf der Menge U_n eine Involution g (d.h. $g(\rho) \neq \rho$ und $g^2(\rho) = \rho$) wie folgt:

Sei $\sigma = \sigma_1 \sigma_2 \cdots \sigma_k \in U_n$, wobei die σ_i paarweise disjunkte Zyklen sind.

Nimm den eindeutigen Zyklus σ_j ungerader Länge, welcher das Minimum aller Elemente aus $\{1, \ldots, n\}$, welche auf ungeraden Zyklen liegen, enthält.

Definiere $g(\sigma) = \sigma_1 \cdots \sigma_{j-1} \sigma_j^{-1} \sigma_{j+1} \cdots \sigma_k$.

Nach den Argumentationen der vorangehenden Folie gilt:

$$\forall \sigma \in U_n : \operatorname{sign}(\sigma) \prod_{i=1}^{n} T_{i,\sigma(i)} = -\operatorname{sign}(g(\sigma)) \prod_{i=1}^{n} T_{i,g(\sigma)(i)}.$$

Dies beweist das Lemma. □

Anwendung: Perfekte Matchings

Wir können nun den Beweis von Tuttes Satz abschließen:

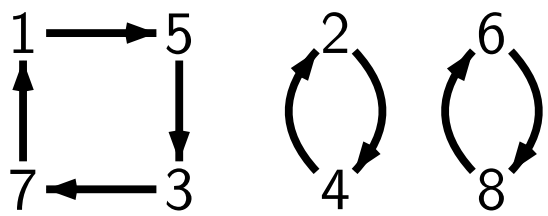
(1) Nimm an, dass $\det(T_G)$ nicht das Null-Polynom ist.

Nach dem vorangehenden Lemma existiert ein $\sigma \in E_n$ so, dass $\prod_{i=1}^{n} T_{i,\sigma(i)} \neq 0$.

Also gilt $\{i, \sigma(i)\} \in E$ für alle $1 \leq i \leq n$.

Wir erhalten ein perfektes Matching für G, indem wir jede zweite Kante von jedem Zyklus von σ auswählen.

Beispiel:



Eine Permutation $\sigma \in E_8$

Anwendung: Perfekte Matchings

Wir können nun den Beweis von Tuttes Satz abschließen:

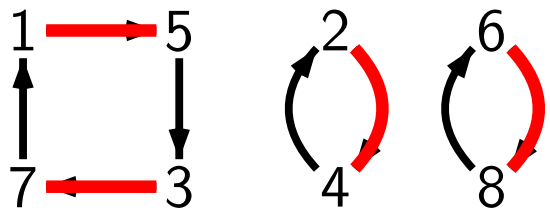
(1) Nimm an, dass $\det(T_G)$ nicht das Null-Polynom ist.

Nach dem vorangehenden Lemma existiert ein $\sigma \in E_n$ so, dass $\prod_{i=1}^{n} T_{i,\sigma(i)} \neq 0$.

Also gilt $\{i, \sigma(i)\} \in E$ für alle $1 \leq i \leq n$.

Wir erhalten ein perfektes Matching für G, indem wir jede zweite Kante von jedem Zyklus von σ auswählen.

Beispiel:



Eine Permutation $\sigma \in E_8$
Ein perfektes Matching, das durch Auswählen jeder zweiten Kante jedes Kreises erhalten wird

Anwendung: Perfekte Matchings

(2) Nimm an, dass G ein perfektes Matching $M \subseteq E$ hat.

Weise den Variablen $x_{u,v}$ ($\{u,v\} \in E$, $u < v$) Werte zu durch

$$x_{u,v} = \begin{cases} 1 & \text{falls } \{u,v\} \in M \\ 0 & \text{sonst} \end{cases}$$

Mit dieser Substitution wird T_G eine Matrix, die in jeder Zeile und in jeder Spalte genau einen von 0 verschiedenen Eintrag (1 oder -1) enthält.

Aus Leibniz Formel folgt, dass die Determinate ungleich 0 ist.

Also kann $\det(T_G)$ nicht das Null-Polynom sein.

Dies schließt den Beweis von Tuttes Satz ab.

Pattern-Matching mit Fingerprints

Sei Σ ein endliches Alphabet.

Sei $T = a_1 a_2 \cdots a_n$ ein Text und $P = b_1 b_2 \cdots b_m$ ein Pattern ($a_i, b_j \in \Sigma$, $m \leq n$).

Ziel: Finde alle Vorkommen von P in T, d.h. alle Positionen $1 \leq i \leq n - m + 1$ so, dass

$$T[i, i + m - 1] := a_i a_{i+1} \cdots a_{i+m-1} = P.$$

Der Algorithmus von Knuth, Morris und Pratt erreicht dies in sequentieller Zeit $\mathcal{O}(m + n)$.

Wir wollen hier einen einfachen randomisierten parallelen Algorithmus entwickeln.

Pattern-Matching mit Fingerprints

Wir nehmen im Folgenden an, dass $\Sigma = \{0, 1\}$.

Definiere $f : \Sigma \to \mathbb{Z}^{2\times 2}$ durch

$$f(0) = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \qquad f(1) = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

Wir erweitern f zu einem Homomorphismus $f : \Sigma^* \to \mathbb{Z}^{2\times 2}$

Lemma 47

Der Homomorphismus f ist injektiv. Falls $f(w) = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$ und $|w| = \ell$, dann $a_i \leq F_{\ell+1}$, wobei F_k die k-te Fibonacci-Zahl ist.

Pattern-Matching mit Fingerprints

Beweis: Es gilt

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} = \begin{pmatrix} a_1 & a_2 \\ a_1 + a_3 & a_2 + a_4 \end{pmatrix}$$

und

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} = \begin{pmatrix} a_1 + a_3 & a_2 + a_4 \\ a_3 & a_4 \end{pmatrix}$$

(1) Wenn $w \in \{0,1\}^*$ und $f(w) = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$, dann $a_1, a_4 > 0$, $a_2, a_3 \geq 0$.

Beweis von (1): Induktion über $|w|$.

Pattern-Matching mit Fingerprints

(2) Falls $w \neq \varepsilon$, dann $f(w) \neq f(\varepsilon) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.

Beweis von (2):

Falls $w = 0u$ und $f(u) = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$, dann $f(w) = \begin{pmatrix} a_1 & a_2 \\ a_1 + a_3 & a_2 + a_4 \end{pmatrix}$.

Falls $f(w) = f(\varepsilon)$, dann würde $a_1 = a_2 = a_3 = 0$ gelten, was ein Widerspruch zu $a_1 > 0$ ist (siehe (1)).

Für $w = 1u$ können wir ähnlich argumentieren.

Pattern-Matching mit Fingerprints

(3) $f(0u) \neq f(1v)$ für alle $u, v \in \{0,1\}^*$.

Beweis von (3): Sei $f(u) = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$ und $f(v) = \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix}$.

Falls $f(0u) = f(1v)$, erhalten wir:

$$\begin{pmatrix} a_1 & a_2 \\ a_1 + a_3 & a_2 + a_4 \end{pmatrix} = \begin{pmatrix} b_1 + b_3 & b_2 + b_4 \\ b_3 & b_4 \end{pmatrix}$$

Also:

$$a_1 = b_1 + b_3, \qquad a_1 + a_3 = b_3$$
$$a_2 = b_2 + b_4, \qquad a_2 + a_4 = b_4$$

$b_1 + a_3 = 0$, d.h. $a_3 = b_1 = 0$, ein Widerspruch zu $b_1 > 0$

$a_4 + b_2 = 0$, d.h. $a_4 = b_2 = 0$, ein Widerspruch zu $a_4 > 0$

Pattern-Matching mit Fingerprints

Nun können wir den Beweis von Lemma 47 beenden.

Angenommen $u, v \in \{0,1\}^*$, $u \neq v$ und $f(u) = f(v)$.

Wir leiten einen Widerspruch ab.

Die Matrizen $f(0)$ und $f(1)$ sind invertierbar ($\det(f(0)) = \det(f(1)) = 1$), also ist jede Matrix $f(x)$ invertierbar.

1. Fall: $u = vw$ mit $w \neq \varepsilon$.

Dann gilt $f(v)f(w) = f(u) = f(v)$, d.h. $f(w) = \mathrm{Id}_2 = f(\varepsilon)$.

Widerspruch zu (2)!

2. Fall: $v = uw$ mit $w \neq \varepsilon$: Analog

Pattern-Matching mit Fingerprints

3. Fall: Es existieren $u'v', w$ existieren mit $u = w0u'$ und $v = w1v'$.

Dann gilt $f(w)f(0u') = f(w)f(1v')$, d.h. $f(0u') = f(1v')$.

Widerspruch zu (3).

4. Fall: Es existieren $u'v', w$ existieren mit $u = w1u'$ und $v = w0v'$: Analog.

Dies beweist die erste Aussage des Lemmas.

Die zweite Aussage über die Fibonacci-Zahlen kann einfach durch Induktion über $|w|$ bewiesen werden. □

Pattern-Matching mit Fingerprints

Erste Idee zum Pattern-Matching: Vergleiche $f(P)$ mit $f(T[i, i+m-1])$ für alle $1 \leq i \leq n-m+1$.

Problem: $f(P)$ kann Einträge der Größe F_{m+1} haben, die $\mathcal{O}(m)$ Bit benötigen. Also benötigt das Vergleichen von $f(P)$ mit $f(T[i, i+m-1])$ Zeit $\mathcal{O}(m)$ und wir gewinnen nichts gegenüber dem direkten Vergleichen von P und $T[i, i+m-1]$.

Lösung: Rechne modulo einer Primzahl, die groß genug ist.

Zu einem Wort $w \in \{0,1\}^*$ und einer Primzahl p sei

$$f_p(w) = \begin{pmatrix} a_1 \bmod p & a_2 \bmod p \\ a_3 \bmod p & a_4 \bmod p \end{pmatrix}, \text{wobei } f(w) = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$$

Die Matrix $f_p(w)$ wird Fingerprint des Strings w (bezüglich der Primzahl p) genannt.

Pattern-Matching mit Fingerprints

Sei p eine Primzahl und seien $X, Y \in \{0,1\}^*$ zwei Strings.

Dann führt p zu einem ungültigen Match für (X, Y), falls $X \neq Y$ (d.h. $f(X) \neq f(Y)$, aber $f_p(X) = f_p(Y)$.

Zu einer natürlichen Zahl k bezeichnet $\pi(k)$ die Anzahl der Primzahlen im Intervall $\{1, \ldots, k\}$.

Dann gilt:

- $\frac{k}{\ln(k)} \leq \pi(k) \leq 1.2551 \frac{k}{\ln(k)}$
- Zu $u \leq 2^k$ ist die Anzahl verschiedener Primfaktoren von u höchstens $\pi(k)$, falls k groß genug ist ($k \geq 29$).

Pattern-Matching mit Fingerprints

Lemma 48

Für $1 \leq i \leq t$ seien X_i und Y_i Strings der Länge m.

Falls m groß genug ist, führt eine zufällig gewählte Primzahl p aus dem Intervall $\{1, \ldots, M\}$ zu einem ungültigen Match mindestens eines Paars (X_i, Y_i) mit einer Wahrscheinlichkeit von höchstens $\frac{\pi(\mathcal{O}(m \cdot t))}{\pi(M)}$.

Beweis: Für $1 \leq i \leq t$ sei

$$f(X_i) = \begin{pmatrix} a_{i,1} & a_{i,2} \\ a_{i,3} & a_{i,4} \end{pmatrix} \text{ und } f(Y_i) = \begin{pmatrix} b_{i,1} & b_{i,2} \\ b_{i,3} & b_{i,4} \end{pmatrix}.$$

Pattern-Matching mit Fingerprints

Dann gilt:

$\exists 1 \leq i \leq t : p$ führt zu einem ungültigen Match von (X_i, Y_i)

$\Longleftrightarrow$

$\exists 1 \leq i \leq t : f(X_i) \neq f(Y_i)$ und $f_p(X_i) = f_p(Y_i)$

$\Longrightarrow$

p teilt das Produkt $\prod \{|a_{i,j} - b_{i,j}| \mid 1 \leq i \leq t, 1 \leq j \leq 4, a_{i,j} \neq b_{i,j}\}$

$\Longrightarrow$

p teilt eine Zahl $u \leq F_{m+1}^{4t} \leq 2^{\lceil 4t \log(F_{m+1}) \rceil}$

Pattern-Matching mit Fingerprints

Zur Erinnerung: $F_{m+1} \in \mathcal{O}(\phi^{m+1})$, wobei $\phi = \frac{1+\sqrt{5}}{2}$.

Die Anzahl verschiedener Primfaktoren von $2^{\lceil 4t \log(F_{m+1}) \rceil}$ ist höchstens $\pi(\lceil 4t \log(F_{m+1}) \rceil) \leq \pi(\mathcal{O}(m \cdot t))$ (falls m groß genug ist).

$\rightsquigarrow$ Wahrscheinlichkeit für ein ungültiges Match $\leq \frac{\pi(\mathcal{O}(m \cdot t))}{\pi(M)}$. □

Pattern-Matching mit Fingerprints

Lemma 49

Seien (X_i, Y_i) Paare von Strings der Länge m ($1 \leq i \leq t$).

Sei $k \geq 1$ eine beliebig gewählte Konstante und $M = mt^k$.

Dann führt eine zufällig gewählte Primzahl $p \in \{1, \ldots, M\}$ zu einem ungültigen Match für mindestens ein Paar (X_i, Y_i) mit einer Wahrscheinlichkeit von höchstens $\mathcal{O}\left(\frac{1}{t^{k-1}}\right)$, falls m groß genug ist.

Beweis: Es gilt $\pi(\mathcal{O}(m \cdot t)) \leq \mathcal{O}\left(\frac{m \cdot t}{\ln(mt)}\right)$ und

$$\pi(M) = \pi(mt^k) \geq \frac{mt^k}{\ln(mt^k)} = \frac{mt^k}{\ln(m) + k\ln(t)}.$$

$$\rightsquigarrow \Pr(\exists \text{ ungült. Match}) \leq \mathcal{O}\left(\frac{m \cdot t \cdot (\ln(m) + k\ln(t))}{\ln(mt) \cdot m \cdot t^k}\right) \leq \mathcal{O}\left(\frac{1}{t^{k-1}}\right)$$

Pattern-Matching mit Fingerprints

Zur Erinnerung: $T = a_1 a_2 \cdots a_n$ (Text), $P = b_1 b_2 \cdots b_m$ (Pattern)

Zu einer Primzahl $p \in n^{\mathcal{O}(1)}$ können wir mit n Prozessoren in Zeit $\mathcal{O}(\log(n))$ alle Fingerprints $f_p(T[i, i+m-1])$ $(1 \leq i \leq n-m+1)$ berechnen:

Dazu berechnen wir mit dem Präfix-Summen-Algorithmus alle Produkte

$$\begin{aligned} R_i &= f_p(T[1,i]) = f_p(a_1) f_p(a_2) \cdots f_p(a_i) \\ R_i^{-1} &= f_p(T[1,i])^{-1} = f_p(a_i)^{-1} f_p(a_{i-1})^{-1} \cdots f_p(a_1)^{-1} \end{aligned}$$

$(1 \leq i \leq n)$ in Zeit $\mathcal{O}(\log(n))$ unter Verwendung von n Prozessoren.

Da wir modulo einer Primzahl der Größe $n^{\mathcal{O}(1)}$ rechnen, haben alle Zahlen, die in der Berechnung vorkommen, höchstens $\mathcal{O}(\log(n))$ viele Bit.

Berechne zuletzt $f_p(T[i, i+m-1]) = R_{i-1}^{-1} \cdot R_{i+m-1}$ in Zeit $\mathcal{O}(1)$ unter Verwendung von n Prozessoren.

Pattern-Matching mit Fingerprints

Satz 50

Sei k eine Konstante. Mittels $\mathcal{O}(n)$ vielen Prozessoren kann in Zeit $\mathcal{O}(\log(n))$ ein Array $MATCH[1,\ldots,n]$ mit folgenden Eigenschaften berechnet werden:

- *Falls $T[i, i+m-1] = P$, dann $MATCH[i] = 1$ mit Wahrscheinlichkeit 1.*
- *Falls $T[i, i+m-1] \neq P$, dann $MATCH[i] = 1$ mit einer Wahrscheinlichkeit, die durch $\mathcal{O}\left(\frac{1}{n^k}\right)$ beschränkt ist.*

Pattern-Matching mit Fingerprints

Beweis:

1. Sei $M = m \cdot n^{k+1} \leq n^{k+2}$.
2. Wähle eine zufällige Primzahl $p \in \{1, \ldots, M\}$.
3. Berechne $f_p(P)$ in Zeit $\mathcal{O}(\log(m))$ unter der Verwendung von $m \leq n$ Prozessoren.
4. Für $1 \leq i \leq n - m + 1$ berechne parallel $L_i := f_p(T[i, i + m - 1])$ unter Verwendung des Algorithmus der vorangehenden Folie.
5. Für $1 \leq i \leq n - m + 1$ setze parallel MATCH$[i] = 1$ genau dann, wenn $L_i = f_p(P)$.

Nach Lemma von Folie 49 ist die Wahrscheinlichkeit, dass ein Eintrag MATCH$[i]$ fälschlicherweise auf 1 gesetzt ist, beschränkt durch $\mathcal{O}\left(\frac{1}{n^k}\right)$.