Diskrete und Geometrische Algorithmen

Michael Neunteufel (e1241601)

30. Januar 2016

Inhaltsverzeichnis

Vorwort
Dieses Skript richtet sich an die Vorlesung Diskrete und geometrische Algorithmen vom Wintersemester 2015/16 und damit auch an [CLR$^+$13].
Die Code-Beispiele wurden teils sinngemäß, teils vollständig aus diesem Buch entnommen.
An dieser Stelle möchte ich meinen Mitstudentinnen und Mitstudenten danken, die mich auf Fehler und Unklarheiten aufmerksam gemacht haben. Ein besonderes Dankeschön an Georg Hofstätter für das ständige Korrekturlesen.

1 Einführung

1.1 Definition. Ein Algorithmus ist eine wohldefinierte Rechenvorschrift, welche eine Größe (eine Menge von Größen) als Eingabe in eine Größe (eine Menge von Größen) als Ausgabe umwandelt.

Ein Algorithmus ist ein Werkzeug zum Lösen eines Rechenproblems.
Formulierung eines Problems: Gewünschte Ein-Ausgabebeziehung.

1.2 Beispiel. Sortierproblem
Eingabe: $(a_1, \ldots, a_n), a_i \in \mathbb{R}$
Ausgabe: Eine Permutation $(a'_1, \ldots, a'_n)$ der Eingabe mit $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

1.3 Definition. Eine konkrete Eingabe, die alle Bedingungen erfüllt, heißt Instanz des Problems.

1.4 Definition. Ein Algorithmus heißt korrekt, wenn er für jede Instanz mit korrekter Ausgabe terminiert.

Spezifikation eines Algorithmus:

1. natürliche Sprache
2. Computerprogramme
3. Hardware-Design (z.B. Schaltkreis)
4. Pseudo-Code

Klassische Probleme:

1. Sortierproblem
2. kürzeste Wege
3. lineare Optimierung
4. konvexe Hülle
5. diskrete Fouriertransformation
6. Assignment-Problem
7. Codierung
8. Verschüsselung

Datenstrukturen:
Methode zum Speichern und Organisieren von Daten, um Zugriff und Modifikation zu erleichtern.

Algorithmus-Design:

1. Korrektheit
2. Effizienzanalyse (Laufzeit/Speicherbedarf etc.)

Schwere Probleme:
Probleme, für die keine effizienten Algorithmen bekannt sind.
$P \ldots$ in polynomieller Zeit lösbar
$NP \ldots$ in polynomieller Zeit verifizierbar
$P \subseteq NP$, NP-schwer $\supsetneq$ NP-vollständig
Stichwort Komplexitätsanalyse

2 Einführende Beispiele

2.1 Sortieren durch Einfügen (Insertion-Sort)

Wird auch in-situ oder in-place Algorithmus genannt.
Eingabe: $(a_1, \ldots, a_n)$
Ausgabe: Eine Permutation $(a'_1, \ldots, a'_n)$ mit $a'_1 \leq a'_2 \leq \cdots \leq a'_n$ von $(a_1, \ldots, a_n)$

Algorithm 1 INSERTION-SORT(A)

```
1: for j = 2 to |A| do
2:     key := A[j]
3:     i := j − 1
4:     while i > 0 and A[i] > key do
5:         A[i + 1] := A[i]
6:         i := i − 1
7:     end while
8:     A[i + 1] := key
9: end for
```

Korrektheit:
Schleifeninvariante:
Eigenschaft von A, die zu Beginn jeder Iteration erfüllt ist.
Invariante hier: $A[1 \ldots j-1]$ besteht aus $A[1 \ldots j-1]$ der Eingabe, aber sortiert.

1. Initialisierung: Wahr vor der ersten Iteration (entspricht Induktionsanfang)
2. Aufrechterhaltung: Falls wahr vor der Iteration, dann auch danach (entspricht Induktionsschritt)
3. Terminierung: Am Ende: Invariante $\hat{=}$ nützliche Eigenschaft von A um die Korrektheit zu beweisen

Behauptung: Der Algorithmus Sortieren durch Einfügen ist korrekt ($|A| = n$):

1. Initialisierung: $j = 2 \Rightarrow A[1 \ldots j-1] = A[1]$ ist offensichtlich sortiert
2. Aufrechterhaltung: Füge $A[j]$ zu $A[1 \ldots j-1]$ hinzu $\Rightarrow A[1 \ldots j]$ ist sortiert
3. Terminierung: $j = n + 1 \Rightarrow A[1 \ldots n]$ ist sortiert

Analyse:
Annahmen:

1. 1-Prozessor-Maschine
2. RAM (+, -, *, /, mod, load, copy, save, call, return, (un)bedingte Verzweigungen) hat konstante Laufzeit
3. Datentypen: $\mathbb{Z}$, Gleitkommazahlen
4. Datenwörter haben beschränkte Länge
5. ~~Speicherhierarchiekonzepte (Cache, virtueller Speicher)~~

Laufzeit:
Die Laufzeit ist abhängig von der Eingabegröße ($|A| = n$) und der Anzahl der Rechenschritte. Dabei hat jede Zeile konstante Kosten.

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + (c_5 + c_6) \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Best-Case: Eingabe bereits sortiert $\Rightarrow t_j = 1, \forall j = 2, \ldots, n$

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_8)n - (c_2 + c_3 + c_4 + c_8) = an + b$$

$\Rightarrow$ lineare Laufzeit

Worst-Case: $A[j]$ mit allen Elementen von $A[1 \ldots j-1]$ vergleichen:

$$\Rightarrow \forall j : t_j = j, \sum_{j=2}^{n} t_j = \frac{n(n+1)}{2} - 1, \quad \sum_{j=2}^{n} (t_j - 1) = \frac{n(n-1)}{2}$$

$$T(n) = \frac{1}{2}(c_4 + c_5 + c_6)n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4 - c_5 - c_6}{2} + c_8\right)n - (c_2 + c_3 + c_4 + c_8) = an^2 + bn$$

$\Rightarrow$ quadratische Laufzeit

Average-Case-Analyse:
Wahrscheinlichkeitsmodell der Eingabe
mittlere Laufzeit = Erwartungswert des Modells

Eingabe ... Zufallspermutation
d.h. $\mathbb{P}(\text{Eingabe} = \pi \in S_n | \text{ Eingabegröße} = \text{n})$

$$\text{mittlere Laufzeit } = \mathbb{E} \text{ Laufzeit } = \frac{1}{n!}(\sum_{\pi \in S_n} \text{Laufzeit für Eingabe } \pi)$$

Notation: $rg(\pi(j)) = k :\Leftrightarrow \pi(j)$ k-t größtes Element in $\pi(1), \ldots, \pi(n)$
$A[1 \ldots n]$ zufällige Permutation $\Rightarrow \mathbb{P}(rg(A[j]) = k \text{ in } A[1 \ldots j]) = \frac{1}{j} \quad 1 \leq k \leq j$

$A[1], A[2], \ldots, A[j-1] || A[j]$

$$\bar{t}_j = \frac{1}{j}(1 + 2 + \cdots + j) = \frac{1}{j}\frac{(j+1)j}{2} = \frac{j+1}{2}$$
$$\sum_{j=2}^{n} \bar{t}_j = \frac{1}{2}\sum_{j=2}^{n}(j+1) = \frac{1}{2}\sum_{j=3}^{n+1} j = \frac{1}{2}\left(\binom{n+2}{2} - 1 - 2\right) = \frac{(n+1)(n+2)}{4} - \frac{3}{2}$$
$$\sum_{j=2}^{n}(\bar{t}_j - 1) = \frac{1}{2}\sum_{j=2}^{n}(j-1) = \frac{1}{2}\sum_{j=1}^{n-1} j = \frac{n(n-1)}{4}$$
$$\overline{T}(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n^2}{4} + \frac{3}{4}n - 1\right) + (c_5 + c_6)\frac{n(n-1)}{4} + c_8(n-1)$$
$$= \frac{1}{4}(c_4 + c_5 + c_6)n^2 + \left(c_1 + c_2 + c_3 + \frac{3}{4}c_4 - \frac{1}{4}(c_5 + c_6) + c_8\right)n - (c_2 + c_3 + c_4 + c_8) = an^2 + bn + c$$
$a > 0 \Rightarrow$ quadratisch

Praxis: Nur Rechenintensive Operationen betrachten und nicht jede Zeile.
Insertion-Sort hat mittlere Laufzeit von $\Theta(n^2)$

2.1 Definition. $a_n = \Theta(b_n) :\Leftrightarrow \exists c_1, c_2 > 0 : c_1 b_n \leq a_n \leq c_2 b_n$

2.2 Sortieren durch Verschmelzen (Merge-Sort)

Insertion-Sort: Inkrementell
Merge-Sort: Divide & Conquer (Divide et Impera, Teile & Herrsche)

1. Eingabefolge von n Elementen in zwei Teilfolgen von $\approx \frac{n}{2}$ Elementen teilen.

2. Teilfolgen sortieren (kleinere Instanzen des selben Problems).

3. Verschmelzen/Zusammenfügen der sortierten Teilfolgen.

Hilfsprozedur Merge:
Eingabe: A Liste und p, q, r Indizes mit $p \leq q \leq r$.
Voraussetzung: $A[p \dots q]$ und $A[q+1 \dots r]$ sind bereits sortiert.
Ausgabe: $A[p \dots r]$ sortiert

Algorithm 2 MERGE(A, p, q, r)

```
1:  n_1 = q − p + 1
2:  n_2 = r − q
3:  seien L[1 ... n_1 + 1] und R[1 ... n_2 + 1] zwei neue Felder
4:  for i = 1 to n_1 do
5:      L[i] = A[p + i − 1]
6:  end for
7:  for j = 1 to n_2 do
8:      R[j] = A[q + j]
9:  end for
10: L[n_1 + 1] = ∞
11: R[n_2 + 1] = ∞
12: i = 1
13: j = 1
14: for k = p to r do
15:     if L[i] ≤ R[j] then
16:         A[k] = L[i]
17:         i = i + 1
18:     else
19:         A[k] = R[j]
20:         j = j + 1
21:     end if
22: end for
```

Korrektheit:
Schleifeninvariante: Vor jeder Iteration k gilt: $A[p \dots k-1]$ enthält die $k-p$ kleinsten Elemente von L und R, sortiert. $L[i]$, $R[j]$ sind die kleinsten Elemente von L bzw. R, welche nicht in A liegen.

Initialisierung: $k = p$:
$A[p \dots p-1]$ enthält die 0 kleinsten Elemente von L und R, sortiert. $L[1], R[1]$ kleinste Elemente von L bzw. R, nicht in A.

Aufrechterhaltung:
Fall 1: $L[i] \leq R[j]$
$L[i] \notin A$, $L[1 \dots i-1] \subseteq A$, $L[i], R[j] \notin A$, minimal. $A[p \dots k-1] : k-p$ kleinste Elemente von L und R, dann $A[k] := L[i] \Rightarrow A[p \dots k] : k-p+1$ kleinste Elemente von L und R, sortiert. $L[i+1] \notin A, R[j] \notin A$
Fall 2: analog

Terminierung: $k = r+1$:
$A[p \dots r] : r-p+1$ kleinste Elemente von L und R sortiert. $n_1 + n_2 + 2 = r - p + 3$
$L[n_1+1] = \infty \notin A, R[n_2+1] = \infty \notin A$

Laufzeit: i- und j-Schleife: $\Theta(n_1 + n_2) = \Theta(n)$, k-Schleife: $\Theta(n)$
$\Rightarrow$ MERGE hat eine Laufzeit von $\Theta(n)$
Analyse von Merge-Sort (Divide & Conquer):

Algorithm 3 MERGESORT(A, p, r)

```
1: if p < r then
2:     q = ⌊(p+r)/2⌋
3:     MERGESORT(A, p, q)
4:     MERGESORT(A, q+1, r)
5:     MERGE(A, p, q, r)
6: end if
```

Annahmen:

1. $n \leq c$ (genügend klein) $\rightarrow$ direkte Lösung $\Theta(1)$
2. Divide: a Teilprobleme der Größe $\frac{n}{b}$ (Merge-Sort: $a = b = 2$)

$$T(n) = \begin{cases} \Theta(1) & \text{, für } n \leq c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & \text{, sonst} \end{cases}$$

$D \ldots$ Divide, $C \ldots$ Combine

Merge-Sort: $n = 2^k \quad (k \in \mathbb{N})$
Divide: $q = \lfloor \frac{n}{2} \rfloor \Rightarrow D(n) = \Theta(1)$
Conquere: $2T\left(\frac{n}{2}\right)$
Merge: $C(n) = \Theta(n)$

$\Rightarrow T(n) = \Theta(n \log n)$ (Master-Theorem (siehe Abschnitt 4.4))

Heuristik: $(n = 2^k)$

$$\tilde{\mathrm{T}}(n) = \begin{cases} c & , n = 1 \\ 2\tilde{\mathrm{T}}(\frac{n}{2}) + cn & , n \geq 2 \end{cases}$$

$$\tilde{\mathrm{T}}(n) = 2\tilde{\mathrm{T}}(\frac{n}{2}) + cn = 4\tilde{\mathrm{T}}(\frac{n}{4}) + cn + 2\frac{cn}{2} = \cdots = cn + \frac{cn}{2}2 + \frac{cn}{4}4 + \cdots + 2^k\frac{cn}{2^k}$$

wobei $\frac{cn}{2^k} = c = \tilde{\mathrm{T}}(1)$

Anzahl der Summanden: $k + 1 = \log_2 n + 1 = \frac{\log n}{\log 2} + 1 = \Theta(\log n)$

Kosten pro Summand: $\Theta(n)$

$\Rightarrow \tilde{\mathrm{T}}(n) = \Theta(n \log n)$

3 Wachstum von Funktionen und elementare Kombinatorik

3.1 Asymptotischer Vergleich von Folgen

3.1 Definition. Seien a_n, b_n Folgen. Für folgende Definitionen reicht es aus, wenn es ein $n_0 \in \mathbb{N}$ gibt, sodass die Aussagen $\forall n \geq n_0$ gelten.

$$
\begin{aligned}
a_n &= \Theta(b_n) :\Leftrightarrow \exists c_1, c_2 > 0 : c_1|b_n| \leq |a_n| \leq c_2|b_n| \Leftrightarrow a_n = \mathcal{O}(b_n) \wedge a_n = \Omega(b_n) \\
a_n &= \mathcal{O}(b_n) :\Leftrightarrow \exists c \in \mathbb{R} : |a_n| \leq c|b_n| \\
a_n &= \Omega(b_n) :\Leftrightarrow b_n = \mathcal{O}(a_n) \\
a_n &= o(b_n) :\Leftrightarrow \frac{a_n}{b_n} \to 0 \\
a_n &= \omega(b_n) :\Leftrightarrow \frac{b_n}{a_n} \to 0 \\
a_n &\sim b_n :\Leftrightarrow \lim_{n\to\infty} \frac{a_n}{b_n} = 1 \\
a_n &\sim \sum_{m\geq 0} b_{m,n} :\Leftrightarrow \forall M \in \mathbb{N} : a_n \sim \sum_{m=0}^{M} b_{m,n}, \quad a_n - \sum_{m=0}^{M} b_{m,n} \sim b_{M+1,n}
\end{aligned}
$$

3.2 Beispiel. Stirling'sche Formel:

$$
n! \sim \left(\frac{n}{e}\right)^n \sqrt{2\pi n} \left(1 + \frac{1}{12n} + \frac{1}{280n^2} + \cdots + \sum_{m\geq 3} \frac{c_m}{n^m}\right) = \left(\frac{n}{e}\right)^n \sqrt{2\pi n} \left(1 + \Theta\left(\frac{1}{n}\right)\right)
$$

Achtung:
$a = b \Rightarrow b = a$, aber $a = \mathcal{O}(b_n) \nRightarrow \mathcal{O}(b_n) = a_n$
sondern: $a_n \in \mathcal{O}(b_n)$ (vergleiche Nebenklassen bei Gruppen: $n^2 + n + 1 = n^2 + \mathcal{O}(n)$)

3.3 Definition.
$f(n)$ ist von polynomiellen Wachstum $:\Leftrightarrow \exists k \in \mathbb{N} : f(n) = \mathcal{O}(n^k)$
$f(n)$ ist von polylogarithmischen Wachstum $:\Leftrightarrow \exists k \in \mathbb{N} : f(n) = \mathcal{O}((\log n)^k)$

3.2 Lösen elementarer Rekursionen

Homogene lineare Rekursion mit konstanten Koeffizienten k-ter Ordnung:
Eine Rekursion der Form

$$\forall n \geq 0 : c_k a_{n+k} + c_{k-1} a_{n+k-1} + \cdots + c_1 a_{n+1} + c_0 a_n = 0 \tag{1}$$

heißt homogene lineare Rekursion mit konstanten Koeffizienten k-ter Ordnung, wobei $c_0, c_1, \ldots, c_k \in \mathbb{R}$, mit $c_0 \neq 0, c_k \neq 0$.
Gesucht: $(a_n)_{n\geq 0}$

Falls zusätzlich $a_0, a_1, \ldots, a_{k-1}$ gegeben sind, hat man ein Anfangswertproblem und die $(a_n)_{n\geq 0}$ sind eindeutig bestimmt.

3.4 Satz. Seien $(a_n^{(1)})_{n\geq 0}, (a_n^{(2)})_{n\geq 0}$ Lösungen von (1). Dann ist $(k_1 a_n^{(1)} + k_2 a_n^{(2)})_{n\geq 0}$ ist ebenfalls eine Lösung von (1) $(\forall k_1, k_2 \in \mathbb{R})$.

Beweis:

$$\begin{aligned} &c_k(k_1 a_{n+k}^{(1)} + k_2 a_{n+k}^{(2)}) + \cdots + c_0(k_1 a_n^{(1)} + k_2 a_n^{(2)}) \\ &= k_1(c_k a_{n+k}^{(1)} + \cdots + c_0 a_n^{(1)}) + k_2(c_k a_{n+k}^{(2)} + \cdots + c_0 a_{n+k}^{(2)}) = 0 \end{aligned}$$

■

3.5 Korollar. Die Lösungsmenge L von (1) ist ein Vektorraum.

Dimension: Seien $a_0, a_1, \ldots, a_{k-1}$ beliebig, dann ist $(a_n)_{n\geq 0}$ eindeutig bestimmt und damit ist $\dim L = k$.

3.6 Bemerkung. Die c_k müssen nicht konstant sein, dürfen also auch von n abhängen.
z.B.: $a_n + n^2 a_{n-1} + n a_{n-2} = 0$

$(a_n^{(1)}), (a_n^{(2)}), \ldots, (a_n^{(k)})$ Lösungen von (1) sind linear unabhängig $\Leftrightarrow \begin{pmatrix} a_0^{(1)} \\ \vdots \\ a_{k-1}^{(1)} \end{pmatrix}, \ldots, \begin{pmatrix} a_0^{(k)} \\ \vdots \\ a_{k-1}^{(k)} \end{pmatrix}$

linear unabhängig sind $\Leftrightarrow \begin{vmatrix} a_0^{(1)} & \ldots & a_0^{(k)} \\ \vdots & & \vdots \\ a_{k-1}^{(1)} & \ldots & a_{k-1}^{(k)} \end{vmatrix} \neq 0$ (Wronski-Determinante)

3.7 Definition.
$\chi(\lambda) = c_k\lambda^k + c_{k-1}\lambda^{k-1} + \cdots + c_1\lambda + c_0 = 0$ heißt **Charakteristische Gleichung** von (1). $\chi(\lambda)$ heißt **charakteristisches Polynom** von (1).

Ansatz: $\lambda \neq 0, \quad a_n = \lambda^n : c_k\lambda^{n+k} + c_{k-1}\lambda^{n+k-1} + \cdots + c_1\lambda^{n+1} + c_0\lambda^n = 0$
$\chi(\lambda) = 0 \Rightarrow (\lambda^n)_{n\geq 0}$ ist Lösung von (1).

3.8 Lemma. Seien $\lambda_1, \lambda_2, \ldots, \lambda_k$ Nullstellen von $\chi(\lambda)$, paarweise verschieden. Dann sind $(\lambda_1^n)_{n\geq 0}, \ldots, (\lambda_k^n)_{n\geq 0}$ Lösungen von (1).
Beweis:

$$\begin{vmatrix} 1 & 1 & \ldots & 1 \\ \lambda_1 & \lambda_2 & \ldots & \lambda_k \\ \vdots & \vdots & & \vdots \\ \lambda_1^{k-1} & \lambda_2^{k-1} & \ldots & \lambda_k^{k-1} \end{vmatrix} = \prod_{i<j}(\lambda_j - \lambda_i) \neq 0 \quad \text{(Vandermonde-Determinante)}$$

■

3.9 Bemerkung. $L = \{k_1\lambda_1^n + \cdots + k_k\lambda_k^n |\, k_i \in \mathbb{R}, i = 1, \ldots, k\}$

3.10 Lemma. Sei λ l-fache Nullstellen von $\chi(x)$. Dann ist $(\lambda^n\chi(\lambda))^{(l-1)}\lambda^{l-1} = 0$ (genauer: $\lambda^{l-1}(\frac{d}{dx})^{l-1}(x^n\chi(x))|_{x=\lambda}$), also sind $(\lambda^n)_{n\geq 0}, (n\lambda^n)_{n\geq 0}, \ldots, ((n)_{l-1}\lambda^n)_{n\geq 0}$ Lösungen von (1).
Beweis:
Definiere: $(n)_i := n(n-1)\cdots(n-i+1)$
$\Rightarrow (n+k)_{l-1}\lambda^{n+k}c_k + (n+k-1)_{l-1}\lambda^{n+k-1}c_{k-1} + \cdots + (n)_{l-1}\lambda^n c_0 = 0$
$\Rightarrow ((n)_{l-1}\lambda^n)_{n\geq 0}$ linear unabhängige Lösung von (1) ■

Voraussetzung:

> Seien $0 < \lambda_1 < \cdots < \lambda_r$ die Nullstellen von $\chi(x)$ mit Vielfachheiten $\mu_1, \ldots, \mu_r$
> und $(\lambda_i^n)_{n\geq 0}, (n\lambda_i^n)_{n\geq 0}, (n^2\lambda_i^n)_{n\geq 0}, \ldots, (n^{l-1}\lambda_i^n)_{n\geq 0}$, $i = 1, \ldots, r$ Lösungen von (1)

3.11 Satz. Die Allgemeine Lösung von (1) unter obigen Voraussetzung ist gegeben durch $P_{\mu_1-1}(n)\lambda_1^n + P_{\mu_2-1}(n)\lambda_2^n + \cdots + P_{\mu_r-1(n)}\lambda_r^n$, wobei $P_s(n)\ldots$ Polynom in n mit Grad $\leq s$.

3.12 Beispiel. $F_n = F_{n-1} + F_{n-2},\ n \geq 2 \quad F_0 = 0, F_1 = 1$ (Fibonacci-Folge)

$$\lambda^2 - \lambda - 1 = 0 \quad \lambda_{1,2} = \frac{1}{2} \pm \frac{\sqrt{5}}{2}$$
$$F_n = C_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + C_2 \left(\frac{1-\sqrt{5}}{2}\right)^n$$
$$F_0 = C_1 + C_2 = 0$$
$$F_1 = C_1 \frac{1+\sqrt{5}}{2} + C_2 \frac{1-\sqrt{5}}{2} = 1$$
$$C_1 \left(\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2}\right) = 1 \Rightarrow C_1 = \frac{1}{\sqrt{5}} \quad C_2 = -\frac{1}{\sqrt{5}}$$
Anfangswertproblem: $F_n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2}\right)^n$

$0, 1, 1, 2, 3, 5, 8, 13, \ldots$

3.13 Bemerkung. $\frac{1+\sqrt{5}}{2} \approx 1.618 \rightarrow$ Vergleiche Goldener Schnitt.
Inhomogene lineare Rekursion mit konstanten Koeffizienten:

$$\forall n \geq 0 : c_k a_{n+k} + c_{k-1} a_{n+k-1} + \cdots + c_1 a_{n+1} + c_0 a_n = S_n \tag{2}$$
$$\forall n \geq 0 : c_k a_{n+k} + c_{k-1} a_{n+k-1} + \cdots + c_1 a_{n+1} + c_0 a_n = 0$$

S_n wird **Störfunktion** genannt.

3.14 Satz.

1. Sei $(a_n^{(h)})_{n\geq 0}$ eine Lösung von (1) und $(a_n^{(p)})_{n\geq 0}$ eine Lösung von (2). Dann ist $(a_n^{(h)} + a_n^{(p)})_{n\geq 0}$ eine Lösung von (2).

2. Seien $(a_n)_{n\geq 0}$ und $(a_n^{(p)})_{n\geq 0}$ Lösungen von (2). Dann gibt es eine Lösung $(a_n^{(h)})_{n\geq 0}$ von (1) mit $a_n = a_n^{(h)} + a_n^{(p)}$.

Beweis:

$$\begin{array}{rll} a_n: & c_k a_{n+k} + \cdots + c_0 a_n & = S_n \\ a_n^{(p)}: & c_k a_{n+k}^{(p)} + \cdots + c_0 a_n^{(p)} & = S_n \mid \cdot(-1) \\ \hline & c_k(a_{n+k} - a_{n+k}^{(p)}) + \cdots + c_0(a_n - a_n^{(p)}) & = 0 \end{array}$$

$\Rightarrow a_n - a_n^{(p)}$ löst (1) ■

Spezielle Störfunktionen: $S_n = p_j(n)\alpha^n$ mit $p_j(n)\ldots$ Polynom vom Grad j
$\Rightarrow$ Ansatz: $a_n^{(p)} = q_j(n)\alpha^n n^{\nu(\alpha)}$, wobei
$q_j(n)\ldots$ Polynom vom Grad $\leq j$, $\nu(\alpha)$ = Vielfachheit von α als Nullstelle von $\chi(x)$

3.15 Beispiel.

$a_n - 4a_{n-1} + 4a_{n-2} = n2^n$
$(\lambda - 2)^2 = 0 \Rightarrow a_n^{(h)} = C_1 2^n + C_2 n 2^n$
$\alpha = 2, \nu(\alpha) = 2 \Rightarrow a_n^{(p)} = (An + B)n^2 2^n$

$(An + B)n^2 2^n - 4(A(n-1) + B)(n-1)^2 2^{n-1} + 4(A(n-2) + B)(n-2)^2 2^{n-2} = n2^n$
$(An + B)n^2 - 2(A(n-1) + B)(n-1)^2 + (A(n-2) + B)(n-2)^2 = n$
$n^3(A - 2A + A) + n^2(B + 4A + 2A - 2B - 4A - 2A + B) +$
$n(-2A - 4A + 4B + 4A + 8A - 4B) + 2A - 2B + 4B - 8A = n$
$n(6A) + (-6A + 2B) = n$
$\Rightarrow A = \frac{1}{6} \quad B = \frac{1}{2}$
$\Rightarrow a_n = C_1 2^n + C_2 n 2^n + n^2\left(\frac{1}{6}n + \frac{1}{2}\right)2^n$

Lineare Rekursionen 1. Ordnung:
Homogene lineare Rekursionen 1. Ordnung:

$$a_{n+1} = \alpha_n a_n, \ n \geq 0 \tag{3}$$

$$a_{n+1} = \alpha_n a_n = \alpha_n \alpha_{n-1} a_{n-1} = \ldots$$

Die Lösung ist gegeben durch $a_n = C\prod_{j=0}^{n-1}\alpha_j$, $a_0 = C$.

Inhomogene lineare Rekursionen 1. Ordnung

$$a_{n+1} = \alpha_n a_n + \beta_n,\ n \geq 0 \tag{4}$$

Variation der Konstanten: Ansatz: $a_n^{(p)} = C_n\prod_{j=0}^{n-1}\alpha_j$

$$C_{n+1}\prod_{j=0}^{n}\alpha_j = \alpha_n C_n\prod_{j=0}^{n-1}\alpha_j + \beta_n$$

$$C_{n+1} = C_n + \frac{\beta_n}{\prod_{j=0}^{n}\alpha_j} \Rightarrow C_n = \sum_{l=0}^{n-1}\frac{\beta_l}{\prod_{j=0}^{l}\alpha_j} + C_0 \quad (\text{wähle } C_0 = 0)$$

$$\Rightarrow \text{allgemeine Lösung: } a_n = \sum_{l=0}^{n-1}\frac{\beta_l}{\prod_{j=0}^{l}\alpha_j}\prod_{j=0}^{n-1}\alpha_j + C\prod_{j=0}^{n-1}\alpha_j$$

3.3 Kombinatorische Grundprobleme

Grundlegende Abzählaufgaben:
Seien A, B endliche Mengen.

Summenregel:

$$A \cap B = \emptyset \Rightarrow |A \cup B| = |A| + |B|$$

Produktregel:

$$|A \times B| = |A| \cdot |B|$$

Gleichheitsregel:

$$\exists f : A \to B \text{ bijektiv} \Leftrightarrow |A| = |B|$$

Anordnungs- und Auswahlprobleme:
Sei $A = \{a_1, \ldots, a_n\}$.

- **Anordnungen ohne Einschränkung (Variation mit Wiederholung [zur k-ten Klasse])**

$$(b_1, b_2, \ldots, b_k), \quad b_i \in A \quad \rightarrow \quad A^k \Rightarrow |A^k| = |A|^k = n^k$$

- **Anordnungen verschiedener Elemente (Variation ohne Wiederholung)**

$$(b_1, b_2, \ldots, b_k), \quad b_i \in A,\ i \neq j \Rightarrow b_i \neq b_j$$
$$n \cdot (n-1) \cdots (n-k+1) = \frac{n!}{(n-k)!} \quad \text{Möglichkeiten}$$

- **Permutationen einer Menge (von A)**:

  Zweizeilige Darstellung :

$$\begin{pmatrix} a_1 & a_2 & \cdots & a_n \\ \pi(a_1) & \pi(a_2) & \cdots & \pi(a_n) \end{pmatrix} \quad \pi : A \to A, \text{bijektiv}$$

  Wortdarstellung :

$$\pi(a_1)\pi(a_2)\cdots\pi(a_n)$$

  Zyklendarstellung :

$$(a_1, \pi(a_1), \pi(\pi(a_1)), \ldots)(a_{i_1}, \pi(a_{i_1}), \pi(\pi(a_{i_1})), \ldots) \cdots (a_{i_k}, \pi(a_{i_k}), \ldots)$$

  Anzahl $= n!$, (vgl. vorheriger Punkt mit $k = n$)

  3.16 Beispiel.

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 6 & 2 & 7 & 1 & 3 & 8 & 4 & 5 \end{pmatrix} = (1685374)(2)$$

- **Permutationen einer Multimenge**
  (Jedes Element darf auch öfter als einmal vorkommen)

$$\{a_1, a_1, \ldots, a_2, a_2, \ldots, a_i, \ldots, a_n, \ldots, a_n\}$$
$$a_i \text{ wird } k_i\text{-mal aufgenommen}$$

  Anzahl Permutationen:

$$\frac{(k_1 + k_2 + \cdots + k_n)!}{k_1! \cdot k_2! \cdots k_n!}$$

- **Auswahlen einer Teilmenge (Kombination ohne Wiederholung)**

$$\{b_1, b_2, \ldots, b_k\} \subseteq A$$
$$\text{Anzahl} = \frac{n!}{k!(n-k)!} = \binom{n}{k}$$

- **Auswahl einer (k-elementigen) Teil-Multimenge (Kombination mit Wiederholung)**

$\{b_1, \ldots, b_k\}, \quad b_i \neq b_j$ nicht vorausgesetzt
$\{b_1, \ldots, b_k\} (\not\subseteq A)$, $b_i \in A$

Wie oft nimmt man jedes a_i?

Anzahl Kugeln $= k$
Anzahl Trennstriche $= n - 1$
Anzahl der Kugel-Trennstrich-Wörter = Anzahl der Permutationen der Multimenge
$\{o, o, o, o, \ldots, o, |, |, \ldots, |\}$
k-mal Kugeln, $(n-1)$-mal Trennstriche

$$\Rightarrow \text{Anzahl } = \frac{(n-1+k)!}{k!(n-1)!} = \binom{n-1+k}{k} = \binom{n-1+k}{n-1}$$

Weitere Zählmethoden:

- **Doppeltes Zählen (Zählen auf zwei Arten)**

$A = \{a_1, \ldots, a_n\}$, $B = \{b_1, \ldots, b_n\}$
Sei $R \subseteq A \times B$ eine Relation.
$S_i = \{b \in B : a_i R b\}, \quad T_i = \{a \in A : a R b_i\}$

Behauptung: $|R| = |\{(a, b) : aRb\}| = \sum_{i=1}^{m} |S_i| = \sum_{j=1}^{n} |T_j|$.

Beweis:

Definiere die Matrix $(x_{ij})_{i,j=1}^{n}$ mit

$$x_{ij} = \begin{cases} 1 & \text{, für } a_i R b_j \\ 0 & \text{, sonst} \end{cases}$$

■

3.17 Beispiel. $\tau(n) :=$ mittlere Anzahl Teiler von x für $1 \leq x \leq n$
$n = 6$: A = B = $\{1, 2, \ldots, 6\}$

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

Spalten aufsummieren: $1 + 2 + 2 + 3 + 2 + 4 = 14$
$\tau(6) = \frac{14}{6} = \frac{7}{3}$

Allgemein:

$$\tau(n) = \frac{1}{n} \sum_{i=1}^{n} d(i)$$

$d(i) =$ Anzahl der Teiler von i

Beobachtung:

n Primzahl $\Rightarrow d(n) = 2$
$n = p^e \Rightarrow d(n) = e + 1$ (p Primzahl)
$n = \prod_{i=1}^{k} p_i^{e_i} \Rightarrow d(n) = \prod_{i=1}^{k} (e_i + 1)$
$l|n \Rightarrow l = \prod_{i=1}^{k} p_i^{f_i}, \quad f_i \leq e_i$

$A = B = \{1, 2, \ldots, n\}$
$T_i = \{a \in A : a|i\} \ldots$ Teiler von i
$S_j = \{b \in B : j|b\} \ldots$ Vielfache von j

$$\tau(n) = \frac{1}{n} \sum_{i=1}^{n} d(i) = \frac{1}{n} \sum_{i=1}^{n} |T_i| = \frac{1}{n} \sum_{j=1}^{n} |S_j| = \frac{1}{n} \sum_{j=1}^{n} \left\lfloor \frac{n}{j} \right\rfloor =$$

$$\frac{1}{n} \sum_{j=1}^{n} \frac{n}{j} + \mathcal{O}(1) = H_n + \mathcal{O}(1)$$

$\Rightarrow \tau(n) \sim \log(n) \quad H_n \sim \log(n) + \gamma \quad$ (γ ist Euler-Mascheroni-Konstante ≈ 0.57721)

- **Schubfachprinzip**:
  Falls $\exists f : A \to B, |A| > |B| \Rightarrow f$ nicht injektiv.

  Allgemeiner:
  Seien $A_1, A_2, \ldots, A_k$ endliche, paarweise disjunkte Mengen, $|A_1 \cup A_2 \cup \cdots \cup A_k| > kr$
  Dann existiert ein i mit $|A_i| > r$.

3.18 Beispiel. Behauptung: Es gibt mindestens 2 Menschen in Österreich, die am selben Tag in der gleichen Stunde geboren wurden.
Beweis: Schubfachprinzip

3.19 Beispiel. Behauptung: $\forall q \in \mathbb{N}$ ungerade $\exists i \in N : q|(2^i - 1) =: a_i$.

Beweis: Betrachte $a_1, a_2, \ldots, a_q \mod q$. Falls $\exists i : a_i \equiv 0 \mod q \Rightarrow$ fertig.

Andernfalls: $\exists i, j : a_i \equiv a_j \mod q$, o.B.d.A: $i < j$
$\Rightarrow a_i - a_j = qa$ mit $a \in \mathbb{Z}$ passend $\Rightarrow a_i - a_j = 2^i(1-2^{j-i}) \Rightarrow q|(2^{j-i}-1) = a_{j-i}$ ↯

- **Inklusions-Exklusions-Prinzip**
  A endliche Menge, $E_1, \ldots, E_m$ Eigenschaften.

  $$A_i = \{x \in A : \text{x hat die Eigenschaft } E_i\}$$

  Wie viele Elemente der Menge A besitzen keine Eigenschaften?

$$\left| A \setminus \bigcup_{i=1}^{m} A_i \right| = |A| + \sum_{\emptyset \neq I \subseteq \{1,2,\ldots,m\}} (-1)^{|I|} \left| \bigcap_{i \in I} A_i \right|$$

$$\left| \bigcup_{i=1}^{m} A_i \right| = \sum_{\emptyset \neq I \subseteq \{1,\ldots,m\}} (-1)^{|I|+1} \left| \bigcap_{i \in I} A_i \right|$$

3.20 Beispiel.

$|A \cup B| = |A| + |B| - |A \cap B|$
$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$

4 Divide & Conquer

4.1 Strassen-Algorithmus für Matrizenmultiplikation

Seien A, B quadratische Matrizen.

$$
\begin{aligned}
A &= (a_{ij})_{i,j=1}^{n} \\
B &= (b_{ij})_{i,j=1}^{n} \\
C &= AB = (c_{ij})_{i,j=1}^{n} \\
c_{ij} &= \sum_{k=1}^{n} a_{ik} b_{kj}
\end{aligned}
$$

Einfacher Algorithmus:
3 Schleifen, je n Iterationen $\Rightarrow$ Laufzeit $T(n) = \Theta(n^3)$

Algorithm 4 SQUARE-MATRIX-MILTIPLY(A, B)

```
1: n = A.zeilen
2: Sei C eine neue n × n-Matrix
3: for i = 1 to n do
4:     for j = 1 to n do
5:         c_ij = 0
6:         for k = 1 to n do
7:             c_ij = c_ij + a_ik b_kj
8:         end for
9:     end for
10: end for
11: return C
```

Einfaches Divide & Conquer Verfahren (Square-Matrix-Multiply-Recursive):

Algorithm 5 SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)

1: $n = A.zeilen$
2: Sei C eine neue $n \times n$-Matrix
3: **if** $n == 1$ **then**
4: $\quad C_{11} = A_{11}B_{11}$
5: **else**
6: $\quad$ partitioniere A, B und C gemäß Gleichung (5)
7: $\quad C_{11} = SQUARE\text{-}MATRIX\text{-}MULTIPLY\text{-}RECURSIVE(A_{11}, B_{11}) + SQUARE\text{-}MATRIX\text{-}MULTIPLY\text{-}RECURSIVE(A_{12}, B_{21})$
8: $\quad C_{12} = SQUARE\text{-}MATRIX\text{-}MULTIPLY\text{-}RECURSIVE(A_{11}, B_{12}) + SQUARE\text{-}MATRIX\text{-}MULTIPLY\text{-}RECURSIVE(A_{12}, B_{22})$
9: $\quad C_{21} = SQUARE - MATRIX - MULTIPLY - RECURSIVE(A_{21}, B_{11}) + SQUARE\text{-}MATRIX\text{-}MULTIPLY\text{-}RECURSIVE(A_{22}, B_{21})$
10: $\quad C_{22} = SQUARE\text{-}MATRIX\text{-}MULTIPLY\text{-}RECURSIVE(A_{21}, B_{12}) + SQUARE\text{-}MATRIX\text{-}MULTIPLY\text{-}RECURSIVE(A_{22}, B_{22})$
11: **end if**
12: **return** C

Sei $n = 2^k$.
Teile A und B in Viertel, A_{ij}, B_{ij}, C_{ij} sind $\frac{n}{2} \times \frac{n}{2}$ Matrizen.

$$C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} \quad i, j = 1, 2 \tag{5}$$

$$\Rightarrow T(n) \begin{cases} \Theta(1) & \text{, für } n = 1 \\ 8\Theta(\frac{n}{2}) + \Theta(1) + \Theta(n^2) & \text{, sonst} \end{cases}$$

$\Theta(1) \ldots$ Partitionieren (ohne kopieren)
$\Theta(n^2) \ldots$ Addition

Lösen für $n = 2^k$:

$$T(2^k) = \Theta(2^{2k}) + 8T(2^{k-1}) = \Theta(2^{2k}) + 8\Theta(2^{2k-2}) + 64T(2^{k-4}) = \ldots$$
$$\Rightarrow T(2^k) = \sum_{l=0}^{k} 8^l \Theta(2^{2(k-l)}) = \sum_{l=0}^{k} \Theta(2^{2k+l}) = \Theta\left(2^{2k}\frac{2^{k+1}-1}{2-1}\right) = \Theta(2^{3k}) = \Theta(n^3)$$

Dieser Algorithmus ist also nicht besser als das direkte Verfahren.

Strassen-Algorithmus:

Algorithm 6 STRASSEN-ALGORITHMUS(A, B)

1: $n = A.zeilen$
2: Sei C eine neue $n \times n$-Matrix
3: **if** $n == 1$ **then**
4: $\quad C_{11} = A_{11}B_{11}$
5: **else** partitioniere A, B und C gemäß Gleichung (5)
6: $\quad S_1 = A_{1,1} + A_{2,2}$
7: $\quad S_2 = B_{1,1} + B_{2,2}$
8: $\quad S_3 = A_{2,1} + A_{2,2}$
9: $\quad S_4 = B_{1,2} - B_{2,2}$
10: $\quad S_5 = B_{2,1} - B_{1,1}$
11: $\quad S_6 = A_{1,1} + A_{1,2}$
12: $\quad S_7 = A_{2,1} - A_{1,1}$
13: $\quad S_8 = B_{1,1} + B_{1,2}$
14: $\quad S_9 = A_{1,2} - A_{2,2}$
15: $\quad S_{10} = B_{2,1} + B_{2,2}$
16: $\quad P_1 = STRASSEN\text{-}ALGORITHMUS(S_1, S_2)$
17: $\quad P_2 = STRASSEN\text{-}ALGORITHMUS(S_3, B_{1,1})$
18: $\quad P_3 = STRASSEN\text{-}ALGORITHMUS(A_{1,1}, S_4)$
19: $\quad P_4 = STRASSEN\text{-}ALGORITHMUS(A_{2,2}, S_5)$
20: $\quad P_5 = STRASSEN\text{-}ALGORITHMUS(S_6, B_{2,2})$
21: $\quad P_6 = STRASSEN\text{-}ALGORITHMUS(S_7, S_8)$
22: $\quad P_7 = STRASSEN\text{-}ALGORITHMUS(S_9, S_{10})$
23: $\quad C_{11} = P_1 + P_4 - P_5 + P_7$
24: $\quad C_{12} = P_3 + P_5$
25: $\quad C_{21} = P_2 + P_4$
26: $\quad C_{22} = P_1 - P_2 + P_3 + P_6$
27: **end if**
28: **return** C

Korrektheit: Nachrechnen

Aufwand:

Matrizen A, B, C werden geviertelt : $\Theta(1)$
$S_1, \ldots, S_{10} : \Theta(n^2)$
$P_1, \ldots, P_7 : 7T\left(\frac{n}{2}\right)$
$C_{ij} : \Theta(n^2)$
$\Rightarrow T(n) = \begin{cases} \Theta(1) & , n = 1 \\ 7T\left(\frac{n}{2}\right) + \Theta(n^2) & , \text{sonst} \end{cases}$
Sei $n = 2^k$:

$$\Rightarrow T\left(2^k\right) = \sum_{l=0}^{k} 7^l \Theta\left(\frac{2^k}{2^l}\right)^2 = \sum_{l=0}^{k} \Theta\left(2^{2k+l(\log_2 7-2)}\right) = \Theta\left(2^{2k} \sum_{l=0}^{k} 2^{(\log_2 7-2)l}\right) = \Theta\left(2^{2k+(\log_2 7-2)k}\right)$$
$$= \Theta\left(n^{\log_2 7}\right), \text{ wobei } \log_2 7 \approx 2.807$$

Damit ist der Strassen-Algorithmus asymptotisch besser als das direkte Verfahren, auch wenn die Konstanten größer sind. In der Praxis wird der Strassen-Algorithmus für Matrizen großer Dimension verwendet und sobald die Dimension klein genug ist, wird auf das direkte Verfahren zurückgegriffen.

4.2 Substitutionsmethode

Guess & Proof
Idee: Richtige Form erraten und dann beweisen.

4.1 Beispiel. $T(n) = T(n-1) + n, \quad n \geq 1, \quad T(0) = 0, T(1) = 1$
Raten: $T(n) = \Theta(n^2) \Rightarrow$ zu zeigen: $T(n) \leq c_1 n^2 \wedge T(n) \geq c_2 n^2$

$$T(n) \leq c_1(n-1)^2 + n = c_1 n^2 - (2c_1 - 1)n + c_1 = n^2 - n + 1 \leq n^2 = c_1 n^2$$
$$\Rightarrow c_1 = 1$$
$$T(n) \geq c_2(n-1)^2 + n = c_2 n^2 - n(2c_2 - 1) + c_2 = \frac{n^2}{4} + \frac{n}{2} + \frac{1}{4} \geq \frac{n^2}{4} = c_2 n^2$$
$$\Rightarrow c_2 = \frac{1}{4}$$

Rate exakte Form: $T(n) = an^2 + bn + c$
Rechne: $T(n) = \frac{n^2}{2} + \frac{n}{2}$

4.2 Beispiel. $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 1$
Vermutung: $T(n) = \mathcal{O}(n)$, d.h. $T(n) \leq cn$

$$T(n) \leq c \left\lfloor \frac{n}{2} \right\rfloor + c \left\lceil \frac{n}{2} \right\rceil + 1 = cn + 1 > cn \text{ (\textbf{Achtung!})}$$
$T(n) = \Theta(n^2)? \Rightarrow$ Korrekt, aber zu pessimistisch.
Ansatz: $T(n) \leq cn - d$
$$\Rightarrow T(n) \leq c \left\lfloor \frac{n}{2} \right\rfloor - d + c \left\lceil \frac{n}{2} \right\rceil - d + 1 = cn - 2d + 1 \leq cn - d \Leftrightarrow d \geq 1$$

4.3 Bemerkung. Um zu zeigen, dass $T(n) = \mathcal{O}(n^k)$, kann der Ansatz von oben verallgemeinert werden:
Ansatz: $T(n) \leq a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$, wobei die Konstanten $a_k, \ldots, a_0$ zu bestimmen sind.
Hat man diese gefunden erhält man noch immer $T(n) = \mathcal{O}(n^k)$.
Weiters ist es nicht notwendig die Aussage $\forall n \in \mathbb{N}$ zu zeigen. Es reicht, wenn es ein n_0 gibt, sodass die Aussage $\forall n \geq n_0$ gilt.

4.3 Rekursionsbaummethode

Verwendung: Rekursionsbaummethode $\rightarrow$ Vermutung $\rightarrow$ Substitutionsmethode

4.4 Beispiel. $T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + \Theta(n^3), \quad T(1) = \Theta(1)$

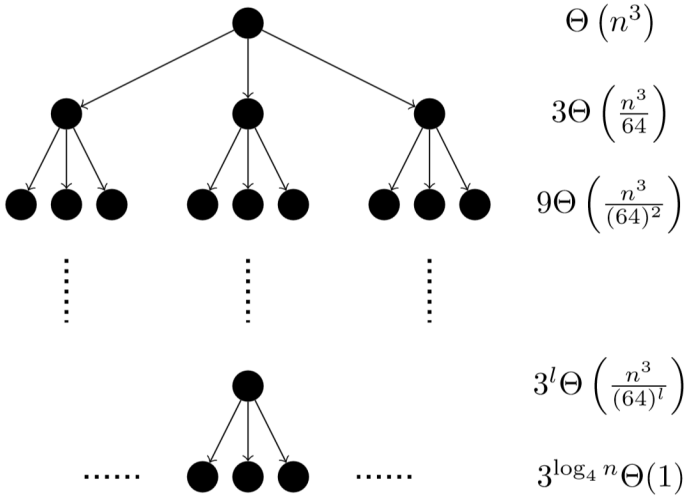


Abbildung 1: Rekursionsbaum

Sei $n = 4^k$

$$\tilde{\mathrm{T}}(n) = 3\tilde{\mathrm{T}}\left(\frac{n}{4}\right) + cn^3, \quad \tilde{\mathrm{T}}(1) = \tilde{\mathrm{c}}$$

$$\tilde{\mathrm{T}}(n) = 3\left(3\tilde{\mathrm{T}}\left(\frac{n}{16}\right) + c\left(\frac{n}{4}\right)^3\right) + cn^3 = 9\tilde{\mathrm{T}}\left(\frac{n}{16}\right) + 3c\frac{n^3}{64} + cn^3 = \ldots$$

$$\Rightarrow \tilde{\mathrm{T}}(n) \leq cn^3 \sum_{l\geq 0}\left(\frac{3}{64}\right)^l + \Theta\left(n^{\log_4 3}\right) = \mathcal{O}\left(n^3\right)$$

$$\Rightarrow \tilde{\mathrm{T}}(n) = \mathcal{O}(n^3)$$

Vermutung: $T(n) = \Theta(n^3)$
Beweis:

$$T(n) \leq 3c\left\lfloor\frac{n}{4}\right\rfloor^3 + n^3 \leq 3c\frac{n^3}{64} + n^3 \overset{!}{\leq} cn^3 \Leftrightarrow n^3 \leq c\frac{61}{64}n^3 \Rightarrow c \geq \frac{64}{61} \Rightarrow T(n) = \mathcal{O}\left(n^3\right)$$

$$T(n) = 3T\left(\left\lfloor\frac{n}{4}\right\rfloor\right) + \Theta\left(n^3\right) \Rightarrow T(n) = \Omega\left(n^3\right)$$

4.4 Mastermethode (Mastertheorem)

Formulierung:

4.5 Satz. Seien $a \geq 1, b > 1, f : \mathbb{R}_0^+ \to \mathbb{R}_0^+, T : \mathbb{N} \to \mathbb{R}, \epsilon > 0$,
$T(n) = aT(\frac{n}{b}) + f(n)$ (statt $\frac{n}{b}$ geht auch $\lfloor \frac{n}{b} \rfloor, \lceil \frac{n}{b} \rceil$). Dann gilt:

1. $f(n) = \mathcal{O}(n^{\log_b a - \epsilon}) \Rightarrow T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$
3. $f(n) = \Omega(n^{\log_b a + \epsilon}) \wedge \exists c < 1 : \forall n \geq n_0 : f(\frac{n}{b}) \leq \frac{c}{a} f(n) \Rightarrow T(n) = \Theta(f(n))$

4.6 Beispiel.

1. $T(n) = 9T(\frac{n}{3}) + n, \quad a = 9, b = 3 \quad \log_3 9 = 2, n = \mathcal{O}(n^{2-\epsilon}) \Rightarrow T(n) = \Theta(n^2)$
2. $T(n) = T(\frac{2n}{3}) + 1 \Rightarrow \log_{\frac{3}{2}} 1 = 0, f(n) = \Theta(1) \Rightarrow T(n) = \Theta(\log n)$
3. $T(n) = 2T(\frac{n}{2}) + n \log n, \log_2 2 = 1, f(n) = n \log n \neq \Theta(n), \neq \Omega(n^{1+\epsilon})$
   $\Rightarrow$ Master-Theorem in dieser Form nicht anwendbar

Vorbereitung:

4.7 Lemma. Sei zunächst $n = b^k$, $T(n) = \begin{cases} \Theta(1) & , \text{ für } n = 1 \\ aT\left(\frac{n}{b}\right) + f(n) & , \text{ für } n = b^k, k > 0 \end{cases}$

$$\Rightarrow T(n) = \Theta\left(n^{\log_b a}\right) + \sum_{l=0}^{\log_b n - 1} a^l f\left(\frac{n}{b^l}\right)$$

Beweis:
Rekursionsbaummethode. ■

4.8 Lemma. Sei $a \geq 1$, $b > 1$, $n = b^k$, $f \geq 0$, $g(n) = \sum_{l=0}^{k-1} a^l f\left(\frac{n}{b^l}\right)$. Dann gilt:

1. $f(n) = \mathcal{O}\left(n^{\log_b a - \epsilon}\right) \Rightarrow g(n) = \mathcal{O}\left(n^{\log_b a}\right)$
2. $f(n) = \Theta\left(n^{\log_b a}\right) \Rightarrow g(n) = \Theta\left(n^{\log_b a} \log n\right)$
3. $\exists c < 1 : \forall n \geq n_0 : f(\frac{n}{b}) \leq \frac{c}{a} f(n), f(n) = \Omega(n^{\log_b a}) \Rightarrow g(n) = \Theta(f(n))$

Beweis:

1.

$$g(n) = \Theta\left(\sum_{l=0}^{k-1} a^l \left(\frac{n}{b^l}\right)^{\log_b a-\epsilon}\right)$$

$$\sum_{l=0}^{k-1} a^l \left(\frac{n}{b^l}\right)^{\log_b a-\epsilon} = n^{\log_b a-\epsilon} \sum_{l=0}^{k-1} \left(\frac{a}{b^{\log_b a-\epsilon}}\right)^l = n^{\log_b a-\epsilon} \sum_{l=0}^{k-1} (b^\epsilon)^l = n^{\log_b a-\epsilon} \frac{b^{\epsilon \log_b n}-1}{b^\epsilon - 1}$$

$$\Rightarrow g(n) = \mathcal{O}\left(n^{\log_b a}\right)$$

2.

$$g(n) = \Theta\left(\sum_{l=0}^{k-1} a^l \left(\frac{n}{b^l}\right)^{\log_b a}\right) = \Theta\left(\sum_{l=0}^{k-1} n^{\log_b a}\right) = \Theta\left(n^{\log_b a} \log n\right)$$

da $k = \log_b n = \Theta(\log n)$

3.

$$g(n) = f(n) + \sum_{l=1}^{k-1} \cdots = \Omega(f(n))$$

$f\left(\frac{n}{b^j}\right) \leq \frac{c}{a} f\left(\frac{n}{b^{j-1}}\right) \leq \cdots \leq \left(\frac{c}{a}\right)^j f(n)$ für $\frac{n}{b^j}$ hinreichend groß, d.h. $\geq n_0$

für $\frac{n}{b^l} \leq n_0$ lässt sich $f(\frac{n}{b^l}) \leq C = \mathcal{O}(1)$ mit der Konstante C abschätzen.

Ab dem Index $k - L + 1$ ist $\frac{n}{b^{k-L+i}} = \frac{n}{nb^{-L+i}} = b^{L-i} \leq n_0$

$$\Rightarrow g(n) = \sum_{l=0}^{k-L} a^l f\left(\frac{n}{b^l}\right) + \sum_{l=k-L+1}^{k-1} a^l f\left(\frac{n}{b^l}\right) \leq \sum_{l=0}^{k-L} a^l f\left(\frac{n}{b^l}\right) + C \sum_{l=k-L+1}^{k-1} a^l$$

$$\leq \sum_{l=0}^{k-L} c^l f(n) + \mathcal{O}\left(a^{\log_b n}\right) \leq$$

$$\leq \sum_{l\geq 0} c^l f(n) + \mathcal{O}\left(a^{\log_b n}\right) = \frac{f(n)}{1-c} + \mathcal{O}\left(n^{\log_b a}\right) = \Theta(f(n))$$

■

4.9 Bemerkung. $f(n) \geq \frac{a}{c} f\left(\frac{n}{b}\right) > af\left(\frac{n}{b}\right) \Rightarrow f(n) = \Omega\left(n^{\log_b(\frac{a}{c})}\right)$ mit $\frac{a}{c} = a + \epsilon$

4.10 Bemerkung.

1. **Strassen Algorithmus** (1969):
   $\mathcal{O}(n^{2.81}) : \mathcal{O}(n^{\omega+\epsilon}) \quad \forall \epsilon > 0$

2. **Coppersmith & Winograd** (1987):
   $\omega < 2.375477$
   $2 \leq \rho \leq 3, V_\rho(t)$ Bewertung der Triliniarform t
   $V_\rho(t^{\otimes m})^{\frac{1}{m}} \geq R(t) \Rightarrow \omega \leq \rho$ mit $t^{\otimes m} \ldots$ m-faches Tensorprodukt mit sich selbst und $R(t) \ldots$ Rang vom Tensor.
   Problem: Finde richtige Trilinearform und m. Für $m = 2$: Numerisch gelöst

3. **Stathers** (2010):
   $m = 4 : \omega < 2.3736898$

4. **Williams** (2012):
   Bessere Numerik: $\Rightarrow \omega < 2.3729$

5. **Le Gall** (2014):
   a) $m = 8 \Rightarrow \omega < 2.3728642$
   b) $m = 16 \Rightarrow \omega < 2.3728640$
   c) $m = 32 \Rightarrow \omega < 2.3728639$

6. Vermutung: $\omega \to 2$ für $m \to \infty$

4.11 Lemma. Sei $a \geq 1, b > 1, f \geq 0, n = b^k, T(n) = \begin{cases} \Theta(1) & , \text{ für } n = 1 \\ aT\left(\frac{n}{b}\right) + f(n) & , \text{ für } n = b^k, k > 0 \end{cases}$
Dann gilt:

1. $f(n) = \mathcal{O}(n^{\log_b a - \epsilon}) \Rightarrow T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$
3. $\exists c < 1 : \forall n \geq n_0 : f(\frac{n}{b}) \leq \frac{c}{a} f(n), f(n) = \Omega(n^{\log_b a + \epsilon}) \Rightarrow T(n) = \Theta(f(n))$

Beweis:
$\overset{\text{Lemma 4.7}}{\Rightarrow} \Theta(n^{\log_b a}) + g(n)$ ■

Beweis des Mastertheorems:
Nun können wir das Master-Theorem allgemein beweisen:
Beweis:

$$T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + f(n) \tag{6}$$
$$T(n) = aT\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + f(n) \tag{7}$$

Es gilt $\frac{n}{b} - 1 < \left\lfloor \frac{n}{b} \right\rfloor \leq \frac{n}{b} \leq \left\lceil \frac{n}{b} \right\rceil < \frac{n}{b} + 1$.
Ziel: Zuerst eine obere Schranke für (6) und damit eine obere Schranke für (7) finden.
Dann analog eine untere Schranke für (7) und damit eine untere Schranke für (6) finden.

$$n_0 := n, n_1 := \left\lceil \frac{n}{b} \right\rceil, n_2 := \left\lceil \frac{\left\lceil \frac{n}{b} \right\rceil}{b} \right\rceil \quad n_j \ldots \text{ Argument der } j\text{-ten Iteration}$$
$$n_j = \begin{cases} n & , \text{ für } j = 0 \\ \left\lceil \frac{n_{j-1}}{b} \right\rceil & , \text{ für } j > 0 \end{cases}$$
$$\lceil x \rceil \leq x + 1$$
$$\Rightarrow n_1 \leq \frac{n}{b} + 1, n_2 \leq \frac{n_1}{b} + 1 \leq \frac{\frac{n}{b} + 1}{b} + 1 = \frac{n}{b^2} + \frac{1}{b} + 1$$
$$\Rightarrow \text{Induktion: } n_l \leq \frac{n}{b^l} + \sum_{i=0}^{l-1} \frac{1}{b^i} < \frac{n}{b^l} + \frac{b}{b-1} \tag{8}$$
$$l = \lfloor \log_b n \rfloor \Rightarrow n_l < \frac{n}{b^{\log_b n - 1}} + \frac{b}{b-1} = b + \frac{b}{b-1} = \mathcal{O}(1)$$

Ab dieser Tiefe ist die Problemsgröße höchstens eine Konstante.

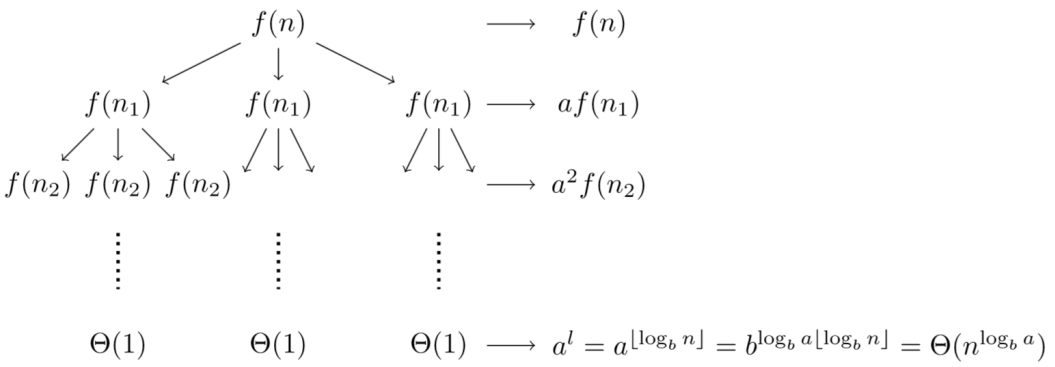


Abbildung 2: Rekursionsbaum für Mastertheorem

$$\Rightarrow T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{l-1} a^j f(n_j)$$

Fall 3): $af\left(\left\lceil \frac{n}{b} \right\rceil\right) \leq cf(n) \quad \forall n \geq n_0$

$\Rightarrow a^j f(n_j) \leq c^j f(n) \quad \forall n \geq n_0$

$\overset{\text{wie Lemma 4.8}}{\Rightarrow} T(n) = \Theta\left(n^{\log_b a}\right) + \Theta\left(f(n)\right)$

Fall 2): $j \leq \lfloor \log_b n \rfloor \Rightarrow \frac{b^j}{n} \leq 1$

$$f(n_j) \overset{f=\mathcal{O}(n^{\log_b a})}{\leq} cn_j^{\log_b a} \overset{(8)}{\leq} c\left(\frac{n}{b^j} + \frac{b}{b-1}\right)^{\log_b a}$$

$$= c\left(\frac{n}{b^j}\right)^{\log_b a}\left(1 + \frac{b}{b-1}\frac{b^j}{n}\right)^{\log_b a} \leq c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \frac{b}{b-1}\right)^{\log_b a} = \Theta\left(\frac{n^{\log_b a}}{a^j}\right)$$

Jeder Summand: $\mathcal{O}(n^{log_b a})$, Anzahl der Summanden: $\mathcal{O}(\log n)$

$\overset{\text{wie Lemma 4.8}}{\Rightarrow} \Theta\left(n^{\log_b a} \log n\right)$

Fall 1): $f(n) = \mathcal{O}\left(n^{\log_b a - \epsilon}\right)$

$$f(n_l) \leq cn_l^{\log_b a - \epsilon} \leq c\left(\frac{n}{b^l}\right)^{\log_b a - \epsilon}\left(1 + \frac{b}{b-1}\frac{b^l}{n}\right)^{\log_b a - \epsilon} \leq \mathcal{O}\left(\frac{n^{\log_b a - \epsilon} b^{\epsilon l}}{a^l}\right)$$

$\overset{\text{wie Lemma 4.8}}{\Rightarrow} \Theta\left(n^{\log_b a} \log n\right)$

$\Rightarrow$ obere Schranke für (6) $\Rightarrow$ obere Schranke für (7)
Mache analogen Beweis mit $x - 1 \leq \lfloor x \rfloor$ statt $\lceil x \rceil \leq x + 1$
$\Rightarrow$ untere Schranke für (7) $\Rightarrow$ untere Schranke für (6) ■

5 Probabilistische Analyse und Randomisierte Algorithmen

5.1 Bewerberprobleme

Seien $K_1, K_2, \ldots, K_n$ Bewerber. Gesucht ist der beste Bewerber.
Kosten für Interview, Headhunter und Kündigung

Strategie: Wenn K_i besser als $K_1, \ldots, K_{i-1}$, dann stelle K_i ein.

Algorithm 7 HIRE(n)

```
1: best := 0
2: for i = 1 to n do
3:     interviewe K_i
4:     if K_i > best then
5:         best := i
6:         stelle K_i ein
7:     end if
8: end for
```

Wie groß sind die Kosten?
$C_I \ldots$ Interviewkosten, $C_H \ldots$ Einstellungskosten

Gesucht: $C_I n + C_H m$ mit $m =$ Anzahl eingestellter K_i

Worst-Case: $m = n$

Average-Case: Sei $rg(K_1), \ldots, rg(K_n)$ zufällige Permutation $\pi \in S_n$
($rg(K_i) = 1 \Rightarrow K_i$ ist schlechtester Bewerber)

$\mathbb{E}$ (Anzahl eingestellter Kandidaten)
$X_n :=$ Anzahl eingestellter Kandidaten ist Zufallsvariable

$$X_{n,i} = \mathbb{1}_{[K_i \text{ eingestellt}]}, \quad X_n = \sum_{i=1}^{n} X_{n,i}$$
$$\mathbb{P}(X_{n,i} = 1) = \mathbb{E}X_{n,i} = \frac{1}{i}$$
$$\mathbb{E}X_n = \sum_{i=1}^{n} \mathbb{E}X_{n,i} = \sum_{i=1}^{n} \frac{1}{i} = \log n + \mathcal{O}(1)$$

5.2 Randomisierte Algorithmen

Problem: Sind die Eingabedaten zufällig?
Lösung: Randomisieren (Algorithmus hängt von Eingabe und einer Zufallszahl ab)

5.1 Beispiel. Bewerberproblem
Eingabe einer zufälligen Permutation $\pi \in S_n \to$ zufällige Permutation $\sigma \in S_n$
$\Rightarrow \mathbb{E}\text{Kosten} = \mathcal{O}(\log n)$

Permutieren durch Sortieren:
Permutieren durch Sortieren: $A[i]$ wird zufällige Priorität zugewiesen, dann wird A nach der Priorität sortiert.

Algorithm 8 PERM-BY-SORT(A)

```
1: n := |A|
2: Sei P[1...n] ein Array
3: for i = 1 to n do
4:     P[i] := RANDOM(1, n^3)
5: end for
6: SORT(A) (bzgl P)
```

5.2 Bemerkung.
$RANDOM(1, n^3)$ erzeugt eine Pseudo-Zufallszahl zwischen 1 und n^3. Damit gilt:
$\mathbb{P}(\forall i, j \text{ mit } i \neq j : P[i] \neq P[j]) = 1 - \frac{1}{n}$

Nachteil: Sortieren braucht $\Theta(n \log n)$ Schritte (bzw. $\Theta(n)$, falls nicht vergleichsbasiert)
In-Place-Permutation (Fisher-Yates):

Algorithm 9 RANDOMIZE-IN-PLACE(A)

```
1: n := |A|
2: for i = 1 to n do
3:     A[i] ↔ A[RANDOM(i, n)]
4: end for
```

Behauptung: $A[\pi(1)], \ldots, A[\pi(n)]$ ist zufällige Permutation von $A[1 \ldots n]$
Beweis:
Schleifeninvariante: Vor der i-ten Iteration der for-Schleife gilt: Sei $(b_1, \ldots, b_{i-1})$ eine Anordnung von $i-1$ Elementen aus $A[1 \ldots n]$

$$\Rightarrow \mathbb{P}(A[1 \ldots i-1] = (b_1, \ldots, b_{i-1})) = \frac{(n-i+1)!}{n!} \tag{9}$$

Initialisierung: $i = 1 : \mathbb{P}(A[] = ()) = 1$

Aufrechterhaltung: Vor i-ten Iteration gelte (9) für i. Nach der i-ten Iteration: Sei $(b_1, \ldots, b_i)$ eine Anordnung von i Elementen aus $A[1 \ldots n]$. Sei b_i mit $A[i] := b_i$

$E_1 = [\text{nach } i-1 \text{ Iterationen gilt } A[1 \ldots n] = (b_1, \ldots, b_{i-1})]$
$E_2 = [i\text{-te Iteration} : A[i] := b_i]$
$\mathbb{P}(E_1) = \frac{(n-i+1)!}{n!}$
$\mathbb{P}(A[1 \ldots i] = (b_1, \ldots b_i)) = \mathbb{P}(E_1 \cap E_2) = \mathbb{P}(E_2|E_1)\mathbb{P}(E_1)$
$= \frac{(n-i+1)!}{n!}\mathbb{P}(A[RANDOM(i,n)] = b_i) = \frac{(n-i+1)!}{n!}\mathbb{P}(RANDOM(i,n) = j)$
$= \frac{(n-i)!}{n!} = (9) \text{ von } i+1$

Terminierung: $i = n+1$
$\mathbb{P}(A[1 \ldots n] = (b_1, \ldots, b_n)) = \frac{1}{n!}$ ■

5.3 Geburtstagsparadoxon

Seien m Personen gegeben. $n = 365$.
Gesucht: $\mathbb{P}_m(2 \text{ Personen haben am gleichen Tag Geburtstag})$

$\Rightarrow A = [\text{alle Geburtstage verschieden}] \Rightarrow \mathbb{P}(A) = \frac{n-1}{n} \cdot \frac{n-2}{n} \cdots \frac{n-m+1}{n} = \frac{(n)_m}{n^m}$

Es gilt: $1 + x \leq e^x \quad \forall x \in \mathbb{R}$

$\frac{n-i}{n} = 1 - \frac{i}{n} \Rightarrow \mathbb{P}(A) \leq \exp(-\sum_{i=1}^{m-1} \frac{i}{n}) = \exp(-\frac{m(m-1)}{2n}) \overset{!}{<} \frac{1}{2}$

$\Leftrightarrow \frac{m(m-1)}{2n} \geq \log 2 \Leftrightarrow m \geq \frac{1}{2} + \sqrt{\frac{1}{4} + 2n \log n}$

5.3 Beispiel. Bei 23 Personen gilt:
$\mathbb{P}_{23}(\text{2 Personen haben am gleichen Tag Geburtstag}) = 50,7\%$

6 Heapsort

6.1 Heaps

6.1 Definition. Ein Heap ist eine Datenstruktur, die als fast vollständiger Binärbaum aufgefasst werden kann. Der Baum ist vollständig bis auf die unterste Ebene. Diese wird von links nach rechts gefüllt.

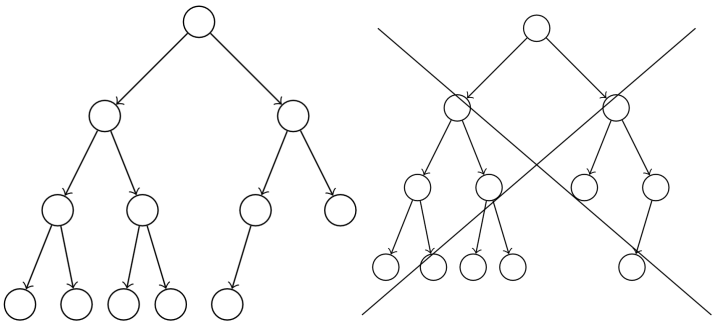


Abbildung 3: Heap

Heap $\hat{=}$ Feld $A, |A| \ldots$ Feldlänge (Feldgröße), $H(A) \ldots$ Anzahl der Elemente, die bereits gespeichert wurden.
$A[1]$ nennt man Wurzel. Die untersten Knoten werden auch Blätter genannt.
Für die Position $A[i]$ (auch Knoten genannt) gilt:

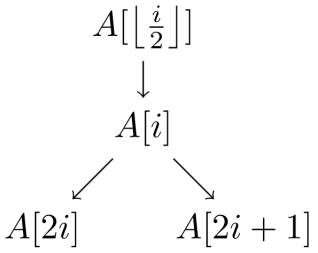


Abbildung 4: Heapknoten

Hilfsfunktionen:

$$PARENT(i) := \left\lfloor \frac{i}{2} \right\rfloor$$
$$LEFT(i) := 2i$$
$$RIGHT(i) := 2i + 1$$

Max-Heap: $A[\lfloor \frac{i}{2} \rfloor] \geq A[i]$
Min-Heap: $A[\lfloor \frac{i}{2} \rfloor] \leq A[i]$

6.2 Beispiel. (Max-Heap)
$A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$
Sei x ein Knoten. Definiere:
$h(x)$ = Höhe von x = Anzahl der Kanten des längsten Weges von x bis zu einem Blatt.
h(Wurzel) = Höhe des Heap $\Rightarrow h$(Wurzel) $= 3$

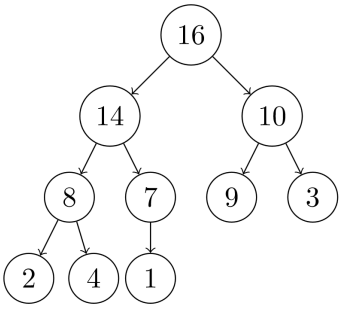


Abbildung 5: Max-Heap

6.2 Aufrechterhaltung der Heap-Eigenschaft

Vorausetzung: Binärbäume mit Wurzeln $LEFT(i)$ und $RIGHT(i)$ sind Max-Heaps, $A[i] <$ Kinder von $A[i]$ (d.h. Störung an der Stelle i)
Input: A, i
Output: Binärbaum mit Wurzel $A[i]$, der Max-Heap ist.

Algorithm 10 MAX-HEAPIFY(A, i)

```
1: l = LEFT(i)
2: r = RIGHT(i)
3: if l ≤ A.heapsize und A[l] > A[i] then
4:     maximum = l
5: else
6:     maximum = i
7: end if
8: if r ≤ A.heapsize und A[r] > A[maximum] then
9:     maximum = r
10: end if
11: if maximum ≠ i then
12:     vertausche A[i] mit A[maximum]
13:     MAX-HEAPIFY(A, maximum)
14: end if
```

6.3 Beispiel.

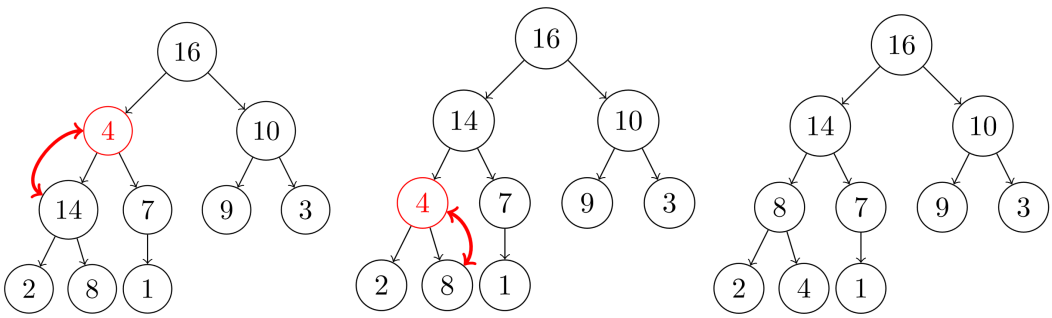


Abbildung 6: Beispiel für MAX-HEAPIFY-Algorithmus

Kosten: $\mathcal{O}$(Anzahl rekursiver Aufrufe) = $\mathcal{O}$(Baumhöhe) = $\mathcal{O}(\log n)$
$h(A[i]) = l \Rightarrow \mathcal{O}(l)$

6.3 Bauen eines Heaps

Max-Heapify bottom-up anwenden um $A[1 \ldots n]$ in Heap zu verwandeln.

Algorithm 11 BUILD-MAX-HEAP(A)

1: **for** $i = \left\lfloor \frac{|A|}{2} \right\rfloor$ downto 1 **do**
2: $\quad MAX\text{-}HEAPIFY(A, i)$
3: **end for**

Korrektheit: Schleifeninvariante: Vor jeder Iteration der for-Schleife ist jeder Knoten der Liste $i+1, i+2, \ldots, n$ Wurzel eines Max-Heaps.

Initialisierung: $\left\lfloor \frac{n}{2} \right\rfloor + 1, \left\lfloor \frac{n}{2} \right\rfloor + 2, \ldots, n$. Der Baum hat $\left\lfloor \frac{n}{2} \right\rfloor$ interne Knoten $\Rightarrow$ Die Knoten in der Liste sind genau die Blätter.

Aufrechterhaltung: (Kinder von i) $> i \Rightarrow$ Kinder sind Wurzeln eines Max-Heap. $MAX\text{-}HEAPIFY(A, i)$: Macht aus i die Wurzel eines Max-Heaps, $i := i - 1$

Terminierung: $i = 0$:
$\Rightarrow 1, 2, \ldots, n$ sind Wurzeln eines Max-Heaps, insbesondere ist 1 Wurzel $\Rightarrow$ Baum ist Max-Heap.

6.4 Bemerkung. Behauptung: Der Heap hat $\left\lfloor \frac{n}{2} \right\rfloor$ interne Knoten.
Sei der Baum vollständig, d.h. jeder Knoten hat entweder 2 oder 0 Kinder. Die Behauptung folgt damit durch Induktion über die ungerade Anzahl der Knoten:

Induktionsanfang: $n = 1$:

$$\Rightarrow 0 = \left\lfloor \frac{1}{2} \right\rfloor \text{ interne Knoten}$$

Induktionsschritt: $n \mapsto n + 2$:
Durch Hinzufügen von 2 Kindern, erhält man 2 weitere Knoten, welche nun außen sitzen, sowie einen zusätzlichen internen Knoten.

$$\Rightarrow \left\lfloor \frac{n}{2} \right\rfloor + 1 = \left\lfloor \frac{n+2}{2} \right\rfloor$$

Bei einem fast vollständigen Binärbaum, also dem Heap, kann zusätzlich der Fall eintreten, dass **genau ein** Knoten **nur ein** Kind hat, welches jedoch **außen** sitzt. Durch das Abrunden bleibt die Aussage weiterhin korrekt.

Kosten: $MAX\text{-}HEAPIFY$ $\frac{n}{2}$-mal aufgerufen, $\mathcal{O}(\log n) \to \mathcal{O}(n \log n)$

Besser:

1. Kosten von $MAX\text{-}HEAPIFY$: $\mathcal{O}(h)$ für Knoten der Höhe h.
2. Es gibt höchstens $\left\lfloor \frac{n}{2^{h+1}} \right\rfloor$ Knoten der Höhe h, Gesamthöhe $= \lfloor \log_2 n \rfloor$

$$\Rightarrow \text{ Kosten : } \sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lfloor \frac{n}{2^{h+1}} \right\rfloor \mathcal{O}(h) =$$
$$\mathcal{O}\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \right) \leq \mathcal{O}\left(n \sum_{l \geq 0} \frac{l}{2^l} \right) = \mathcal{O}(2n) = \mathcal{O}(n)$$

6.5 Bemerkung.

$$\sum_{n \geq 0} n \frac{x^n}{2^n} = \left(\frac{1}{1 - \frac{x}{2}} \right)' x = \frac{1}{1 - \frac{x}{2}} 2 \frac{x}{2}$$
$$\overset{x=1}{\Rightarrow} \sum_{l \geq 0} \frac{l}{2^l} = 2 = \sum_{l \geq 0} \frac{1}{2^l}$$

6.4 Heapsort

Gegeben: $A[1 \ldots n]$
Algorithmus:

1. $BUILD\text{-}MAX\text{-}HEAP \rightarrow A[1]$ ist größtes Element
2. $A[1] \leftrightarrow A[n]$
3. $A[n]$ entfernen $\Rightarrow LEFT(1), RIGHT(1)$ sind Wurzeln eines Max-Heaps, aber $A[1]$ eventuell keine Wurzel mehr
4. $MAX\text{-}HEAPIFY(A, 1) \rightarrow$ wieder Max-Heap erzeugen

Algorithm 12 HEAP-SORT(A)

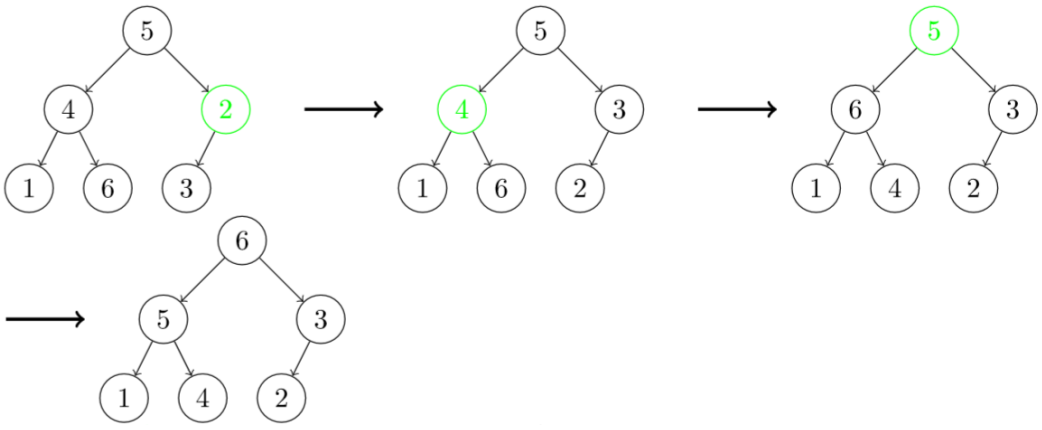
```
1: BUILD-MAX-HEAP(A)
2: for i = A.size downto 2 do
3:     vertausche A[1] mit A[i]
4:     A.heapsize = A.heapsize − 1
5:     MAX-HEAPIFY(A, 1)
6: end for
```

Kosten: $BUILD\text{-}MAX\text{-}HEAP \to \mathcal{O}(n)$
$MAX\text{-}HEAPIFY$: $n-1$ Aufrufe je $\mathcal{O}(\log n)$
$\Rightarrow \mathcal{O}(n \log n)$

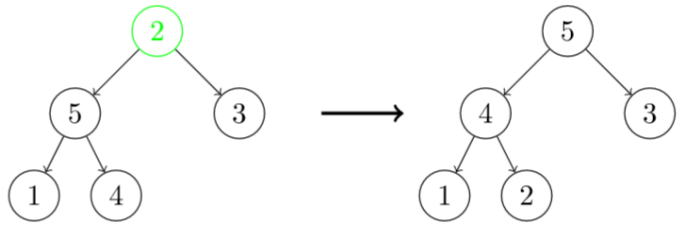
6.6 Bemerkung. **Genauer**: $\mathcal{O}(\log h)$, mit h Höhe des Knotens. Dennoch gilt:

$$
\begin{aligned}
&n! \sim \left(\frac{n}{e}\right)^n \sqrt{2\pi n} \\
&\Rightarrow \log(n!) \sim n \log n - n \log e + \mathcal{O}(n) \\
&\Rightarrow \log 2 + \log 3 + \cdots + \log n = \log(n!) = \Theta(n \log n)
\end{aligned}
$$

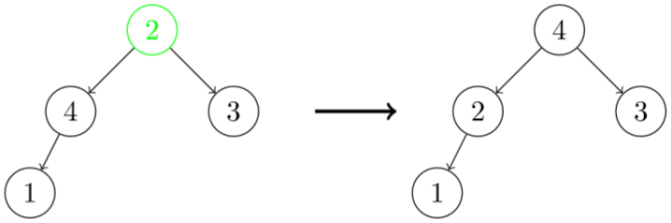
6.7 Beispiel. $A = [5,4,2,1,6,3]$, $\frac{|A|}{2} = 3$. Sortieren mit $HEAP\text{-}SORT$. $Max\text{-}Heapify$ wird oft angewendet:



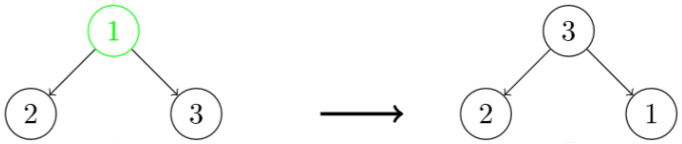
$\hat{=}\ A = [6,5,3,1,4,2] \to [2,5,3,1,4||6] \Rightarrow$



$\hat{=} A = [5, 4, 3, 1, 2||6] \to [2, 4, 3, 1||5, 6] \Rightarrow$



$\hat{=} A = [4, 2, 3, 1||5, 6] \to [1, 2, 3||4, 5, 6] \Rightarrow$



$\hat{=} A = [3, 2, 1||4, 5, 6] \to [1, 2||3, 4, 5, 6]$

6.5 Prioritätswarteschlangen

Datenstruktur zum Speichern einer Menge S. Jedes Element hat einen Schlüssel.
Max-PWS. . . größter Schlüssel $\hat{=}$ hohe Priorität

Operationen: Einfügen, Maximum von S, Entfernen des Maximums, Erhöhen eines Schlüssels

Algorithm 13 HEAP-MAXIMUM(A)

```
1: return A[1]
```


Algorithm 14 HEAP-EXTRACT-MAX(A)

```
1: if A.heapsize < 1 then
2:     error „Heap-Unterlauf“
3: end if
4: max = A[1]
5: A[1] = A[A.heapsize]
6: A.heapsize = A.heapsize − 1
7: MAX-HEAPIFY(A, 1)
8: return max
```

Algorithm 15 HEAP-INCREASE-KEY(A, i, key)

```
1: if key < A[i] then
2:     error „neuer Schlüssel kleiner als aktueller“
3: end if
4: A[i] = key
5: while i > 1 und A[PARENT(i)] < A[i] do
6:     vertausche A[i] mit A[PARENT(i)]
7:     i = PARENT(i)
8: end while
```

Algorithm 16 MAX-HEAP-INSERT(A, key)

```
1: A.heapsize = A.heapsize + 1
2: A[A.heapsize] = −∞
3: HEAP-INCREASE-KEY(A, A.heapsize, key)
```

Kosten:

1. Heap-Max: $\Theta(1)$
2. Extract: Schleifenrumpf wie bei Heapsort $\rightarrow \mathcal{O}(\log n)$ wegen $MAX\text{-}HEAPIFY$
3. Increase-Key: $\mathcal{O}(\log n)$, bzw. Pfadlänge: $h(Wurzel) - h(A[i])$
4. Insert: $\mathcal{O}(\log n)$

7 Quicksort

Quicksort wird in der Praxis sehr häufig eingesetzt, da er im Mittel der schnellste der vergleichsbasierten Sortieralgorithmen ist (hat kleine Konstanten beim asymptotischen Aufwand).

7.1 Der Algorithmus

In-place-Verfahren (braucht keinen zusätzlichen Speicher). Verwendet Divide & Conquer.

1. Divide: $A[p \dots r]$ in $A[p \dots q-1]$, $A[q]$ und $A[q+1 \dots r]$ zerlegen, wobei $\forall i < q : A[i] < A[q]$ und $\forall i > q : A[i] > A[q]$. Dazu muss q zuerst bestimmt werden.

2. Conquer: $A[p \dots q-1], A[q+1 \dots r]$ mit Quicksort sortieren.

7.1 Bemerkung. Kein Zusammenfügen mehr notwendig, da $A[q]$ bereits an der richtigen Position ist.

Algorithm 17 QUICKSORT(A, p, r)

```
1: if p < r then
2:     q := PARTITION(A, p, r)
3:     QUICKSORT(A, p, q − 1)
4:     QUICKSORT(A, q + 1, r)
5: end if
```

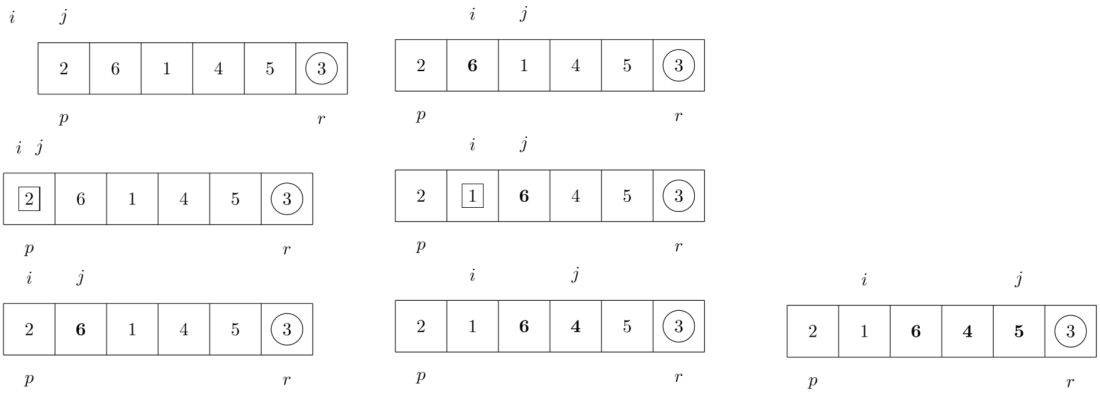
Algorithm 18 PARTITION(A, p, r)

```
1: x = A[r]
2: i = p − 1
3: for j = p to r − 1 do
4:     if A[j] ≤ x then
5:         i = i + 1
6:         vertausche A[i] mit A[j]
7:     end if
8: end for
9: vertausche A[i + 1] mit A[r]
10: return i + 1
```


Partitionierung:
$x = A[r]$ als Pivot-Element (franz. Angelpunkt) als Vergleichselement. Durch q, i, j, r enstehen vier Teilfelder.

Kann als Schleifeninvariante aufgefasst werden, womit man die Korrektheit von Quicksort zeigen kann.

7.2 Beispiel. Partitionierung für $[2, 6, 1, 4, 5, 3]$ mit 3 als Pivot-Element.



Laufzeit: $n - 1$ Vergleiche, $\mathcal{O}(n)$ Vertauschungen $\rightarrow \mathcal{O}(n)$

7.2 Analyse von Quicksort

Worst-Case: Partitionierung erzeugt Felder der Größe $n - 1$ und 0
$\Rightarrow T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n) \overset{\text{Substitutionsmethode}}{\Rightarrow} T(n) = \Theta(n^2)$.
Tritt auf wenn $A[1 \dots n]$ auf- oder absteigend sortiert ist.

Best-Case: $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \overset{Mastertheorem}{\Rightarrow} T(n) = \Theta(n \log n)$

Average-Case: Näher beim Best-Case $\Rightarrow T(n) = \Theta(n \log n)$.

7.3 Beispiel. 9 : 1-Teilung: $T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + cn \overset{Rekursionsbaum}{\Rightarrow} T(n) = \Theta(n \log n)$
99 : 1 bzw. cn, dn, mit $c + d = 1 \Rightarrow T(n) = \Theta(n \log n)$
Bei $o(n), n - o(n)$ nicht mehr. Dieser Fall ist aber sehr unwahrscheinlich.

7.3 Randomisiertes Quicksort

Oft teilsortierte Datensätze in der Praxis $\Rightarrow$ schlecht für Quicksort
$\Rightarrow$ Randomisieren durch:

1. Permutieren der Eingabe
2. Zufällige Wahl des Pivot-Elements

Algorithm 19 RANDOMIZED-QUICKSORT(A, p, r)

```
1: if p < r then
2:     q := RANDOMIZED-PARTITION(A, p, r)
3:     RANDOMIZED-QUICKSORT(A, p, q − 1)
4:     RANDOMIZED-QUICKSORT(A, q + 1, r)
5: end if
```

Algorithm 20 RANDOMIZED-PARTITION(A, p, r)

```
1: i := RANDOM(p, r)
2: A[i] ↔ A[r]
3: return PARTITION(A, p, r)
```

7.4 Analyse von Randomisiertes Quicksort

Worst-Case: $T(n) = \max\limits_{0 \le q \le n-1} (T(q) + T(n-q)) + \Theta(n) \overset{Substitutionsmethode}{\Rightarrow} T(n) = \Theta(n^2)$

Best-Case: $T(n) = \min\limits_{0 \le q \le n-1} (T(q) + T(n-q)) + \Theta(n) \overset{Master-Theorem}{\Rightarrow} T(n) = \Theta(n \log n)$

Average-Case: X = Anzahl der Schlüsselvergleiche (1 Vergleich in for-Schleife von $PARTITION$)

Anzahl der Aufrufe von $PARTITION \leq n$, da jedes Pivot-Element nur einmal auftreten kann. $\Rightarrow T(n) = \mathcal{O}(n + X)$
Beweis:
$PARTITION$ hat Laufzeit $\mathcal{O}(1)+\mathcal{O}$(Anzahl der Iterationen der for-Schleif) $\rightarrow \mathcal{O}(n)+$ $\mathcal{O}(X)$

$X \ldots$ Zufallsvariable
n Datensätze $z_1 < z_2 < \cdots < z_n, \quad Z_{i,j} := \{z_i, z_{i+1}, \ldots, z_j\}$
$X_{i,j} = \mathbb{1}_{[z_i \text{ wird mit } z_j \text{ verglichen}]}$
Vergleiche nur mit Pivot-Element $\Rightarrow$ Paar z_i, z_j wird höchstens einmal verglichen.

Pivot-Element $x : z_i < x < z_j \Rightarrow z_i$ und z_j in verschiedenen Teilfeldern.

$$\Rightarrow X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j}$$

$$\mathbb{E}X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}X_{i,j} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{P}(z_i \text{ wird mit } z_j \text{ verglichen})$$

$\mathbb{P}(z_i$ wird mit z_j verglichen$) = \mathbb{P}($in $Z_{i,j}$ wird z_i oder z_j als erstes als Pivot-Element gewählt werden$)$

$$= \frac{2}{j - i + 1}$$

$$\Rightarrow \mathbb{E}X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} = \sum_{k=1}^{n-1} \frac{2}{k+1} \sum_{i=1}^{n-k} 1 = (n+1)2(H_n - 1) - 2(n-1) =$$

$$= 2(n+1)H_n - 4n \sim 2n \log n + \mathcal{O}(n)$$

8 Sortieren in linearer Zeit

8.1 Untere Schranke für vergleichsbasierte Sortierverfahren

Annahme: Alle Elemente $a_1, a_2, \ldots, a_n$ sind paarweise verschieden.
Damit gilt:

1. Abfragen der Form $a_i = a_j$ unnötig
2. Abfragen der Form $a_i < a_j, a_i \leq a_j, a_i > a_j, a_i \geq a_j$ sind äquivalent

Entscheidungsbaum: Vollständiger Binärbaum

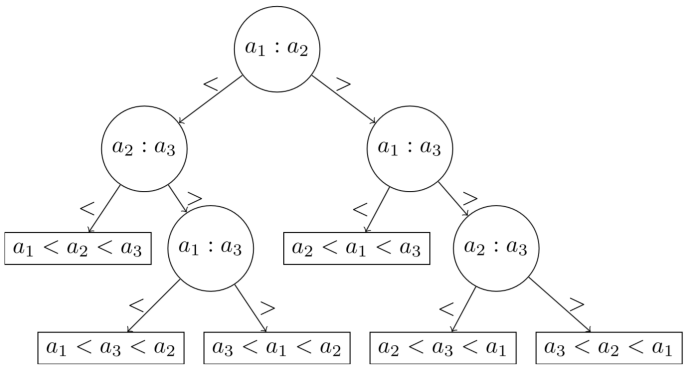


Abbildung 7: Entscheidungsbaum für Insertionsort

Jeder Algorithmus muss alle $n!$ Permutation erzeugen können $\Rightarrow n!$ Blätter.
Worst-Case $\hat{=}$ längsten Wurzel-Blatt-Pfad
Anzahl der Vergleiche im Worst-Case eines Algorithmus = Höhe des Entscheidungsbaum dieses Algorithmus.

8.1 Satz. Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche im Worst-Case.
Beweis:
Baum der Höhe $h \Rightarrow$ höchstens 2^h Blätter
$n! \leq$ Anzahl der Blätter $\leq 2^h \Rightarrow h \geq \log_2(n!) \overset{\text{Stirling'sche Formel}}{=} \Omega(n \log n)$ ■

8.2 Korollar. Heap- und Mergesort sind asymptotisch optimal.

8.2 Sortieren durch Zählen (Counting-Sort)

Annahme: $a_1, a_2, \ldots, a_n \in \{0, 1, \ldots, k\}$
Idee: Bestimme $\forall a_i : |\{a_j | a_j < a_i\}|$
Eingabe: $A[1 \ldots n]$
Ausgabe: $B[1 \ldots n]$
Hilfsfeld: $C[0 \ldots k]$
Counting-Sort bestimmt die Anzahl der Elemente zu einem Wert $\in \{0, 1, \ldots, k\}$ und legt damit die Position jedes Elements in der sortierten Liste fest.

Algorithm 21 COUNTING-SORT(A, B, k)

```
1: sei C[0...k] ein neues Feld
2: for i = 0 to k do
3:     C[i] = 0
4: end for
5: for j = 1 to A.size do
6:     C[A[j]] = C[A[j]] + 1
7: end for
8: // C[i] enthält nun die Anzahl der Elemente, die gleich i sind
9: for i = 1 to k do
10:    C[i] = C[i] + C[i − 1]
11: end for
12: // C[i] enthält nun die Anzahl der Elemente, die kleiner oder gleich i sind
13: for j = A.size downto 1 do
14:    B[C[A[j]]] = A[j]
15:    C[A[j]] = C[A[j]] − 1
16: end for
```

8.3 Definition. Ein Algorithmus heißt stabil, wenn gleiche Elemente ihre Reihenfolge nicht ändern.

8.4 Beispiel. Quicksort und Heapsort sind nicht stabil. Insertion-Sort und Mergesort sind stabil.
Counting-Sort in dieser Implementation ist stabil.

Kosten: 4 for-Schleifen: $\Theta(k), \Theta(n), \Theta(k), \Theta(n) \Rightarrow \Theta(k + n) = \Theta(n)$, falls $k = \mathcal{O}(n)$
Nachteil: Hoher Speicheraufwand, nur für Ganzzahlen geeignet

8.3 Radix-Sort

Sortieren von d-stelligen Zahlen. Hier wird zuerst die Liste nach der niedrigsten Stelle sortiert. Anschließend sortiert man die nächsthöheren Stellen usw. Durch die Stabilität von Counting-Sort bleibt die Sortierung der untersten Stellen dabei erhalten. Radix-Sort ist stabil.

Algorithm 22 RADIX-SORT(A, d)

```
1: for i = 0 to d do
2:     wende COUNTING-SORT an, um A nach Stelle i zu sortieren
3: end for
```

8.5 Bemerkung. Statt $COUNTING\text{-}SORT$ kann man ein beliebiges stabiles Sortierverfahren verwenden.

8.6 Lemma. Gegeben seien n d-stellige Zahlen, jede Stelle $\in \{0, 1, \ldots, k\} \Rightarrow$ Laufzeit von Radix-Sort $= \Theta(d(n + k))$ für korrekte Sortierung.

8.7 Korollar. Gegeben seien n b-Bit Zahlen, $r \in \mathbb{N}, r \leq b \Rightarrow$ Laufzeit $= \Theta\left(\frac{b}{r}\left(n + 2^r\right)\right)$

Beweis:
b Bit auffassen als d-stellige Zahl.
Stelle $\hat{=}$ r Bits, $d = \left\lfloor \frac{b}{r} \right\rfloor$, "Ziffern"$\in \{0, 1, \ldots, 2^r - 1\}$ ■

8.4 Bucket-Sort

Annahme: Eingabe: Zufallsvariablen $X_1, \ldots, X_n \in U[0, 1)$ (Gleichverteilung auf $[0, 1)$)
Idee: Teilen in n Buckets (Teilintervalle) $[0, \frac{1}{n}), [\frac{1}{n}, \frac{2}{n}), \ldots, [\frac{n-1}{n}, 1)$ und sortiere jedes Einzelne

Algorithm 23 BUCKET-SORT(A, B, k)

```
1: n = A.size
2: sei B[0...n-1] ein neues Feld
3: for i = 0 to n-1 do
4:     mache B[i] zu einer leeren Liste (B[i] = [])
5: end for
6: for i = 1 to n do
7:     füge A[i] in die Liste B[⌊nA[i]⌋] ein
8: end for
9: for i = 0 to n-1 do
10:    INSERTION-SORT(B[i])
11: end for
12: hänge die Listen B[0], B[1], ..., B[n-1] in dieser Reihenfolge hintereinander
```

Korrektheit: $A[i] \leq A[j]$. Zwei Fälle möglich:

1. Landen in verschiedene Buckets

2. Landen in gleichem Bucket $\rightarrow$ Insertion Sort

In beiden Fällen werden die Elemente korrekt sortiert.

8.8 Beispiel. $n = 100$, $a_1 = 0.577$, $a_2 = \frac{1}{e}$, $a_3 = \frac{1}{\sqrt{2}}, \ldots, a_{100} = \frac{1}{\pi}$.
$0.577 \rightarrow 57 \Rightarrow B[57] = [a_1]$

Kosten: N_i = Anzahl der Zahlen in $B[i]$ Zufallsvariable

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} \mathcal{O}\left(N_i^2\right)$$

$$\mathbb{E}T(n) = \Theta(n) + \sum_{i=0}^{n-1} \mathcal{O}\left(\mathbb{E}\left(N_i^2\right)\right)$$

Behautpung: $\mathbb{E}\left(N_i^2\right) = 2 - \frac{1}{n}$

Beweis: $X_{ij} := \mathbb{1}_{[A[j] \text{ fällt in } B[i]]}$

$$\Rightarrow N_i = \sum_{j=1}^{n} X_{ij} \Rightarrow \mathbb{E}\left(N_i^2\right) = \mathbb{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] = \mathbb{E}\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\sum_{\substack{1\le k\le n\\ k\neq j}} X_{ij}X_{ik}\right] =$$

$$\sum_{j=1}^{n}\mathbb{E}\left(X_{ij}^2\right) + \sum_{1\le j\le n}\sum_{\substack{1\le k\le n\\ k\neq j}} \mathbb{E}\left(X_{ij}X_{ik}\right) = 1 + n(n-1)\frac{1}{n^2} = 2 - \frac{1}{n} \Rightarrow \mathbb{E}T(n) = \Theta(n)$$

$$\text{da } X_{ij}^2 = X_{ij}, \quad \mathbb{P}(X_{ij} = 1) = \frac{1}{n}, \quad \mathbb{P}(X_{ij} = 1, X_{ik} = 1) = \frac{1}{n^2}$$

8.9 Bemerkung. Im Worst-Case landen alle Elemente in einem Bucket $\Rightarrow \Theta\left(n^2\right)$

9 Ordnungsstatistiken

9.1 Definition. i-te Ordnungsstatistik von $a_1, \ldots, a_n :=$ jenes $a_j : |\{a_l | a_l < a_j\}| = i-1$
$i = 1 \to$ Min $\quad i = n \to$ Max
$i = \lceil \frac{n+1}{2} \rceil$ bzw. $i = \lfloor \frac{n+1}{2} \rfloor$ nennt man oberen bzw. unteren Median.

Auswahlproblem:
Eingabe: Menge A von n paarweise verschiedenen Zahlen und $i \in \mathbb{Z} : 1 \leq i \leq n$.
Ausgabe: $x \in A$ mit $|\{y \in A | \, y < x\}| = i - 1$
Möglich durch Sortieren $\Rightarrow \Theta(n \log n)$

9.1 Min und Max

Algorithm 24 MIN(A)

```
1: Min := A[1]
2: for j = 2 to n do
3:     if A[j] < Min then
4:         Min := A[j]
5:     end if
6: end for
```

Algorithm 25 MAX(A)

```
1: Max := A[1]
2: for j = 2 to n do
3:     if A[j] > Max then
4:         Max := A[j]
5:     end if
6: end for
```

Anzahl der Vergleiche im besten Fall: $n - 1$

Idee: Minimum und Maximum „gleichzeitig bestimmen“.
Die Menge A habe eine gerade Anzahl an Elementen.

Algorithm 26 MINMAX(A)

```
1: n = A.size
2: Max := max(A[1], A[2])
3: Min := min(A[1], A[2])
4: for i = 2 to n/2 do
5:     Min = min(min(A[2i − 1], A[2i]), Min)
6:     Max = max(max(A[2i − 1], A[2i]), Max)
7: end for
```

Falls A eine ungerade Anzahl an Elemente hat, dann wird zuerst sowohl das Minimum, als auch das Maximum auf $A[1]$ gesetzt.

Laufzeit:

$$\Rightarrow \begin{cases} n \text{ ungerade}: & 3\frac{n-1}{2} = 3\left\lfloor \frac{n}{2} \right\rfloor \\ n \text{ gerade}: & 1 + 3\frac{n-2}{2} = \frac{3n}{2} - 2 \leq 3\left\lfloor \frac{n}{2} \right\rfloor \end{cases}$$

9.2 Bemerkung. Wenn man zuerst MIN und dann MAX verwendet, hat man $2n-2$ Vergleiche $\Rightarrow MINMAX$ um ca. 25 Prozent besser.

9.2 Auswahl in erwarteter linearer Zeit

Divide & Conquer (Quickselect, Hoare's Find-Algorithm)

$\rightarrow$ i-te Ordnungsstatistik von $A[p \dots r]$

Algorithm 27 RANDOMIZED-SELECT(A, p, r, i)

```
1: if p == r then
2:     return A[p]
3: end if
4: q = RANDOMIZED-PARTITION(A, p, r)
5: k = q − p + 1
6: if i == k then
7:     return A[q]
8: else if i < k then
9:     return RANDOMIZED-SELECT(A, p, q − 1, i)
10: else
11:     return RANDOMIZED-SELECT(A, q + 1, r, i − k)
12: end if
```


Analyse: $n = r - p + 1 \ldots$ Anzahl Elemente, alle paarweise verschieden.
$RANDOMIZED\text{-}PARTITION : \mathcal{O}(n)$
$\mathbb{P}(|A[p \ldots q]| = k) = \frac{1}{n}$ für $1 \leq k \leq n$
$B_k = [|A[p \ldots q]| = k]$, $X_k = \mathbb{1}_{B_k} \Rightarrow \mathbb{E}X_k = \mathbb{P}(B_k) = \frac{1}{n}$

Annahme: $T(n)$ monoton wachsend

$$\Rightarrow T(n) \leq \sum_{k=1}^{n} X_k T(\max(k-1, n-k)) + \mathcal{O}(n)$$
$$= \sum_{k=1}^{n} X_k (T(\max(k-1, n-k)) + \mathcal{O}(n))$$
$$\Rightarrow \mathbb{E}T(n) \leq \sum_{k=1}^{n} \mathbb{E}X_k \mathbb{E}T(\max(k-1, n-k)) + \mathcal{O}(n)$$

$$\max(k-1, n-k) = \begin{cases} k-1 & \text{, für } k > \left\lceil \frac{n}{2} \right\rceil \\ n-k & \text{, für } k \leq \left\lceil \frac{n}{2} \right\rceil \end{cases}$$

n gerade $\Rightarrow T\left(\left\lceil \frac{n}{2} \right\rceil\right), \ldots, T(n-1)$ kommen genau 2-mal in der Summe vor

n ungerade $\Rightarrow T\left(\left\lceil \frac{n}{2} \right\rceil\right), \ldots, T(n-1)$ kommen genau 2-mal in der Summe vor, $T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$ 1-mal

$$\Rightarrow \mathbb{E}T(n) \leq \frac{2}{n} \sum_{k=\left\lfloor \frac{n}{2} \right\rfloor}^{n-1} \mathbb{E}T(k) + \mathcal{O}(n)$$

Wir zeigen $\mathbb{E}T(n) = \mathcal{O}(n)$ mittels Substitutionsmethode, dass also ein $c > 0$ existiert mit $\mathbb{E}T(n) \leq cn$.
Annahmen:

1. $T(n) = \mathcal{O}(1)$ für $n \leq N$

2. $\mathcal{O}(n) \leq an$

$$\Rightarrow \mathbb{E}T(n) \leq \frac{2}{n} \sum_{k=\lfloor \frac{n}{2} \rfloor}^{n-1} ck + an = \frac{2c}{n} \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor - 1} \right) + an \leq$$

$$\frac{c}{n} \left(n(n-1) - \left(\frac{n}{2} - 1 \right) \left(\frac{n}{2} - 2 \right) \right) + an = c \left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n} \right) + an =$$

$$\frac{3cn}{4} + \frac{c}{2} + an = cn - \left(\frac{cn}{4} - \frac{c}{2} - an \right) \overset{!}{\leq} cn \Leftrightarrow$$

$$\frac{cn}{a} - \frac{c}{2} - an \geq 0 \Leftrightarrow \left(\frac{c}{4} - a \right) n \geq \frac{c}{2}$$

Wähle $c > 4a \Rightarrow n \geq \frac{\frac{c}{2}}{\frac{c}{4} - a} = \frac{2c}{c - 4a} =: N$

9.3 Auswählen in linearer Zeit (Worst-Case)

Idee: Modifiziere $PARTITION(A, x)$, $x \in \{A[1], A[2], \ldots, A[n]\}$, wobei x das Pivot-Element ist.

Algorithmus:

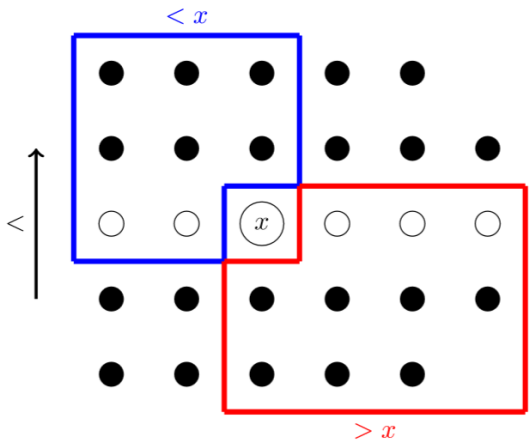
1. n Elemente in $\lfloor \frac{n}{5} \rfloor$ 5er-Gruppen und eine ($n \mod 5$)-Gruppe aufteilen

2. $SELECT$: Median jeder $\lceil \frac{n}{5} \rceil$ Gruppe mit Insertion-Sort bestimmen

3. Median x dieser $\lceil \frac{n}{5} \rceil$ Mediane mit $SELECT$ bestimmen

4. Zerlege das Eingabefeld um x mit $PARTITION(A, x) \rightarrow A[1 \ldots k-1]$, $A[k+1 \ldots n]$, $x = A[k]$

5. Wenn $i = k \rightarrow$ return x, sonst $\begin{cases} SELECT(A[1 \ldots k-1]) & \text{, für } i < k \\ SELECT(A[k-1 \ldots n]) & \text{, für } i > k \end{cases}$



Die Schritte 1,2 und 4 haben eine Laufzeit von $\mathcal{O}(n)$. Schritt 3 hat Kosten von $T\left(\left\lceil \frac{n}{5} \right\rceil\right)$.
Nun schauen wir uns die Kosten für Schritt 5 genauer an:
Mindestens $\frac{1}{2}\left\lceil \frac{n}{5} \right\rceil$ Gruppen mit 3 Elementen sind größer als x, außer der Gruppe von x und die $(n \mod 5)$-Gruppe.
Damit sind mindestens $3\left(\frac{1}{2}\left\lceil \frac{n}{5} \right\rceil - 2\right)$ Elemente größer als x.
$\Rightarrow$ mindestens $\frac{3n}{10} - 6$ Elemente sind größer als x und mindestens $\frac{3n}{10} - 6$ Elemente sind kleiner als x.
$\Rightarrow$ rekursive Aufrufe auf höchstens $\frac{7n}{10} + 6$ Elemente und damit hat der letzte Schritt eine Laufzeit von höchstens $T\left(\frac{7n}{10} + 6\right)$

$$T(n) \leq \begin{cases} \mathcal{O}(1) & \text{, für } n < N \\ T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7n}{10} + 6\right) + \mathcal{O}(n) & \text{, für } n \geq N \end{cases} \overset{N=140}{\Rightarrow} T(n) = \mathcal{O}(n)$$

10 Elementare Graphenalgorithmen

10.1 Darstellung von Graphen

10.1 Definition.

1. Ein Graph $G = (V, E)$ ist eine Menge von Knoten (Vertices) und Kanten (Edges), wobei $E \subseteq V \times V$, $e \in E, e = (u, v)$, bei einem gerichteten Graphen und $e = \{u, v\}$, $e = uv$ bei einem ungerichteten Graphen.

2. Ein Graph, der keine Schlingen und keine Mehrfachkanten hat, heißt schlicht.

3. $u, v \in V$ heißen adjazent (benachbart) $u \sim v :\Leftrightarrow uv \in E$.

4. $\Gamma(u) := \{w \in V | u \sim w\} \ldots$ Menge aller Nachbarn

5. $d(u) = |\Gamma(u)| \ldots$ Grad von u

Für einen gerichteten Graphen gilt weiters:

1. u heißt Nachfolger von $v :\Leftrightarrow (v, u) \in E$

2. u heißt Vorgänger von $v :\Leftrightarrow (u, v) \in E$

3. $\Gamma^+(v) \ldots$ Menge aller Nachfolger von v

4. $d^+(v) = |\Gamma^+(v)| \ldots$ Weggrad von v.

5. $\Gamma^-(v) \ldots$ Menge aller Vorgänger von v

6. $d^-(v) = |\Gamma^-(v)| \ldots$ Hingrad von v.

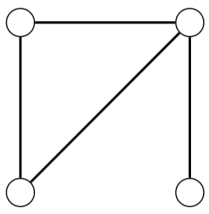


Abbildung 8: Beispiel für einen einfachen Graphen

Abbildung 9: Ein gerichteter bzw. ungerichteter Graph

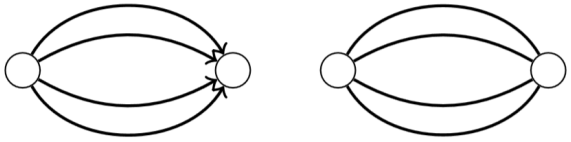


Abbildung 10: Graphen mit Mehrfachkanten

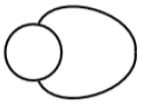


Abbildung 11: Graph mit Schlinge

10.2 Lemma. (Handschlaglemma)
$\sum_{v \in V} d(v) = 2|E|$.
Bei einem gerichtetem Graphen gilt: $\sum_{v \in V} d^+(v) = \sum_{v \in V} d^-(v) = |E|$

Es gilt: $|E| = \mathcal{O}(|V|^2)$.

10.3 Definition.

1. Ein Graph heißt dünn besetzt $:\Leftrightarrow |E| = o(|V|^2)$.
2. Ein Graph heißt dicht besetzt $:\Leftrightarrow |E| = \Omega(|V|^2)$.

Adjazenzliste (dünn besetzter Graph):
Feld von $|V|$ Listen $u \in V, Adj[u] = [v_{1,u}, v_{2,u}, \ldots, v_{k_u,u}] \Leftrightarrow$

1. $\Gamma(u) = \{v_{1,u}, \ldots, v_{k_u,u}\}$
2. $d(u) = k_u$

Bei einem gerichteten Graph ersetze $\Gamma(u)$ durch $\Gamma^+(u)$ oder $\Gamma^-(u)$
$\sum_{u \in V} |Adj[u]| = 2|E|$
Speicherbedarf $= \Theta(|V| + |E|)$

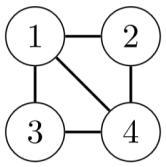
Adjazenzmatrix (dicht besetzter Graph):

$$V = \{v_1, \ldots, v_n\}$$
$$A = (a_{i,j})_{i,j=1}^{n} : a_{i,j} = \begin{cases} 1 & v_i \sim v_j \\ 0 & \text{, sonst} \end{cases}$$

Für einen ungerichteten Graphen gilt: $A = A^T$
Speicherbedarf $= \Theta\left(|V|^2\right)$

10.4 Bemerkung. In der obigen Matrixdefinition kann statt 1 auch eine Gewichtsfunktion $w(v_i, v_j)$ verwendet werden. Der Graph heißt dann gewichtet.

10.5 Beispiel. Speicherung mit Adjazenzmatrix:



$$A = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Nachteil: Mehr Speicherplatzbedarf
Vorteil: Direkter Zugriff auf Kanten

10.2 Breitensuche

Algorithmus zum Durchsuchen eines Graphen.

Eingabe: Graph $G = (V, E)$, Startknoten $s \in V$

Verfahren:

1. Knoten von G ausgehend von s erforschen

2. Distanz zu s berechnen

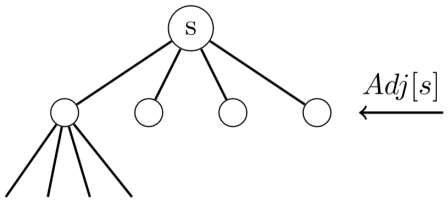
$$d(s,v) = \begin{cases} \min\limits_{\text{Wege } s \rightsquigarrow v} \text{Anzahl der Kanten des Weges } s \rightsquigarrow v & \text{, falls } \exists \text{ Weg} \\ \infty & \text{, sonst} \end{cases}$$

3. Erzeugen eines Breitensuchbaums (Breath-First-Search, kurz BFS) T. Die Wurzel von T ist s. Der Weg $s \rightsquigarrow v$ in $T \hat{=}$ kürzesten Weg $s \rightsquigarrow v$ in G.

Status eines Knotens: Attribut Farbe, $f(u), u \in V$

1. Weiß: Unentdeckt
2. Grau: Entdeckt
3. Schwarz: Suche für u abgeschlossen

Eigenschaften: Algorithmus besucht alle $u \in V$ mit $d(s,u) = k$, bevor irgendein $w \in V$ mit $d(s,w) = k+1$ besucht wird.
Breitensuchbaum: Anfangs nur s, dann jeder weiße Knoten v in $Adj[u]$ an u angehängt $\rightarrow u$ Vorfahre (Elternknoten) von v.



Jeder Knoten speichert Attribute: $u \in V$:

1. $f(u)$ Farbe
2. $\pi(u)$ = Vorgänger bzw. NIL
3. $d(u)$ Abstand von Wurzel

Algorithmus benutzt FIFO-Warteschlange Q (first-in-first-out):
Operationen:

1. $ENQUEUE(Q,x)$: $x \in V$ an Q anfügen, i.Z. $Q \leftarrow x$

2. $DEQUEUE(Q)$: 1. Element entfernen und zurückgeben, i.Z. $x \leftarrow Q$

$$DEQUEUE \leftarrow [v_1, v_2, \ldots, v_k] \leftarrow ENQUEUE$$

Annahme:
Der Graph $G = (V, E)$ wird durch Adjazenzlisten dargestellt.

Q enthält alle grauen Knoten
Am Anfang: $Q = [s]$

Algorithm 28 BFS(G, s)

```
1: for jeden Knoten u ∈ G.V do
2:     u.farbe = WEISS
3:     u.d = ∞
4:     u.π = NIL
5: end for
6: s.farbe = GRAU
7: s.d = 0
8: s.π = NIL
9: Q = ∅
10: ENQUEUE(Q, s)
11: while Q ≠ ∅ do
12:     u = DEQUEUE(Q)
13:     for jeden Knoten v ∈ G.Adj[u] do
14:         if v.farbe == WEISS then
15:             v.farbe = GRAU
16:             v.d = u.d + 1
17:             v.π = u
18:             ENQUEUE(Q, v)
19:         end if
20:     end for
21:     u.farbe = SCHWARZ
22: end while
```

Laufzeit:
Eingabe: $G = (V, E)$
Eigenschaften:

1. Jeder Knoten derselben Komponente wie s wird genau einmal an Q angehängt.
   Kosten: $\mathcal{O}(1) \Rightarrow$ Warteschlangenoperationen kosten $\mathcal{O}(|V|)$
2. Jede Adjazenzliste wird einmal durchsucht ($Adj[u]$, wenn $u \in Q$)
   $\sum_{u \in V} |Adj[u]| = \sum_{u \in V} d(u) = 2|E| \rightarrow \mathcal{O}(|E|)$
3. Initialisierung: $\mathcal{O}(|V|)$

$\Rightarrow$ Kosten von BFS: $\mathcal{O}(|V| + |E|)$

Korrektheit:
Gegeben: $G = (V, E), s \in V$

$$d(s,v) = \begin{cases} \min\limits_{\text{Wege } s \rightsquigarrow v} \text{Anzahl der Kanten des Weges } s \rightsquigarrow v & \text{, falls } \exists \text{ Weg} \\ \infty & \text{, sonst} \end{cases}$$

$d : V \times V \rightarrow \mathbb{N} \cup \{\infty\}$ ist eine Metrik:

1. Definitheit: $d(x,y) \geq 0, d(x,y) = 0 \Leftrightarrow x = y$
2. Symmetrie: $d(x,y) = d(y,x)$
3. Dreiecksungleichung: $d(x,z) \leq d(x,y) + d(y,z)$

10.6 Lemma. $uv \in E \Rightarrow d(s,v) \leq d(s,u) + d(u,v) = d(s,u) + 1$

10.7 Lemma. Annahme: $BFS(u,s)$ ausgeführt $\Rightarrow \forall v \in V : d(v) \geq d(s,v)$
Beweis:
Induktion nach Anzahl der $ENQUEUE$-Aufrufe
1. Aufruf: $d(s) = 0, d(s,s) = 0 \quad \forall v \neq s : d(v) = \infty \geq d(s,v)$
Sei u der aktuelle Knoten: $d(u) \geq d(s,u)$.
Nun wird ein weißer Knoten v entdeckt $\Rightarrow d(v) := d(u) + 1 \geq d(s,u) + 1 \geq d(s,v) \rightarrow v$ grau
$\Rightarrow d(u)$ bleibt unverändert. ■

10.8 Lemma. $BFS(G,s)$ werde gerade ausgeführt, $Q = [v_1, v_2, \ldots, v_r] \Rightarrow d(v_r) \leq d(v_1) + 1, \forall i : 1 \leq i \leq r-1 : d(v_i) \leq d(v_{i+1})$
Beweis:
Induktion nach Anzahl der Warteschlangenoperationen ($ENQUEUE, DEQUEUE$)

Induktionsanfang: $Q = [s]$
Induktionsschritt: Fallunterscheidung:

1. $DEQUEUE$: $\rightarrow Q = [v_2, \ldots, v_r]$. Laut Induktionsvoraussetzung gilt: $d(v_1) \leq d(v_2), d(v_r) \leq d(v_1) + 1 \leq d(v_2) + 1$
2. $ENQUEUE$: $\rightarrow Q = [v_1, v_2, \ldots, v_r, v_{r+1}], v_{r+1} \in Adj[u]$
   $\Rightarrow u \leftarrow Q$ war unmittelbar vorher $\Rightarrow d(u) \leq d(v_1) \Rightarrow d(v_{r+1}) := d(u) + 1 \leq d(v_1) + 1$
   $d(v_r) \overset{\text{IV}}{\leq} d(u) + 1 = d(v_{r+1})$

■

10.9 Korollar. $Q \leftarrow v_i$ bevor $Q \leftarrow v_j \Rightarrow d(v_i) \leq d(v_j)$

10.10 Satz. BFS ist korrekt, d.h., nach der Ausführung gilt
$\forall v \in V : d(v) = d(s, v)$. Ein kürzester Weg $s \rightsquigarrow v$ ist $s \rightsquigarrow \pi(v) + \pi(v)v$.
Beweis:
Annahme: $\exists v \in V : d(v) > d(s, v)$. Sei $d(s, v)$ minimal ($\Rightarrow v \neq s, d(s, v) < \infty$)
Sei $s - v_1 - v_2 - \cdots - v_k - u - v$ ein kürzester Weg. Dann gilt:

1. $d(s, v) = d(s, u) + 1$
2. $d(u) = d(s, u)$

$$\Rightarrow d(v) > d(s, v) = d(s, u) + 1 = d(u) + 1 \tag{10}$$

Während BFS: $DEQUEUE(Q) = u \Rightarrow u$ aktueller Knoten

1. Fall: v weiß $\Rightarrow d(v) := d(u) + 1$ ↯ zu (10)
2. Fall: v schwarz $\Rightarrow DEQUEUE(Q) = v$ passiert bevor $DEQUEUE(Q) = u \Rightarrow Q \leftarrow v$ bevor $Q \leftarrow u \overset{10.9}{\Rightarrow} d(v) \leq d(u)$ ↯
3. Fall: v grau $\Rightarrow v$ gefärbt nach $DEQUEUE(Q) = w \Rightarrow Q \leftarrow w$ vor $Q \leftarrow u \Rightarrow d(w) \leq d(u)$, aber $d(v) := d(w) + 1 \leq d(u) + 1$ ↯

■

10.11 Bemerkung. $Q = [u, \ldots, v, \ldots, l] \Rightarrow d(u) \leq d(v), d(v) \leq d(l) \leq d(u) + 1$

10.12 Definition. Sei $G = (V, E)$ ungerichtet

1. G heißt Wald $:\Leftrightarrow G$ hat keine Kreise
2. Ein Teilgraph $(\{v_1, \ldots, v_k\}, \{v_i v_{i+1} | \, i = 1, \ldots, k-1\} \cup \{v_k v_1\})$ für $k \geq 3$ heißt Kreis
3. Ein Weg ist eine Kantenfolge ohne Wiederholungen von Knoten oder Kanten
4. G heißt Baum $:\Leftrightarrow G$ ist ein zusammenhängender Wald
5. G heißt zusammenhängend $:\Leftrightarrow \forall v, w \in V : \exists$ Weg $v \rightsquigarrow w$
6. G_1 sei ein maximal zusammenhängender Teilgraph von G. Dann heißt G_1 Zusammenhangskomponente von G.
7. Sei $v \in V$ mit $d(v) \leq 1$. Dann heißt v Blatt.

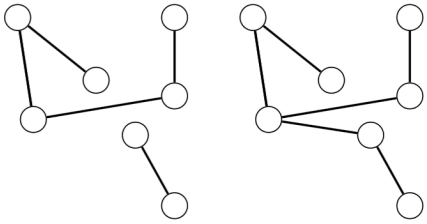


Abbildung 12: Links: Wald, Rechts: Baum

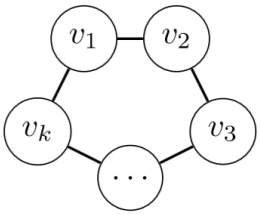


Abbildung 13: Kreis

Sei $R \subseteq V \times V$. Definiere folgende Relation für $v, w \in V$:

$$vRw :\Leftrightarrow \exists \text{Weg } v \rightsquigarrow w$$

Dann gilt:

1. Reflexivität: $vRv \quad \forall v \in V$
2. Symmetrie: $vRw \Rightarrow wRv$
3. Transitivität: $vRw, wRu \Rightarrow vRu$

Damit ist R eine Äquivalenzrelation und induziert eine Partition auf V. Die Äquivalenzklassen sind genau die Zusammenhangskomponenten.

10.13 Satz. Sei $G = (V, E)$ ein Graph. Dann sind folgende Aussagen äquivalent:

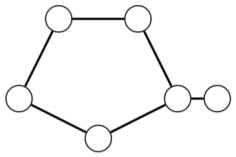
1. G ist ein Baum
2. $\forall v, w \in V : \exists!$ Weg $v \rightsquigarrow w$
3. G ist zusammenhängend und $|V| = |E| + 1$
4. G ist minimal zusammenhängend, d.h. Entfernen einer Kante zerstört den Zusammenhang
5. G ist maximal azyklisch d.h. G hat keine Kreise und Hinzufügen einer Kante erzeugt einen Kreis

10.14 Lemma. Sei G ein Wald mit $|V| \geq 2$. Dann besitzt G mindestens zwei Blätter.
Beweis:
Induktion nach $|V| = n$
$n = 2$: klar
Induktionsschritt: Übung ■

Wir beweisen nur (1) $\Leftrightarrow$ (3).
Beweis:
„$\Rightarrow$“: zu zeigen $|V| = |E| + 1$
Induktion nach $n = |V|$
IA: $n = 1 : |V| = 1, |E| = 0$
IS: $n > 1$: Nach obigen Lemma existiert ein Blatt l. Definiere $G' := G - \{l\}$. Dann gilt: $|V(G')| = n - 1 \overset{IV}{\Rightarrow} |E(G')| = n - 2 \Rightarrow |V(G)| = n,\ |E(G)| = n - 1$

„$\Leftarrow$“: Annahme: $\{v_1, \ldots, v_k\} =: V_k$ bilden einen Kreis.
Sei $G_K = (V_k, E_k)$ dieser Kreis. Dann gilt: $|V_K| = k, |E_k| = k$
$k < |V| \Rightarrow \exists\, v_{k+1} \in V \backslash V_k : \exists i \leq k : v_i v_{k+1} \in E(G)$

$G_{k+1} := (V_k \cup \{v_{k+1}\}, E_k \cup \{v_i v_{k+1}\}) \Rightarrow |V_{k+1}| = |E_{k+1}| = k+1 < |V|$
$\Rightarrow G_{k+2}, G_{k+3}, \cdots \Rightarrow \lightning$



■

BFS baut Breitensuchbaum (BFS-Baum) $\hat{=}$ Vorgängerfunktion π.

$$\begin{aligned} G_\pi &:= (V_\pi, E_\pi) \\ V_\pi &:= \{v \in V | \, \pi(v) \neq NIL\} \cup \{s\} \\ E_\pi &:= \{(\pi(v), v) | \, v \in V_\pi \backslash \{s\}\} \end{aligned}$$

Pfade $s \rightsquigarrow v$ in G_π sind kürzeste Pfade in G.

$$\left.\begin{array}{l} G_\pi \text{ zusammenhängend} \\ |E_\pi| = |V_\pi| - 1 \end{array}\right\} \Rightarrow G_\pi \text{ ist Baum}$$

Nun können wir den kürzesten Weg ausgeben:

Algorithm 29 PRINT-PATH(G, s, v)

```
1: if v == s then
2:     print s
3: else if v.π == NIL then
4:     print „es gibt keinen Pfad von s nach v“
5: else
6:     PRINT-PATH(G, s, v.π)
7:     print v
8: end if
```

10.3 Tiefensuche (Depth-First-Search, DFS)

1. Untersucht Kanten, die inzident zum letzten Knoten v sind, der noch nicht untersuchte ausgehende Knoten hat

2. Sobald alle mit v inzidenten Kanten untersucht: Suche kehrt zurück und untersucht Kanten des Knotens, von dem aus v entdeckt wurde
3. So lange fortsetzen bis alle Knoten der Komponente entdeckt wurden
4. Falls nötig: Neuer Startknoten und fortsetzen der Suche

Vorgängergraph: $G_\pi = (V, E_\pi)$

$$E_\pi = \{(\pi(v), v) |\, v \in V, \pi(v) \neq NIL\}$$
$G_\pi \ldots$ Tiefensuchwald (DFS-Wald)

Zustände der Knoten:

1. Weiß: unentdeckt
2. Grau: entdeckt
3. Schwarz: $Adj[Knoten]$ vollständig abgearbeitet

Jeder Knoten speichert wieder einige Attribute:

1. $c(u) \ldots$ Farbe
2. $d(u) \ldots$ Zeitpunkt des Grauwerdens
3. $f(u) \ldots$ Zeitpunkt des Schwarzwerdens

Algorithm 30 DFS(G)

```
1: for jeden Knoten u ∈ G.V do
2:     u.c = WEISS
3:     u.π = NIL
4: end for
5: zeit = 0
6: for jeden Knoten u ∈ G.V do
7:     if u.c == WEISS then
8:         DFS-VISIT(G, u)
9:     end if
10: end for
```

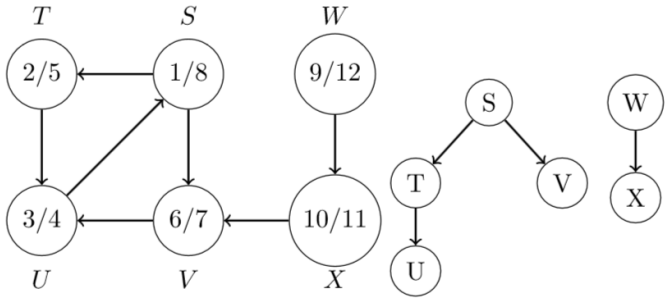

Algorithm 31 DFS-VISIT(G, u)

```
1:  zeit = zeit + 1    //der weiße Knoten u wurde gerade entdeckt
2:  u.d = zeit
3:  u.c = GRAU
4:  for jeden Knoten v ∈ G.Adj[u] do    //verfolge die Kante (u, v)
5:      if v.c == WEISS then
6:          v.π = u
7:          DFS-VISIT(G, v)
8:      end if
9:  end for
10: u.c = SCHWARZ    //färbe u schwarz; er ist abgearbeitet
11: zeit = zeit + 1
12: u.f = zeit
```

10.15 Bemerkung. $1 \leq d(u) < f(u) \leq 2|V|$

Laufzeit:
$\Theta(|V|) + T_{DFS\text{-}VISIT}$
$DFS\text{-}VISIT$: $1\times$ aufgerufen für jedes $v \in V : v \rightarrow$ grau
Schleife: $|Adj[v]| \Rightarrow \Theta(|E|)$
Kosten: $\Theta(|V| + |E|)$

10.16 Beispiel.



Klammernstruktur:

1	2	3	4	5	6	7	8	9	10	11	12
(s	(t	(u	u)	t)	(v	v)	s)	(w	(x	x)	w)

$v \to$grau: $(v$
$v \to$schwarz: $v)$

10.17 Satz. (Klammerntheorem)
Bei DFS in $G = (V, E)$ (gerichtet oder ungerichtet) gilt:
$\forall u, v \in V$ ist genau eine der folgenden Bedingungen erfüllt:

1. $[d(u), f(u)] \cap [d(v), f(v)] = \emptyset$, weder u noch v ist Nachfahre von dem jeweils anderen Knoten im DFS-Wald
2. $[d(u), f(u)] \subseteq [d(v), f(v)]$, u Nachfahre von v
3. $[d(v), f(v)] \subseteq [d(u), f(u)]$, v Nachfahre von u

Beweis:
oBdA. $d(u) < d(v)$

1. Fall 1: $d(v) < f(u) \Rightarrow v$ entdeckt während u grau $\Rightarrow v$ Nachfahre von u
   Aber $d(u) < d(v) \Rightarrow v$ später entdeckt als $u \Rightarrow$ alle von v ausgehenden Kanten erforscht $\Rightarrow v$ fertig, bevor der Algorithmus zu u zurückkehrt $\Rightarrow f(v) < f(u)$
2. Fall 2: $d(v) > f(u) \Rightarrow d(u) < f(u) < d(v) < f(v)$

■

10.18 Korollar. v ist ein echter Nachfahre von $u \Leftrightarrow d(u) < d(v) < f(v) < f(u)$

10.19 Satz. (Satz der weißen Pfade)
In einem DFS-Wald von $G = (V, E)$ ist v genau dann ein Nachfahre von u, wenn es zum Zeitpunkt $d(u)$ einen Pfad $u \rightsquigarrow v$ gibt, der nur aus weißen Knoten besteht.
Beweis:
„$\Rightarrow$“:

1. Fall: $v = u \Rightarrow u \rightsquigarrow v$ besteht nur aus u, u weiß, wenn $d(u) := Zeit$
2. Fall: v ist echter Nachfahre von $u \Rightarrow d(u) < d(v) \Rightarrow$ zum Zeitpunkt $d(u)$ ist v weiß, (v beliebig) $\Rightarrow$ alle Nachfahren auf Pfad $u \rightsquigarrow v$ sind weiß

„$\Leftarrow$“:
Zur Zeit $d(u)$ gebe es einen Pfad $u \rightsquigarrow v$ aus lauter weißen Knoten. Annahme: v kein Nachfahre im DFS-Wald

oBdA.: alle Knoten auf $u \rightsquigarrow v$ außer v selbst, sind Nachfahren von u.
Sei w Vorgänger von v auf $u \rightsquigarrow v \Rightarrow w$ Nachfahre von $u \Rightarrow f(w) \leq f(u) \Rightarrow v$ wird entdeckt, nachdem u entdeckt und bevor w fertig ist
$\Rightarrow d(u) < d(v) < f(w) \leq f(u) \overset{\text{Klammerntheorem}}{\Rightarrow} f(v) < f(u) \Rightarrow v$ Nachfolger von u $\quad \lightning$ ■

Klassifikation der Kanten in $G = (V, E)$

1. **Baumkanten** (E_B): Kanten des DFS-Waldes G_π, $(u, v) \in E_B \Leftrightarrow v$ beim Untersuchen von u entdeckt
2. **Rückwärtskanten** (E_R): Kanten (u, v), die u mit Vorfahren v im DFS-Wald verbinden
3. **Vorwärtskanten** (E_V): Kanten (u, v) die u mit Nachfahren v im DFS-Wald verbinden
4. **Querkanten** (E_Q): alle übrigen Kanten

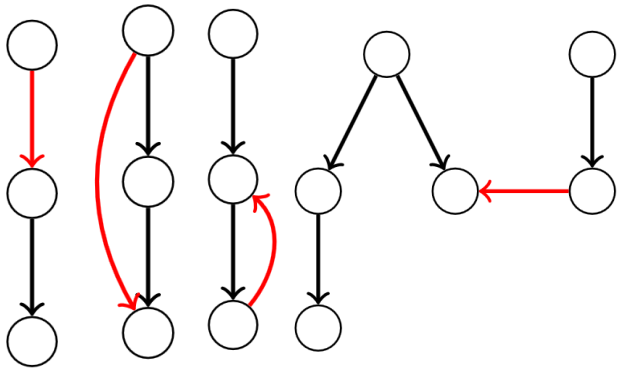


Abbildung 14: Baum-, Vorwärts-, Rückwärts- und Querkante

Während DFS: DFS untersucht (u, v):

1. v weiß $\Rightarrow (u, v) \in E_B$
2. v grau: graue Knoten bilden Pfade aus lauter Nachfahren eines Knoten v_0 ($\hat{=}$ Stack der Aufrufe des DFS-Visit) $\Rightarrow (u, v) \in E_R$
3. v schwarz $\Rightarrow (u, v) \in E_V \cup E_Q$

a) $(u, v) \in E_V \Leftrightarrow d(u) < d(v)$

b) $(u, v) \in E_Q \Leftrightarrow d(u) > d(v)$

G ungerichtet $\Rightarrow (u, v)$ wird zweimal untersucht werden.

10.20 Satz. DFS auf $G = (V, E)$ ungerichtet $\Rightarrow$ Es gibt nur Baum- und Rückwärtskanten d.h. $E = E_B \cup E_R$.
Beweis:
Sei (u, v) eine beliebige Kante, oBdA. $d(u) < d(v)$
$\Rightarrow$ DFS muss v fertig untersucht haben, bevor u fertig ($f(u) > f(v)$)
Währenddessen: u grau, da $v \in Adj[u]$
Zum ersten Zeitpunkt wo (u, v) von u in Richtung v untersucht wird, ist v noch weiß $\Rightarrow (u, v) \in E_B$
Falls zu diesem Zeitpunkt (u, v) in Richtung u untersucht $\Rightarrow u$ grau $\Rightarrow (u, v) \in E_R$ ■

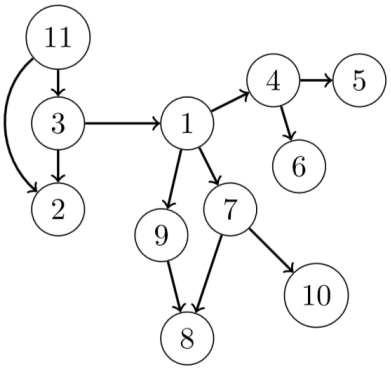
10.21 Bemerkung. Bei einem ungerichtetem Graphen wird eine Kante bei der ersten Untersuchung einer Kategorie zugeordnet.

10.4 Topologisches Sortieren

10.22 Definition. Betrachte gerichteten, azyklischen Graph (Directed Acyclic Graph, kurz DAG)
Eine topologische Sortierung ist eine lineare Ordnung der Knoten des DAGs derart, dass u vor v, falls $(u, v) \in E$

10.23 Bemerkung. Falls es Zyklen gibt $\Rightarrow \nexists$ topologische Sortierung.

10.24 Beispiel.



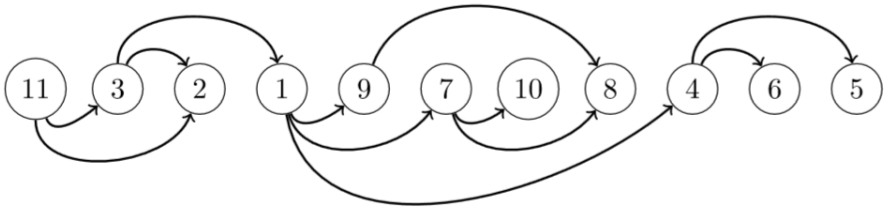


Abbildung 15: Beispiel für *TOP-SORT*

TOP-SORT funktioniert wie DFS, wobei jeder abgearbeitete Knoten an den Kopf einer verketteten Liste eingefügt wird. Diese wird dann zurückgegeben:

Algorithm 32 TOP-SORT(G)

```
1: for jeden Knoten u ∈ G.V do
2:     u.c = WEISS
3:     u.π = NIL
4: end for
5: Sei s eine verkettete Liste
6: zeit = 0
7: for jeden Knoten u ∈ G.V do
8:     if u.c == WEISS then
9:         DFS-VISIT'(G, u)
10:    end if
11: end for
12: return s
```

Algorithm 33 DFS-VISIT'(G, u)

```
1: zeit = zeit + 1    //der weiße Knoten u wurde gerade entdeckt
2: u.d = zeit
3: u.c = GRAU
4: for jeden Knoten v ∈ G.Adj[u] do    //verfolge die Kante (u, v)
5:     if v.c == WEISS then
6:         v.π = u
7:         DFS-VISIT'(G, v)
8:     end if
9: end for
10: u.c = SCHWARZ    //färbe u schwarz; er ist abgearbeitet
11: zeit = zeit + 1
12: u.f = zeit
13: s.push_front(u)    //füge Knoten an den Kopf der verketteten Liste ein
```

Aufwand: $\Theta(|V| + |E|) + \Theta(|V|) = \Theta(|V| + |E|)$

10.25 Lemma. $G = (V, E)$ gerichtet, azyklisch $\Leftrightarrow$ DFS liefert keine Rückwärtskante.
Beweis:
„$\Rightarrow$“
Ann: $\exists e \in E_R \Rightarrow v$ Vorfahre von $u \Rightarrow \exists$ Pfad $P : u \rightsquigarrow v \Rightarrow P + (u, v)$ ist Zyklus $\lightning$
„$\Leftarrow$“
Angenommen G enthält Zyklus C. Sei $v \in V(C)$ der erste Knoten von C, der entdeckt wird.
$(u, v) \in E(C) \Rightarrow$ zum Zeitpunkt $d(u)$ bildet $V(C)$ einen Pfad $v \rightsquigarrow u$ aus lauter weißen Knoten $\Rightarrow u$ Nachfahre von v in DFS-Wald $\Rightarrow (u, v) \in E_R$ $\lightning$ ■

10.26 Satz. *TOP-SORT* ist korrekt
Beweis:
DFS auf G anwenden $\Rightarrow f(v)$ bestimmt
zu zeigen: $u \neq v, (u, v) \in E \Rightarrow f(v) < f(u)$
Werde (u, v) gerade von DFS untersucht $\Rightarrow c(v) \neq$ grau, denn sonst $(u, v) \in E_R$.
$c(v)$ weiß $\Rightarrow v$ Nachfahre von u im DFS-Wald $\Rightarrow f(v) < f(u)$.
$c(v)$ schwarz $\Rightarrow f(v)$ bereits gesetzt $\Rightarrow f(v) < f(u)$ ■

10.27 Bemerkung. Die Beweisidee von Satz 10.26 kann man sich auch wie folgt vorstellen:

Ein Knoten wird erst dann schwarz gefärbt, wenn alle Nachfolger schon schwarz sind - wegen Lemma 10.25 gibt es keine grauen Nachfolger.
Wenn eine Element auf die Top-Sort-Liste gesetzt wird, also schwarz gefärbt wird, sind alle Nachfolger bereits schwarz und damit weiter rechts auf der Liste, was genau das Ziel von Top-Sort ist. Damit folgt die Korrektheit von Top-Sort.

11 Dynamische Programmierung

1. Divide & Conquer
   a) Problem ⇒ Zerlegen in unabhängige Teilprobleme
   b) Rekursives Lösen von den Teilproblemen
   c) Eine Lösung aus den Teillösungen konstruieren
2. Dynamische Programmierung
   a) Problem ⇒ Zerlegen in Teilprobleme (nicht mehr unabhängig!)
   b) Teilprobleme lösen und das Ergebnis speichern und wiederverwenden
   c) Eine Lösung für das Problem aus den Lösungen der Teilprobleme konstruieren

Oft Optimierungsprobleme, d.h., es ist eine optimale Lösung gesucht (meist Minimum oder Maximum)

4 Schritte:

1. Charakterisieren der Struktur einer optimalen Lösung $\rightsquigarrow$ Wert
2. Definieren des Werts (rekursiv) einer optimalen Lösung
3. Berechnen des Wertes einer optimalen Lösung
4. Konstruieren einer optimalen Lösung

11.1 Beispiel. Schneiden von Eisenstangen (Stabzerlegungsproblem, Rod Cutting)
Gegeben:

1. Eine Stange der Länge n
2. Eine Preisliste $p = \langle p_1, p_2, \ldots, p_n \rangle$

Gesucht: Der maximale Erlös r_n

Länge	1	2	3	4	5	6	7	8	9	10
Preis	1	5	8	9	10	17	17	20	24	30

Es gibt 2^{n-1} Möglichkeiten den Stab zu zerschneiden.

$$r_n = \max_{1 \leq k < n} \{p_n, r_k + r_{n-k}\}$$

11.1 Optimale Teilstruktur Eigenschaft

$$r_n = \max_{1 \leq k \leq n} \{p_k + r_{n-k}\}, \quad r_0 = 0$$

Algorithm 34 CUT-ROD(p, n)

```
1: if n == 0 then
2:     return 0
3: end if
4: q = −∞
5: for i = 1 to n do
6:     q = max{q, p[i] + CUT-ROD(p, n − i)}
7: end for
8: return q
```

Laufzeit:

$$T(n) = 1 + \sum_{i=0}^{n-1} T(i)$$
$$T(n) - T(n-1) = T(n-1)$$
$$\Rightarrow T(n) = 2T(n-1) = \cdots = 2^n$$

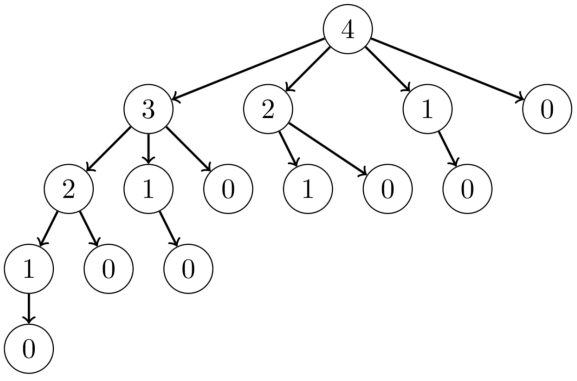


Abbildung 16: Rekursionbaum von CUT-ROD für $n = 4$

11.2 Top-Down Ansatz mit Memoisation

Algorithm 35 MEMOISED-CUT-ROD(p, n)

```
1: Sei r[0...n] ein neues Feld
2: for i = 0 to n do
3:     r[i] = −∞
4: end for
5: return MEMOISED-CUT-ROD-AUX(p, n, r)
```

Algorithm 36 MEMOISED-CUT-ROD-AUX(p, n, r)

```
1: if r[n] ≥ 0 then
2:     return r[n]
3: end if
4: if n == 0 then
5:     q = 0
6: else
7:     q = −∞
8:     for i = 1 to n do
9:         q = max(q, p[i] + MEMOISED-CUT-ROD-AUX(p, n − i, r))
10:    end for
11:    r[n] = q
12: end if
13: return q
```

Laufzeit:

$$\sum_{k=1}^{n} k = \Theta\left(n^2\right)$$

11.3 Bottum-up Ansatz

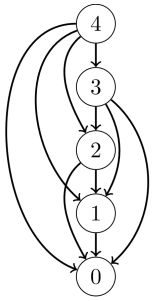
Algorithm 37 BOTTOM-UP-CUT-ROD(p, n)

```
1: Sei r[0...n] ein neues Feld
2: r[0] = 0
3: for j = 1 to n do
4:     q = -∞
5:     for i = 1 to j do
6:         q = max(q, p[i] + r[j - i])
7:     end for
8:     r[j] = q
9: end for
10: return r[n]
```

Laufzeit: $\Theta(n^2)$ (kleinere Konstante als bei $MEMOISED\text{-}CUT\text{-}ROD$)

11.4 Teilproblem-Graphen

1. Gerichteter Graph
2. Die Knoten entsprechen verschiedenen Teilproblemen
3. Eine Kante vom Knoten x zum Knoten y bedeutet, dass um eine optimale Lösung für Teilproblem x zu finden, wir zuerst eine optimale Lösung für das Teilproblem y bestimmen müssen



Bottum-up $\hat{=}$ umgekehrte topologische Sortierung
Normalerweise Laufzeit $= \mathcal{O}(|V| + |E|)$
Top-Down mit Memoisation $\hat{=}$ Tiefensuche

Algorithm 38 EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

```
1: Sei r[0...n] und s[0...n] neue Felder
2: r[0] = 0
3: s[0] = 0
4: for j = 1 to n do
5:     q = -∞
6:     for i = 1 to j do
7:         if q < p[i] + r[j - i] then
8:             q = p[i] + r[j - i]
9:             s[j] = i
10:        end if
11:    end for
12:    r[j] = q
13: end for
14: return r und s
```

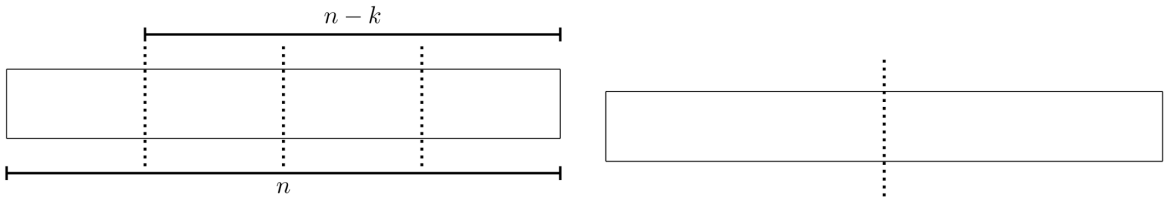
11.2 Beispiel. Stabzerlegungsproblem von oben:

Länge	0	1	2	3	4	5	6	7	8	9	10
Preis	0	1	5	8	9	10	17	17	20	24	30
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

11.5 Eigenschaften von Dynamischer Programmierung

1. Optimale Teilstruktur
2. Überlappende Teilprobleme

Optimale Teilstruktur:
Wenn wir eine optimale Lösung für das Problem haben, enthält diese Lösung optimale Lösungen für Teilprobleme.



Allgemeines Muster:

1. Zeigen, dass eine Lösung darin besteht, eine Entscheidung zu treffen, wie das Problem aufgeteilt wird.

2. Annahme, dass eine Entscheidung existiert, die zu einer optimalen Lösung führt

3. Bestimmen, welche Teilprobleme sich dadurch ergeben

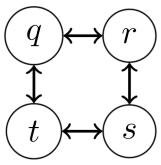
4. Zeigen, dass die Lösungen für die Teilprobleme selbst optimal sind

(cut-and-paste Methode)

Knoten $\approx$ Teilprobleme
Kanten $\approx$ welche Teilprobleme für das Problem zu betrachten sind

Wichtig: Es gibt Probleme, die keine optimale Teilstruktur haben.

11.3 Beispiel. Sei $G = (V, E)$, $u, v \in V$
Ungewichteter kürzester Pfad (UKP): Einen Pfad $u \rightsquigarrow v$ zu finden, der die kleinste Länge hat. Hier stimmt die optimale Teilstruktur.
$u \rightsquigarrow w \rightsquigarrow v$ kürzester Pfad $\Rightarrow u \rightsquigarrow w, w \rightsquigarrow v$ kürzeste Pfade
Ungewichteter längster Pfad (ULP): Einen einfachen Pfad $u \rightsquigarrow v$ zu finden, der die größte Länge hat. Dieses Problem hat keine optimale Teilstruktur.
Gegenbeispiel:



Überlappende Teilprobleme:
Der Algorithmus verwendet immer wieder Lösungen für die gleichen Teilprobleme.
Typischerweise: Anzahl der Teilprobleme ist polynomiell in der Eingabegröße.

Problem 2: Längste gemeinsame Teilsequenz (Longest Common Sequence, LCS) (z.B. DNA)

$$S_1 = ATTGCGAATGA$$
$$S_2 = ATGGCGTAAGA$$
$$\Rightarrow ATGCGAAGA \text{ ist LCS}$$

11.4 Definition. Sei $X = \langle x_1, x_2, \ldots, x_m \rangle$. $Z = \langle z_1, z_2, \ldots, z_k \rangle$ heißt eine Teilsequenz von $X \Leftrightarrow \exists i_1 < i_2 < \cdots < i_k$, sodass $x_{i_j} = z_j$.

11.5 Definition. Seien $X = \langle x_1, x_2, \ldots, x_m \rangle$, $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Dann heißt $Z = \langle z_1, z_2, \ldots, z_k \rangle$ Teilsequenz von X und $Y \Leftrightarrow \exists i_1 < i_2 < \cdots < i_k$, $j_1 < j_2 < \cdots < j_k$: $x_{i_l} = z_l$, $y_{j_l} = z_l$.
Z heißt LCS von X und $Y \Leftrightarrow Z$ ist die längste Teilsequenz von X und Y.

11.6 Bemerkung. Es gibt 2^m Teilsequenzen von $X = \langle x_1, x_2, \ldots, x_m \rangle$.

11.7 Definition. i-Präfix von $X = \langle x_1, x_2, \ldots, x_m \rangle$ ist $X_i = \langle x_1, x_2, \ldots, x_i \rangle$

11.8 Satz. Seien $X = \langle x_1, x_2, \ldots, x_m \rangle$, $Y = \langle y_1, y_2, \ldots, y_n \rangle$ und sei $Z = \langle z_1, z_2, \ldots, z_k \rangle$ LCS von X und Y. Dann gilt:

1. Ist $x_m = y_n$, dann $z_k = x_m = y_n$ und $Z_{k-1} = \langle z_1, z_2, \ldots, z_{k-1} \rangle$ ist eine LCS von X_{m-1}, Y_{n-1}
2. Ist $x_m \neq y_n$, dann folgt aus $z_k \neq x_m$, dass Z eine LCS von X_{m-1} und Y ist
3. Ist $x_m \neq y_n$, dann folgt aus $z_k \neq y_n$, dass Z eine LCS von Y_{n-1} und X ist

Beweis:
Übung ■

Sei die Matrix C folgendermaßen definiert:

$$C[i,j] = \begin{cases} 0 & \text{, wenn } i = 0 \vee j = 0 \\ C[i-1, j-1] + 1 & \text{, wenn } x_i = y_j \\ \max(C[i-1,j], C[i,j-1]) & \text{, wenn } x_i \neq y_j \end{cases}$$
$$\Rightarrow C[i,j] \text{ ist die Länge der LCS für } X_i, Y_j$$

Algorithm 39 LCS-LENGTH(X, Y)

```
1: m = X.length
2: n = Y.length
3: Seien b[1...m, 1...n] und C[0...m, 0...n]
4: for i = 1 to m do
5:     C[i, 0] = 0
6: end for
7: for j = 0 to n do
8:     C[0, j] = 0
9: end for
10: for i = 1 to m do
11:     for j = 1 to n do
12:         if x_i == y_j then
13:             C[i, j] = C[i − 1, j − 1] + 1
14:             b[i, j] = „↖“
15:         else if C[i − 1, j] ≥ C[i, j − 1] then
16:             C[i, j] = C[i − 1, j]
17:             b[i, j] = „↑“
18:         else
19:             C[i, j] = C[i, j − 1]
20:             b[i, j] = „←“
21:         end if
22:     end for
23: end for
```

11.9 Beispiel.

$$X = \langle A, B, C, B, D, A, B \rangle \quad Y = \langle B, D, C, A, B, A \rangle$$

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0↑	0↑	0↑	1↖	1←	1↖
2	B	0	1↖	1←	1←	1↑	2↖	2←
3	C	0	1↑	1↑	2↖	2←	2↑	2↑
4	B	0	1↖	1←	2↑	2↑	3↖	3←
5	D	0	1↑	2↖	2↑	2↑	3↑	3↑
6	A	0	1↑	2↑	2↑	3↖	3↑	4↖
7	B	0	1↖	2↑	2↑	3↑	4↖	4↑

$\Rightarrow BCBA$ ist LCS

Aufwand: $\mathcal{O}(mn)$

12 Minimale Spannbäume

Generalvoraussetzung:
Der Graph $G = (V, E)$ sei zusammenhängend und ungerichtet.

12.1 Allgemeines

Sei $G = (V, E)$ ein ungerichteter Graph, $w : E \to \mathbb{R}$ eine Gewichtsfunktion, $F \subseteq E$: $w(F) = \sum_{e \in F} w(e)$

12.1 Definition. $F \subseteq E$, $T = (V, F)$ heißt Spannbaum von $G :\Leftrightarrow (V, F)$ ist ein Baum.

Gegeben: $G = (V, E)$, $w : E \to \mathbb{R}$
Gesucht: Spannbaum mit minimalem Gewicht ($|V| - 1$ Kanten, minimal spanning tree MST)

Generische Methode: Erzeugen Mengen $A \subseteq E$, sodass vor bzw. nach jeder Iteration gilt: (V, A) ist Teilgraph eines minimalen Spannbaumes $T = (V, F)$.

Algorithm 40 GENERIC-MST(G, w)

```
1: A := ∅
2: while (V, A) ≠ Spannbaum do
3:     finde e ∈ E mit (V, A ∪ {e}) Teilgraph eines minimalen Spannbaums
4:     A := A ∪ {e}
5: end while
6: return A
```

12.2 Definition. $C := (S, V \backslash S), S \subseteq V, S \neq \emptyset, S \neq V$ heißt Schnitt.
$e = v_1 v_2$ Kante von $C :\Leftrightarrow v_1 \in S, v_2 \in V \backslash S$.
e heißt leichte Kante von $C :\Leftrightarrow e$ Kante von C und $w(e)$ ist minimal.

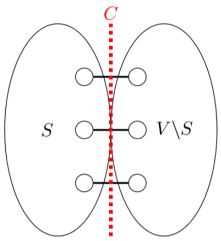


Abbildung 17: Schnitt von S und $V \backslash S$

12.3 Satz. Seien $G = (V, E)$ ein zusammenhängender, ungerichteter Graph, $w : E \to \mathbb{R}$, $F \subseteq E$, $T := (V, F)$ ein minimaler Spannbaum, $A \subseteq F$, $C = (S, V \backslash S)$ ein Schnitt und C respektiere die Kantenmenge A, d.h. kein $e \in A$ ist Kante von C. e sei eine leichte Kante von C. Dann gilt: $\exists$ minimaler Spannbaum $T' = (V, F') : A \cup \{e\} \subseteq F'$
Beweis:

1. Fall: $e \in F$ $(T' := T = (V, F)) \Rightarrow$ klar

2. Fall: $e \notin F, e = (u, v) \Rightarrow u$ und v sind verbunden in T (Pfad $P \subseteq E$, sogar $P \subseteq F$) $\Rightarrow P \cup \{e\}$ ist Kreis, $u \in S$, $v \in V \backslash S \Rightarrow \exists f \in P : f$ liegt in C, $f = (x, y) \notin A, x \in S, y \notin S$
   $T \backslash \{f\} = (V, F \backslash \{f\}) \ldots$ zwei Komponenten $K_1, K_2 \subseteq V$, $u \in K_1$, $v \in K_2 \Rightarrow T' = (T \backslash \{f\}) \cup \{e\}$ ist Spannbaum.
   zu zeigen: T' minimal: E leicht $\Rightarrow w(e) \leq w(f) \Rightarrow w(T') \leq w(T) \Rightarrow w(T') = w(T)$

■

Ad *GENERIC-MST*:
$G_A = (V, A)$ ist Wald
Am Anfang: $G_\emptyset = (V, \emptyset)$, $|V|$ Komponenten
$A := A \cup \{e\}$, e verbindet zwei Komponenten von G_A
$\Rightarrow |V| - 1$ Iterationen der *while*-Schleife

12.4 Korollar. Seien $G = (V, E)$ ein zusammenhängender, ungerichteter Graph, $w : E \to \mathbb{R}$, $F \subseteq E$, $T := (V, F)$ ein minimaler Spannbaum, $A \subseteq F$. Sei $K = (V_K, E_K)$ eine Zusammenhangskomponente von $G_A = (V, A)$. Dann gilt:
Falls $e \in E$ leicht bzgl. $(V_K, V \backslash V_K) \Rightarrow (V, A \cup \{e\})$ ist kreisfrei.

Beweis:
$(V_K, V \backslash V_K)$ respektiert A und e leicht ■

12.2 Der Algorithmus von Kruskal

Gegeben: $E = \{e_1, e_2, \ldots, e_m\}$, $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$

Algorithm 41 MST-KRUSKAL(G, w)

```
1: A := ∅
2: for jeden Knoten v ∈ G.V do
3:     MAKE-SET(v)
4: end for
5: sortiere die Kanten aus G.E in nichtfallender Reihenfolge nach dem Gewicht w
6: for jede Kante (u, v) ∈ G.E, in nichtfallender Reihenfolge nach ihrem Gewicht do
7:     if FIND-SET(u) ≠ FIND-SET(v) then
8:         A = A ∪ {(u, v)}
9:         UNION(u, v)
10:    end if
11: end for
12: return A
```

Der Algorithmus verwendet folgende Hilfsfunktionen:

1. $MAKE\text{-}SET(v)$: Erzeugt eine neue Menge V mit $v \in V$
2. $FIND\text{-}SET(u)$: Gibt die Menge U zurück, wo $u \in U$
3. $UNION(u, v)$: Vereinigt die beiden Mengen U und V mit $u \in U$ und $v \in V$

12.5 Bemerkung. Die Sets entsprechen den Zusammenhangskomponenten und $M := \{\text{Sets}|$ die momentan vorhanden sind$\}$ ist immer eine Partition des Graphen G. Am Anfang gilt: $M = \{\{v\} | \, v \in V\}$

$MST\text{-}KRUSKAL$ gehört zu den Greedy-Algorithmen. Die Zwischenergebnisse des Algorithmus müssen keine Bäume sein.

12.3 Der Algorithmus von Prim

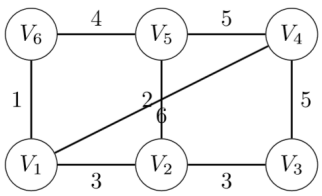
Sei $r \in V$ ein beliebiger Startpunkt.

Algorithm 42 MST-PRIM(G, w, r)

```
1: for each u ∈ G.V do
2:     u.key = ∞
3:     u.π = NIL
4: end for
5: r.key = 0
6: Q = G.V
7: while Q ≠ ∅ do
8:     u = EXTRACT-MIN(Q)
9:     for each v ∈ G.Adj[u] do
10:        if v ∈ Q und w(u, v) < v.key then
11:            v.π = u
12:            v.key = w(u, v)
13:        end if
14:    end for
15: end while
```

Schleifeninvariante: $A = \{(v, \pi(v)) | \, v \in V \setminus (\{r\} \cup Q)\}$ ist immer Baum, da zusammenhängend und kreisfrei.

12.6 Beispiel.



	$g(V_2)$	$g(V_3)$	$g(V_4)$	$g(V_5)$	$g(V_6)$
V_1	3	∞	2	∞	1
V_6	3	∞	2	4	-
V_4	3	5	-	4	-
V_2	-	3	-	4	-
V_3	-	-	-	4	-

13 Matroide und Greedy-Algorithmen

13.1 Definition. Ein Tupel (E, S) heißt Unabhängigkeitssystem, wenn $E \neq \emptyset$, $S \subseteq \mathcal{P}(E)$ abgeschlossen bzgl. $\subseteq$ d.h. $A \in S$, $B \subseteq A \Rightarrow B \in S$.

Optimierungsproblem:
Gegeben: (E, S) Unabhängigkeitssystem, $w : E \to \mathbb{R}$, $w(A) = \sum_{x \in A} w(x)$.
Gesucht: $A \in S : A$ maximal bzgl. $\subseteq$, $w(A)$ minimal.

13.2 Bemerkung. S heißt auch die Menge der unabhängigen Mengen.

13.3 Beispiel. Sei $G = (V, E)$ ungerichtet, $S = \{F \subseteq E | (V, F) \text{ Wald}\} \Rightarrow (E, S)$ Unabhängigkeitssystem.
Beweis: Sei (V, F) Wald und $F' \subseteq F \Rightarrow (V, F')$ Wald

Algorithm 43 GREEDY(E, S, w)

```
1: Sortiere E = {e_1, e_2, ..., e_m}, w(e_1) ≤ w(e_2) ≤ ··· ≤ w(e_m)
2: T := ∅
3: for k = 1 to m do
4:     if T ∪ {e_k} ∈ S then
5:         T := T ∪ {e_k}
6:     end if
7: end for
```

Für allgemeine Problemstellungen ist a-priori nicht klar, ob der Algorithmus richtig arbeitet. Es lässt sich jedoch zeigen, dass für spezielle Problemklassen, sogenannte Matroide, der Greedy-Algorithmus korrekt ist.

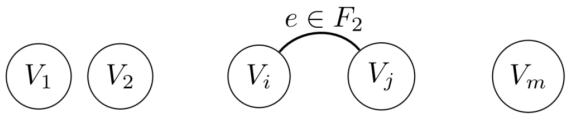
13.4 Definition. Ein Unabhängigkeitssystem $M = (E, S)$ heißt Matroid, falls $\forall A, B \in S$ mit $|B| = |A| + 1$ gilt $\exists v \in B \backslash A : A \cup \{v\} \in S$

13.5 Bemerkung. $|A| < |B|$ genügt.

13.6 Definition. Sei $M = (E, S)$ ein Matroid. $A \in S$ heißt Basis von $M :\Leftrightarrow A$ maximal in S (bzgl. $\subseteq$). $r(M) := |A|$ heißt Rang von M.

13.7 Beispiel. Basen von Vektorräumen: $a_1, a_2, \ldots, a_n \in \mathbb{R}^n$, $E = \{a_1, a_2, \ldots, a_n\}$, $S = \{A \subseteq E | \, A \text{ linear unabhängig } \} \Rightarrow (E, S) = M$ ist Matroid, A Basis $\Leftrightarrow$ $A \in S \wedge |A| = \dim[E]$, $r(M) = \dim[E]$

13.8 Satz. Sei $G = (V, E)$ ein Graph, $S = \{F \subseteq E | \, (V, F) \text{ Wald}\} \Rightarrow (E, S)$ Matroid (graphisches Matroid).
Beweis:
Sei $F_1, F_2 \subseteq E$ und sei F_2 ein Wald, $F_1 \subseteq F_2 \Rightarrow F_1$ Wald
F_1, F_2 Wälder, $|F_2| = |F_1| + 1$
zu zeigen: $\exists e \in F_2 \backslash F_1 \Rightarrow (V, F_1 \cup \{e\})$ Wald
F_1 habe m Komponenten, $T_i = (V_i, A_i), \quad i = 1, \ldots, m, \quad V = \bigcup_{i=1}^{m} V_i, \quad F_1 = \bigcup_{i=1}^{m} A_i$
Es gilt $|A_i| = |V_i| - 1$.
F_2 Wald $\Rightarrow \exists$ höchstens $|V_i| - 1$ Kanten von F_2, die Knoten $v, w \in V_i$, $i = 1, \ldots, m$ verbinden.
$\Rightarrow \exists e \in F_2 \backslash F_1 : F_1 \cup \{e\}$ ist Wald



■

13.9 Satz. Sei $M = (E, S)$ ein Matroid, $w : E \to \mathbb{R} \Rightarrow GREEDY$ löst das Optimierungsproblem „A max $\in S$, $w(A)$ min“ korrekt, d.h. $GREEDY$ berechnet eine Basis minimalen Gewichts.
Beweis:
$A = \{a_1, \ldots, a_r\}$ nach einem Aufruf von $GREEDY$

a) A ist Basis: $A \in S$ nach Konstruktion, Annahme A nicht maximal $\Rightarrow \exists e \notin A$: $A \cup \{e\} \in S$: Zur Zeit als e getestet wird: $A_1 \subseteq A \Rightarrow A_1 \cup \{e\} \in S$ ↯

b) $w(a_1) \leq w(a_2) \leq \cdots \leq w(a_r)$ (o.B.d.A.)

c) $w(A)$ minimal: Annahme: $\exists B$ Basis, $B = \{b_1, \ldots, b_r\}$, $w(b_1) \leq \cdots \leq w(b_r)$, $w(B) < w(A)$
Sei $i := \min\{j \in \mathbb{N} | \, w(b_j) < w(a_j)\}$

$A_{i-1} := \{a_1, a_2, \ldots, a_{i-1}\}$ Status nach $j_i \geq i-1$ Iterationen.
$B_i := \{b_1, b_2, \ldots, b_i\}$
Matroideigenschaft $\Rightarrow \exists b_j \in B_i \backslash A_{i-1} : A_{i-1} \cup \{b_j\} \in S$, aber $w(b_j) \leq w(b_i) < w(a_i) \quad (b_j \notin A) \quad \lightning$, denn $GREEDY$ wählt in jedem Schritt immer optimal

■

13.10 Satz. Sei $M = (E, S)$ ein Unabhängigkeitssystem, $GREEDY$ löse das Problem „A max $\in S$, $w(A)$ min“ korrekt $\forall w : E \to \mathbb{R}$. Dann ist M ein Matroid.
Beweis:
Annahme: $\exists A, B \in S : |B| = |A| + 1$ und $\forall x \in B \backslash A : A \cup \{x\} \notin S$.
Definiere Gewichtsfunktion w:

$$w(e) := \begin{cases} \epsilon & \text{, falls } e \in A \cap B \\ 2\epsilon & \text{, falls } e \in A \backslash B \\ 3\epsilon & \text{, falls } e \in B \backslash A \\ 1 & \text{, sonst} \end{cases}$$

Sei $\epsilon > 0$ hinreichend klein.
$GREEDY$ konstruiert $A \subseteq A'$, $A' \cap (B \backslash A) = \emptyset$
$w(A') = |A \cap B|\epsilon + |A \backslash B|2\epsilon + (|A'| - |A|)$
Sei B' Basis mit $B \subseteq B'$
$w(B') = |A \cap B|\epsilon + |B \backslash A|3\epsilon + (|B'| - |B|)$ falls $B' \cap (A \backslash B) = \emptyset$
$w(A') - w(B') \overset{|A'| \leq |B'|}{\geq} |A \backslash B|2\epsilon - |B \backslash A|3\epsilon - |A| + |B| = |A \backslash B|2\epsilon - |B \backslash A|3\epsilon + 1 > 0$ für ϵ hinreichend klein
$\Rightarrow w(A')$ ist nicht minimal $\quad \lightning$ ■

14 Kürzeste Pfade

14.1 Grundlagen

Gegeben: $G = (V, E)$ gerichteter Graph, $w : E \to \mathbb{R}$, $v_0 \in V$

14.1 Definition. $\delta : V \times V \to \mathbb{R} \cup \{\infty\}$

$$\delta(u, v) = \begin{cases} \min\limits_{v_0=u, v_1, \ldots, v_k=v} \sum\limits_{i=1}^{k} w(v_{i-1}, v_i) & \text{, falls } \exists \text{ Pfad } u \rightsquigarrow v \\ \infty & \text{, sonst} \end{cases}$$

δ erfüllt die Dreiecksungleichung

Gesucht: $\forall v \in V : \delta(v_0, v)$, d.h. der „kürzeste“ Pfad $v_0 \rightsquigarrow v$ im Graphen

Varianten:

1. Sei G ungewichtet $\Rightarrow \delta(u, v) =$ Anzahl Kanten (d.h. $w(e) = 1, \forall e \in E$) $\to$ gelöst durch BFS
2. kürzeste Pfade zu einem Zielknoten ($\Leftrightarrow \delta(v_0, v)\, \forall v \in V$, Orientierung umdrehen)
3. Finde kürzesten Pfad für ein Knotenpaar (u, v), also $\delta(u, v)$: gelöst durch die Wahl $v_0 = u$
4. Finde kürzeste Pfade für alle Knotenpaare, d.h. $\delta(u, v)\, \forall u, v \in V$: Setze $v_0 = u\, \forall u \in V$

14.2 Lemma. $G = (V, E)$, $w : E \to \mathbb{R}$, $p = (v_0, v_1, \ldots, v_k)$ kürzester Pfad $v_0 \rightsquigarrow v_k \Rightarrow \forall\, 0 \leq i \leq j \leq k : (v_i, \ldots, v_j)$ kürzester Pfad $v_i \rightsquigarrow v_j$.

14.3 Lemma. Falls von v_0 aus ein Zyklus C mit $w(C) < 0$ erreichbar ist, ist $\delta(v_0, v)$ für alle $v \in V$, die vom Zyklus aus erreichbar sind, nicht wohldefiniert. $\to$ Setze $\delta(v_0, v) := -\infty$.

$\Rightarrow$O.B.d.A.: Voraussetzung, dass kürzeste Pfade keine Zyklen enthalten $\to$ Wege $\Rightarrow$ Anzahl der Kanten $\leq |V| - 1$.

Darstellung kürzester Wege:

$\forall v \in V : \pi(v) \in V \cup \{NIL\}$
$G_\pi = (V_\pi, E_\pi)$, $V_\pi = \{v \in V | \, \pi(v) \neq NIL\} \cup \{v_0\}$ Baum,
$E_\pi = \{(\pi(v), v) \in E | \, v \in V_\pi \backslash \{v_0\}\}$ Entfernungsbaum

Relaxation: $\forall v \in V : d(v) \ldots$ Vorläufige Distanz, $d(v) \geq \delta(v_0, v)$

Algorithm 44 INIT(G, v_0)

```
1: for v ∈ V do
2:     d(v) := ∞
3:     π(v) := NIL
4: end for
5: d(v_0) := 0
```

Algorithm 45 RELAX(u, v, w)

```
1: if d(v) > d(u) + w(u, v) then
2:     d(v) := d(u) + w(u, v)
3:     π(v) := u
4: end if
```

Algorithmus: $INIT$ und Folge von $RELAX$-Schritten.

14.4 Lemma. Es gilt $d(v) \geq \delta(v_0, v) \, \forall v \in V$.
Beweis:
Induktion nach Anzahl der $RELAX$-Schritte
IA: $d(v_0) = 0 \geq \delta(v_0, v_0) = 0$, $\infty \geq \delta(v_0, v) \, \forall v \neq v_0$
IV: $\forall x \in V : d(x) \geq \delta(v_0, x)$
IS: $d(v) := d(u) + w(u, v) \overset{IV}{\geq} \delta(v_0, u) + w(u, v) \geq \delta(v_0, v)$ ■

14.5 Lemma. Sei $v_0 \rightsquigarrow u \to v$ ein kürzester Weg in G. Falls $d(u) = \delta(v_0, u)$ zum Zeitpunkt t_0 gilt, dann gilt nach Aufruf von $RELAX(u, v, w)$: $d(v) = \delta(v_0, v)$.

14.6 Korollar. Sei $p = (v_0, \ldots, v_k)$ ein kürzester Weg in G, Relaxationen von $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ in dieser Reihenfolge $\Rightarrow d(v_k) = \delta(v_0, v_k)$.

14.7 Lemma. Sei $G = (V, E)$ ein Graph, $w : E \to \mathbb{R}$, $\nexists$ Zyklen $C : w(C) < 0, v_0 \in V$.
Angenommen: $INIT$ + mehrere Relaxationen auf G liefert $\forall v \in V_\pi : d(v) = \delta(v_0, v)$

$\Rightarrow G_\pi$ ist Baum kürzester Wege mit Wurzel v_0
Beweis:
G_π Baum: Übung
Noch zu zeigen: $p = \{v_0, v_1, \ldots, v_k\}$ in $G_\pi \Rightarrow p$ ist kürzester Weg $v_0 \rightsquigarrow v_k$ in G.
Es gilt: $d(v_i) = \delta(v_0, v_i) \quad \forall i$
$d(v_i) \geq d(v_{i-1}) + w(v_{i-1}, v_i) \Rightarrow w(v_{i-1}, v_i) \leq \delta(v_0, v_i) - \delta(v_0, v_{i-1})$
$\overset{\text{Teleskopsumme}}{\Rightarrow} w(p) \leq \delta(v_0, v_k) \Rightarrow w(p) = \delta(v_0, v_k)$, da $\pi(v_i) = v_{i-1}$ ■

14.2 Algorithmus von Dijkstra

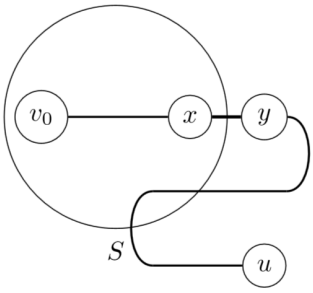
Algorithm 46 DIJKSTRA(G, w, v_0)

```
1: INIT(G, v_0)
2: S := ∅
3: Q := G.V
4: while Q ≠ ∅ do
5:     u := EXTRACT-MIN(Q)
6:     S := S ∪ {u}
7:     for v ∈ Adj[u] do
8:         RELAX(u, v, w)
9:     end for
10: end while
```

Vor jeder Iteration der *while*-Schleife: $Q = V \backslash S$

14.8 Satz. Sei $G = (V, E)$ ein Graph, $w : E \to \mathbb{R}_0^+$, $v_0 \in V \Rightarrow DIJKSTRA$ terminiert mit $d(u) = \delta(v_0, u) \, \forall u \in V$.
Beweis:
Schleifeninvariante: Vor jeder Iteration der *while*-Schleife gilt:
$d(v) = \delta(v_0, v) \, \forall v \in S$
Aufrechterhaltung:
unmittelbar vor der aktuellen Iteration: $RELAX(x, y, w), x \in S, y \in Adj[x]$
Sei $u = EXTRACT\text{-}MIN(Q)$, Annahme: $d(u) > \delta(v_0, u)$
$\Rightarrow$ Der minimale Pfad $v_0 \rightsquigarrow u$ verläuft über $y \notin S$, wobei y als erster Knoten des Pfades außerhalb von S gewählt sei (mit Vorgänger x, vgl. Abbildung 14.2 auf der nächsten Seite).

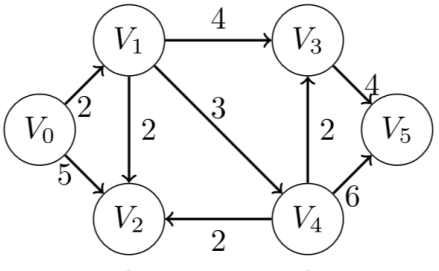
$\Rightarrow \delta(v_0, y) \leq \delta(v_0, u)$
Außerdem wurde die Kante (x, y) zuvor relaxiert $\Rightarrow d(y) = \delta(v_0, y)$
$d(u) > \delta(v_0, u) \geq \delta(v_0, y) = d(y) \lightning$, da $d(u) = \min$! (sonst wäre im vorherigen Schritt y „entfernt“ worden und nicht u)



■

14.9 Korollar. *DIJKSTRA* generiert G_π Entfernungsbaum mit Wurzel v_0.

14.10 Beispiel.



	v_0	v_1	v_2	v_3	v_4	v_5	u
0	0	∞	∞	∞	∞	∞	v_0
1	0	$2/v_0$	$5/v_0$	∞	∞	∞	v_1
2	0	$2/v_0$	$4/v_1$	$6/v_1$	$5/v_1$	∞	v_2
3	0	$2/v_0$	$4/v_1$	$6/v_1$	$5/v_1$	∞	v_4
4	0	$2/v_0$	$4/v_1$	$6/v_1$	$5/v_1$	$11/v_4$	v_3
5	0	$2/v_0$	$4/v_1$	$6/v_1$	$5/v_1$	$10/v_3$	v_5

Laufzeit: Q Liste $\rightarrow \mathcal{O}(|V|^2 + |E|)$
Q Heap $\rightarrow \mathcal{O}((|V| + |E|) \log |V|)$
Q Fibonacci-Heap $\rightarrow \mathcal{O}(|V| \log |V| + |E|)$

14.3 Algorithmus von Bellman-Ford

Vorteil: $w : E \rightarrow \mathbb{R}$, erkennt Zyklen negativen Gewichts
Nachteil: $\mathcal{O}(|V||E|)$

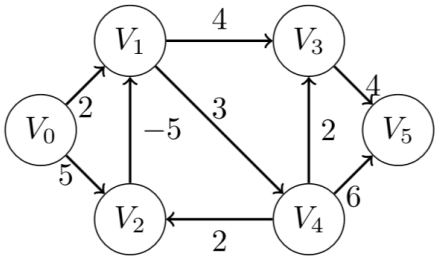
Algorithm 47 BF(G, w, v_0)

```
1: INIT(G, v_0)
2: for i = 1 to |V| − 1 do
3:     for (u, v) ∈ E do
4:         RELAX(u, v, w)
5:     end for
6: end for
7: for (u, v) ∈ E do
8:     if d(v) > d(u) + w(u, v) then
9:         STOP (Zyklen mit negativen Gewicht!)
10:        return FALSE
11:    end if
12: end for
13: return TRUE
```

Korrektheit:

14.11 Lemma. Sei $G = (V, E)$ ein gerichteter Graph, $w : E \rightarrow \mathbb{R}$, $v_0 \in V$. Von v_0 aus seien keine Zyklen C mit $w(C) < 0$ erreichbar. Dann gilt nach $|V| - 1$ Iterationen der for-Schleife $d(v) = \delta(v_0, v)\, \forall v \in V$.
Beweis:
Sei $v \in V$ von v_0 aus erreichbar. $p = (v_0, v_1, \ldots, v_k = v)$ kürzester Weg, $k \leq |V| - 1$
BF relaxiert in jeder Iteration alle Kanten.
$\Rightarrow$ insbesondere: (v_{i-1}, v_i) wird in der i-ten Iteration relaxiert $\Rightarrow$ Kanten von p in gegebener Reihenfolge relaxiert $\overset{14.6}{\Rightarrow} \delta(v_0, v) = d(v)$ ■

14.12 Beispiel.



$Adj[v]/d/\pi$	v_0	v_1	v_2	v_3	v_4	v_5	i
	$0/\text{-}$	$\infty/\text{-}$	$\infty/\text{-}$	$\infty/\text{-}$	$\infty/\text{-}$	$\infty/\text{-}$	
$Adj[v_0]$	$0/\text{-}$	$2/v_0$	$5/v_0$	$\infty/\text{-}$	$\infty/\text{-}$	$\infty/\text{-}$	1
v_1	$0/\text{-}$	$2/v_0$	$5/v_0$	$6/v_1$	$5/v_1$	$\infty/\text{-}$	1
v_2	$0/\text{-}$	$0/v_2$	$5/v_0$	$6/v_1$	$5/v_1$	$\infty/\text{-}$	1
v_3	$0/\text{-}$	$0/v_2$	$5/v_0$	$6/v_1$	$5/v_1$	$10/v_3$	1
v_4	$0/\text{-}$	$0/v_2$	$5/v_0$	$6/v_1$	$5/v_1$	$10/v_3$	1
v_1	$0/\text{-}$	$0/v_2$	$5/v_0$	$4/v_1$	$3/v_1$	$10/v_3$	2
v_3	$0/\text{-}$	$0/v_2$	$5/v_0$	$4/v_1$	$3/v_1$	$8/v_3$	2

14.13 Korollar. $\exists v_0 \rightsquigarrow v \Leftrightarrow BF$ terminiert mit $d(v) < \infty$
Beweis:
$\nexists$ Pfad $\Rightarrow d(v) = \infty$
$\exists$ Pfad $\Rightarrow \exists$ Pfad mit maximal $|V| - 1$ Kanten $\Rightarrow d(v) < \infty$. ■

14.14 Satz. Falls in G keine Zyklen negativen Gewichts vorkommen, die von v_0 aus erreichbar sind, liefert $BF\ TRUE$ und G_π ist ein Entfernungsbaum. Andernfalls liefert $BF\ FALSE$.
Beweis:

1. Fall: $\nexists$ solche Zyklen: Falls $\exists$ Weg $v_0 \rightsquigarrow v \Rightarrow d(v) = \delta(v_0, v)$, andernfalls $d(v) = \infty$
   $\forall (u, v) \in E$ gilt: $d(v) = \delta(v_0, v) \leq \delta(v_0, u) + w(u, v) = d(u) + w(u, v) \Rightarrow TRUE$

2. Fall: $\exists$ solche Zyklen: $C = (v_0, v_1, \ldots, v_k = v_0)$, $w(C) < 0$, $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$.
   Annahme: BF liefert $TRUE$
   $\Rightarrow d(v_i) \leq d(v_{i-1}) + w(v_{i-1}, v_i) \quad \forall 1 \leq i \leq k$
   $\Rightarrow \sum_{i=1}^{k} d(v_i) \leq \sum_{i=1}^{k} d(v_{i-1}) + w(C)$. Summen sind gleich (Zyklus), aber $w(C) < 0$ ↯

■

14.4 Algorithmus von Floyd und Warshall

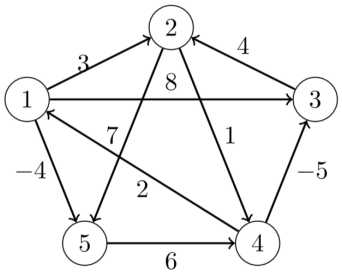
$V = \{v_1, \ldots, v_n\}$, $W = (w_{ij})_{1 \le i,j \le n}$, $w_{ij} = w(v_i, v_j)$
Algorithmus für das Kürzeste-Pfade-Problem für alle Knotenpaare. Benutzt dynamische Programmierung.

Algorithm 48 FW(W)

```
1:  n = W.zeilen
2:  D^(0) = W
3:  for k = 1 to n do
4:      sei D^(k) = (d_ij^(k)) eine neue n × n-Matrix
5:      for i = 1 to n do
6:          for j = 1 to n do
7:              d_ij^(k) := min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8:          end for
9:          if d_ii < 0 then
10:             STOP (Zyklen mit negativen Gewicht!)
11:         end if
12:     end for
13: end for
14: return D^(n)
```

Laufzeit: $\Theta(V^3)$

14.15 Beispiel.



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & NIL & 4 & NIL & NIL \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} NIL & 3 & 4 & 5 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

15 Flüsse

Generalvoraussetzung:
Der gerichteter Graph $G = (V, E)$ ist zusammenhängend und hat keine Zyklen der Länge zwei.

15.1 Definition. Sei $G = (V, E)$ ein gerichteter Graph, $w : E \to \mathbb{R}_0^+$, $s, t \in V$: $d^-(s) = d^+(t) = 0$. Es existiere ein Weg $s \rightsquigarrow t$. s heißt Quelle und t Senke des Netzwerks. $G = (V, E, w, s, t)$ heißt Flussnetzwerk.

15.2 Definition. Sei $G = (V, E)$ ein gerichteter Graph. Der Schatten von G bezeichnet den ungerichteten Graphen $G = (V, E)$.

15.3 Definition. Sei G ein Flussnetzwerk, $\phi : E \to \mathbb{R}_0^+$ heißt Fluss auf $G \Leftrightarrow$

1. Kapazitätsbeschränkung: $\forall e \in E : 0 \leq \phi(e) \leq w(e)$
2. Flusserhaltung: $\forall v \in V \backslash \{s, t\} : \sum_{u \in \Gamma^-(v)} \phi(u, v) = \sum_{w \in \Gamma^+(v)} \phi(v, w)$ (Zufluss = Abfluss)

Wert (Größe) von $\phi : |\phi| := \sum_{v \in \Gamma^+(s)} \phi(s, v) \overset{UE}{=} \sum_{v \in \Gamma^-(t)} \phi(v, t)$.

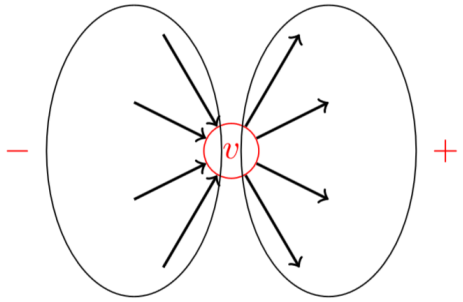
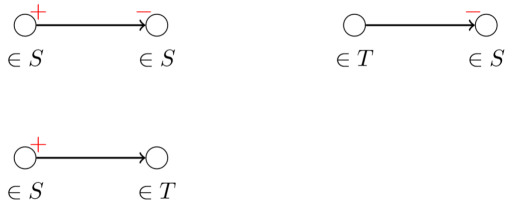


Abbildung 18: Zu- und Abfluss an einem Knoten

Maximales Flussproblem:
Gegeben: G Flussnetzwerk
Gesucht: Maximaler Fluss, d.h. Fluss ϕ, sodass $|\phi| = \max!$

15.4 Definition. Sei G ein Flussnetzwerk, ein Schnitt (S, T) von G ist eine Partition von V, $V = S \cup T, s \in S, t \in T$.

Kapazität von (S,T) : $c(S,T) := \sum_{(x,y):\, x\in S,\, y\in T} w(x,y)$



15.5 Lemma. Sei G ein Flussnetzwerk, ϕ ein Fluss auf G und (S,T) ein Schnitt von $G \Rightarrow |\phi| = \sum_{(x,y):\, x\in S,\, y\in T} \phi(x,y) - \sum_{(x,y):\, x\in T,\, y\in S} \phi(x,y) \leq c(S,T)$

Beweis:

$$|\phi| = \sum_{v\in\Gamma^+(s)} \phi(s,v) =$$
$$\overset{(*)}{=} \sum_{v\in S} \left(\sum_{x\in\Gamma^+(v)} \phi(v,x) - \sum_{y\in\Gamma^-(v)} \phi(y,v) \right)$$

(*) Differenz der inneren Summen ist 0, außer für $v = s$ (Flusserhaltung, außer in der Quelle).
Dies ist aber nur eine andere Schreibweise für die erste Behauptung (Gleichheitszeichen).
Die Abschätzung folgt mithilfe der Beschränktheit des Flusses:
$\forall x,y \in G : \phi(x,y) \leq w(x,y)$

$$\sum_{(x,y):\, x\in S,\, y\in T} \phi(x,y) - \sum_{(x,y):\, x\in T,\, y\in S} \phi(x,y) \leq \sum_{(x,y):\, x\in S,\, y\in T} w(x,y) - \sum_{(x,y):\, x\in T,\, y\in S} \phi(x,y) \leq$$
$$\leq \sum_{(x,y):\, x\in S,\, y\in T} w(x,y) = c(S,T)$$

da $\sum_{(x,y):\, x\in T,\, y\in S} \phi(x,y) \geq 0$

■

15.6 Korollar. Sei ϕ ein maximaler Fluss, d.h. $|\phi|$ ist maximal und (S,T) ein minimaler

Schnitt, d.h. $c(S,T)$ ist minimal. Dann gilt $|\phi| \leq c(S,T)$.

15.7 Definition. Ein augmentierender Pfad P ist eine Kantenfolge $s = v_0, v_1, \ldots, v_k = t$, die einen Pfad im Schatten von G darstellt, und für die gilt:

1. $\phi(e) < w(e)$ auf allen Vorwärtskanten ($e \in E \cap E(P)$)
2. $\phi(e) > 0$ auf allen Rückwärtskanten

15.8 Satz. Seien $G = (V, E, w, s, t)$ ein Flussnetzwerk, ϕ ein Fluss. Dann gilt: ϕ ist maximal $\Leftrightarrow \nexists$ augmentierende Pfade.
Beweis:
„ $\Rightarrow$ “:
Annahme: ϕ maximal und es existiert ein augmentierender Pfad p.

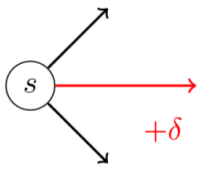
$$\delta' = \min_{e \in E(p) \text{ vorwärts}} (w(e) - \phi(e)) > 0$$

$$\delta'' = \min_{e \in E(p) \text{ rückwärts}} \phi(e) > 0$$

$$\delta = \min(\delta', \delta'') > 0$$

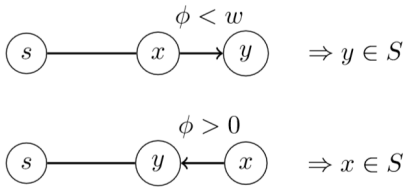
$$\tilde{\phi}(e) := \begin{cases} \phi(e) + \delta & \text{, falls } e \text{ Vorwärtskante von } E(p) \\ \phi(e) - \delta & \text{, falls } e \text{ Rückwärtskante von } E(p) \\ \phi(e) & \text{, sonst} \end{cases}$$

$\tilde{\phi}$ ist wieder ein Fluss.



$|\widetilde{\phi}| = |\phi| + \delta \quad \lightning$

„$\Leftarrow$“:
$S := \{v \in V \mid \exists \text{ augmentierenden Pfad } s \rightsquigarrow v\}$
laut Voraussetzung: $t \notin S$
$(S, T) := (S, V \backslash S)$ ist Schnitt



Behauptung: Alle Kanten (x, y) mit $x \in S$, $y \in T$ sind gesättigt, d.h. $\phi(x, y) = w(x, y)$
Auf allen Kanten (x, y) mit $x \in T$, $y \in S$ keinen Fluss: $\phi(x, y) = 0$
$\overset{\text{Lemma } 15.5}{\Rightarrow}$: $|\phi| = \sum_{(x,y):\, x \in S,\, y \in T} \phi(x, y) - \sum_{(x,y):\, x \in T,\, y \in S} \phi(x, y) = c(S, T) \Rightarrow \phi$ maximal ■

15.9 Satz. Sei G ein Flussnetzwerk, $w(e) \in \mathbb{N}$ oder $w(e) \in \mathbb{Q} \Rightarrow \exists$ maximaler Fluss.

Beweis:
Definiere

$$\phi_0 : \begin{cases} E \to \mathbb{R} \\ e \mapsto 0 \end{cases}$$

$\delta \in \mathbb{N}$ oder $\delta \in \frac{\mathbb{N}}{k}$ endliche, ganzzahlige Bewertung. Nach endlich vielen Schritten erhalten wir damit $\phi_{\max}$.
Stetigkeits-/Kompaktheitsgründe garantieren Existenz eines $\phi_{\max}$. (muss algorithmisch

nicht konvergieren):

$$
\begin{aligned}
&M := \{\phi : E \to \mathbb{R} |\, \phi \text{ Fluss auf } G\} \\
&f : \begin{cases} M \to \mathbb{R} \\ \phi \mapsto |\phi| \end{cases} \\
&d(\phi_1, \phi_2) = \max_{e \in E} |\phi_1(e) - \phi_2(e)| \\
&\forall \epsilon > 0\, \exists \delta > 0 : d(\phi_1, \phi_2) < \delta \Rightarrow \big| |\phi_1| - |\phi_2| \big| < \epsilon \\
&\text{Wähle } \delta < \frac{\epsilon}{n} \\
&(\phi_n)_{n \geq 1}\,,\ \phi_n \to \phi,\ \phi \in M \\
&\sup_{\phi \in M} d(0, \phi) \leq \max_{e \in E} w(e) \\
&\Rightarrow M \text{ beschränkt (als Teilmenge von } \mathbb{R}^k, k = |E|) \Rightarrow M \text{ kompakt}
\end{aligned}
$$

■

15.1 Ford-Fulkerson Algorithmus

Algorithm 49 FORD-FULKERSON(V, E, w, s, t)

```
1: for (u, v) ∈ E do
2:     φ(u, v) = 0
3: end for
4: while PATH(G_φ, s, t) = TRUE do
5:     p = FINDPATH(G_φ, s, t)
6:     C_φ(p) = min{c_φ(u, v) | (u, v) ∈ E(p)}
7:     for (u, v) ∈ E(p) do    // ∃ augmentierender Pfad
8:         if (u, v) ∈ E then
9:             φ(u, v) = φ(u, v) + C_φ(p)
10:        else
11:            φ(u, v) = φ(u, v) − C_φ(p)
12:        end if
13:    end for
14: end while
```

Dabei ist G_ϕ das sogenannte Restnetzwerk. Es gilt:

$$V(G_\phi) = V$$
$$E(G_\phi) = \{(u,v) |\, (u,v) \in E,\, \phi(u,v) < w(u,v) \text{ oder } (v,u) \in E \wedge \phi(v,u) > 0\}$$
$$c_\phi(u,v) = \begin{cases} w(u,v) - \phi(u,v) & \text{, falls } (u,v) \in E(G_\phi) \cap E \\ \phi(u,v) & \text{, falls } (u,v) \in E(G_\phi),\, (v,u) \in E \end{cases}$$

Laufzeit:
$\mathcal{O}\left(|\phi_{\max}|\,(|V| + |E|)\right) = \mathcal{O}\left(\max\limits_{e \in E} w(e)\,(|V| + |E|)\right)$ nicht polynomiell
Finden der augmentierenden Pfade über $BFS \rightarrow$ Edmonds-Karp $\mathcal{O}\left(|V||E|^2\right)$
Algorithmus von Dinic (Pfade mit großem δ suchen) $\rightarrow \mathcal{O}\left(|V|^2|E|\right)$
Algorithmus von Goldberg-Tarjan: $\mathcal{O}\left(|V|^2\sqrt{|E|}\right)$. Arbeitet nicht mit augmentierenden Flüssen, andere Methode (abgewandelte Flüsse, quasi-Flüsse, Push-Relabel-Algorithmus).

16 Die schnelle Fouriertransformation

$$A(x), B(x) \in \mathbb{C}[x] \quad \deg A, \deg B < n, \text{ Gradschranke } n$$
$$A(x) = \sum_{i=0}^{n-1} a_i x^i,\ B(x) = \sum_{i=0}^{n-1} b_i x^i$$
$$A(x) + B(x) \Rightarrow \Theta(n),\ C(x) = A(x) + B(x),\ \deg C < n$$
$$C(x) = A(x)B(x) = \sum_{i=0}^{2n-2} c_i x^i,\ c_i = \sum_{j=0}^{i} a_j b_{i-j},\ \deg C < 2n$$

16.1 Darstellung von Polynomen

Koeffizientendarstellung (KD):

$$A(x) = \sum_{i=0}^{n-1} a_i x^i \mathrel{\hat{=}} (a_0, a_1, \ldots, a_{n-1}) = \underline{a}$$

Horner-Schema:

$$A(x_0) = a_0 + x_0\left(a_1 + x_0(a_2 \ldots x_0(a_{n-2} + x_0 a_{n-1}))\right)$$

Addition und Multiplikation:
$A(x) \mathrel{\hat{=}} \underline{a},\ B(x) \mathrel{\hat{=}} \underline{b} \Rightarrow A(x) + B(x) \mathrel{\hat{=}} \underline{a} + \underline{b}$
$C(x) = A(x)B(x) :\ \underline{c} = \underline{a} * \underline{b}$ (Faltung)

Stützstellendarstellung (SSD):

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\},\ A(x_i) = y_i$$
$$V(x_0, x_1, \ldots, x_n) \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}$$
$$V(x_0, x_1, \ldots, x_n) := \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^{n-1} \end{pmatrix} \quad \text{(Vandermonde-Matrix)}$$

$\det V \neq 0 \Leftrightarrow x_0, \ldots, x_n$ paarweise verschieden
Lineares Gleichungssystem lösen: $\rightarrow \Theta\left(n^3\right)$
schneller: Lagrange-Interpolation:

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)} \rightarrow \Theta\left(n^2\right)$$

16.1 Bemerkung. Seien $x_0, \ldots, x_n$ paarweise verschieden. Dann heißen

$$\prod_{j \neq k} \frac{x - x_j}{x_k - x_j}$$

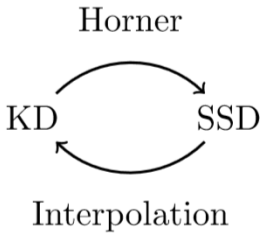
Lagrange-Polynome.

Operationen für SSD:
$C(x_k) = A(x_k) + B(x_k)$ gleiche Stützstellen für SSD von A bzw. $B \rightarrow \Theta(n)$
Multiplikation: $C(x_k) = A(x_k)B(x_k)$, aber $\deg C = \deg A + \deg B < 2n - 1$
$\Rightarrow$ erweiterte SSD von $A(x)$ bzw. $B(x)$

$$\begin{aligned}
&\{(x_0, y_0), \ldots, (x_{2n-1}, y_{2n-1})\} \leftrightarrow A(x) \\
&\{(x_0, y_0'), \ldots, (x_{2n-1}, y_{2n-1}')\} \leftrightarrow B(x) \\
&\Rightarrow C(x) \leftrightarrow \{(x_0, y_0 y_0'), \ldots, (x_{2n-1}, y_{2n-1} y_{2n-1}')\}
\end{aligned}$$

$\rightarrow \Theta(n)$, falls erweiterte SSD von $A(x)$, $B(x)$ gegeben.
Auswerten eines Polynoms in SSD: Konvertieren in KD, dann Auswerten.

Schnelle Multiplikation von Polynomen in KD
schnelle Konversion KD $\leftrightarrow$ SSD nötig



geschickte Wahl der Stützstellen, Auswerten: geschickt mit diskreter Fouriertransformation (DFT), liefert SSD, Interpolation: inverse DFT
DFT und IDFT als Divide & Conquer ⇒ Fast Fourier Transformation (FFT)

Gegeben:

$$A(x) = \sum_{i=0}^{n-1} a_i x^i,\ B(x) = \sum_{i=0}^{n-1} b_i x^i,\ \text{ Gradschranke } n,\ \text{oBdA} : n = 2^k, (\text{mit 0 auffüllen})$$

Vorgangsweise bei der Multiplikation:

(i) Verdoppeln der Gradschranke: $A(x)$, $B(x)$ um n Nullkoeffizienten erweitern

(ii) Auswerten, SSD der Länge $2n$ von $A(x)$, $B(x)$ (FFT)

(iii) Punktweise Multiplikation: $C(x_k) = A(x_k)B(x_k)$

(iv) Interpolation: KD von $C(x)$ (FFT, IDFT)

$w = \sqrt[2n]{1} = e^{\frac{\pi i}{n}}$

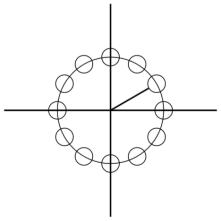
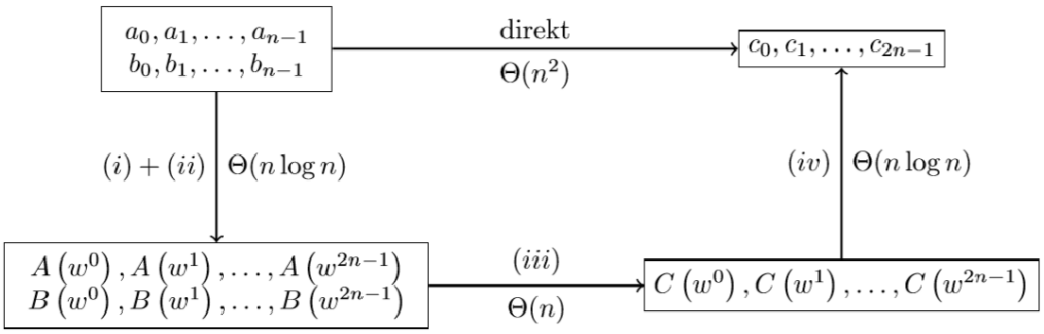


Abbildung 19: n-te Einheitswurzel



16.2 DFT und FFT

Sei $\zeta_n = e^{\frac{2\pi i}{n}} \Rightarrow (\{\zeta_n^0, \zeta_n^1, \ldots, \zeta_n^{n-1}\}, \cdot) \cong (\mathbb{Z}_n, +)$

16.2 Lemma.

1. $\zeta_{dn}^{dk} = \zeta_n^k$

2. $\zeta_{2n}^2 = \zeta_n$

3. $\sum_{j=0}^{n-1} \left(\zeta_n^k\right)^j = \frac{\left(\zeta_n^k\right)^n - 1}{\zeta_n^k - 1} = 0, \quad k \not\equiv 0 \mod n$

DFT: $A(x) = \sum_{j=0}^{n-1} a_j x^j$, $w := \zeta_n$, $y_k := A(w^k) = \sum_{j=0}^{n-1} a_j w^{kj}$
$\underline{y} = (y_0, \ldots, y_{n-1})$ ist DFT von $\underline{a} = (a_0, \ldots, a_{n-1})$
$\underline{y} = DFT_n(\underline{a}) \Leftrightarrow \underline{y} = V(1, w, \ldots, w^{n-1}) \cdot \underline{a}$
Sei $n = 2^k$

$$\begin{aligned} A_0(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{\frac{n}{2}-1} \\ A_1(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{\frac{n}{2}-1} \\ A(x) &= A_0(x^2) + x A_1(x^2) \end{aligned}$$

Man muss also $A_i(w^{2j})$ ausrechnen.
w^{sj} sind $\frac{n}{2}$ Wurzeln $\sqrt[\frac{n}{2}]{1}$, jede tritt genau zweimal auf $\Rightarrow$ Berechnung von DFT_n in 2 $DFT_{\frac{n}{2}}$-Berechnungen aufteilen.

Algorithm 50 REKURSIVE-FFT(a)

```
1: n := |a|   // n = 2^k
2: if n = 1 then
3:     return a
4: end if
5: w := e^{2πi/n}
6: w' := 1
7: a_0 := (a_0, a_2 ..., a_{n-2})
8: a_1 := (a_1, a_3 ..., a_{n-1})
9: y_0 := RECURSIVE-FFT(a_0)
10: y_1 := RECURSIVE-FFT(a_1)
11: for k = 0 to n/2 - 1 do
12:     y_k := y_{0,k} + w' y_{1,k}
13:     y_{k+n/2} := y_{0,k} - w' y_{1,k}
14:     w' := w' w
15: end for
16: return y
```

Korrektheit:
Per Induktion:
$n = 1 : y_0 = a_0 w^0 = a_0$

Sei w' das aktuelle w^k: $w' = w^k$ in der for-Schleife
Ziel: $y_k = A(w^k) = A_0(w^{2k}) + w^k A_1(w^{2k})$

$$y_{0,k} = A_0(w^{2k}) = A_0(\zeta_{\frac{n}{2}}^k),\ 0 \le k \le \frac{n}{2} - 1$$
$$y_{1,k} = A_1(\zeta_{\frac{n}{2}}^k),\ 0 \le k \le \frac{n}{2} - 1$$
$$y_k = A_0(w^{2k}) + w^k A_1(w^{2k}) = A(w^k),\ 0 \le k \le \frac{n}{2} - 1$$
$$y_{k+\frac{n}{2}} = A_0(w^{2k}) - w^k A_1(w^{2k}) = A_0(w^{2k+n}) + w^{k+\frac{n}{2}} A_1(w^{2k+n})$$
$$\text{da } w^{2k} = w^{2k+n}, -w^k = w^{k+\frac{n}{2}} \Leftrightarrow -1 = w^{\frac{n}{2}}$$
$$\Rightarrow \underline{y} = DFT_n(\underline{a})$$

$$T(n) = 2T(\tfrac{n}{2}) + \Theta(n) \overset{\text{Master-Theorem}}{\Rightarrow} T(n) = \Theta(n \log n)$$

IDFT:
Sei $\underline{y} = DFT_n(\underline{a})$, $V_n = V(1, w, w^2, \ldots, w^{n-1})$
$\underline{y} = V_n\underline{a} \Rightarrow \underline{a} = V_n^{-1}\underline{y}$, $V_n^{-1} = \frac{1}{n}V(1, w^{-1}, w^{-2}, \ldots, w^{-n+1}) =: V_n'$
Beweis:
$(V_n' \cdot V_n)_{j,k} = \frac{1}{n}\sum_{j=0}^{n-1} w^{-lj}w^{lk} = \frac{1}{n}\sum_{j=0}^{n-1}\left(w^{k-j}\right)^l \quad -(n+1) \leq k-j \leq n-1$

1. $k = j \Rightarrow w^{k-j} = 1 \Rightarrow (V_n' \cdot V_n)_{j,j} = 1$
2. $k \neq j \Rightarrow w^{k-j} \neq 1 \Rightarrow (V_n' \cdot V_n)_{j,k} = 0$

$V_n' = \frac{1}{n}\overline{V_n} = \frac{1}{n}V_n^*$

Algorithmus für IDFT:

1. In FFT $\underline{a} \leftrightarrow \underline{y}$
2. $w \to w^{-1} = \overline{w}$
3. Am Ende: Ergebnis $\cdot\frac{1}{n}$

$DFT_n^{-1} \to \Theta(n \log n)$

16.3 Satz. Seien $\underline{a}, \underline{b} \in \mathbb{C}^n$, $n = 2^k \Rightarrow \underline{a} * \underline{b} = DFT_{2n}^{-1}\left(DFT_{2n}(\underline{a'}) \cdot DFT_{2n}(\underline{b'})\right)$, wobei $\underline{a'} = (\underline{a}, \underline{0}) \in \mathbb{C}^{2n}$, $\underline{b'} = (\underline{b}, \underline{0}) \in \mathbb{C}^{2n}$, und $\cdot$ ist als komponentenweises Produkt zu verstehen.

17 Geometrische Algorithmen

Algorithmische Geometrie in der Ebene ($\mathbb{R}^2$)
Eingabe: Punktmenge Q, Punkte $p_i = \begin{pmatrix} x_i \\ y_i \end{pmatrix}$, $x_i, y_i \in \mathbb{R}$

17.1 Eigenschaften von Strecken

$p_1 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}, p_2 = \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}, p_3 = \alpha p_1 + (1-\alpha)p_2$, $0 \leq \alpha \leq 1$, konvexe Linearkombination.
$\overline{p_1p_2} = \{\alpha p_1 + (1-\alpha)p_2 |\, 0 \leq \alpha \leq 1\}$, $\vec{p_1p_2}$: gerichtete Strecke
$p_1 \otimes p_2 = \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} = x_1y_2 - x_2y_1 = -p_2 \otimes p_1 = \pm||p_1 \times p_2|| \ldots$ vorzeichenbehafteter

Flächeninhalt des von $\overline{\begin{pmatrix}0\\0\end{pmatrix} p_1}, \overline{\begin{pmatrix}0\\0\end{pmatrix} p_2}$ aufgespannten Parallelogramms

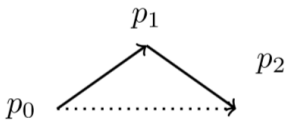
Lage von $\vec{p_0 p_1}$ zu $\vec{p_0 p_2}$

(I) (II)

Wähle p_0 als Ursprung:

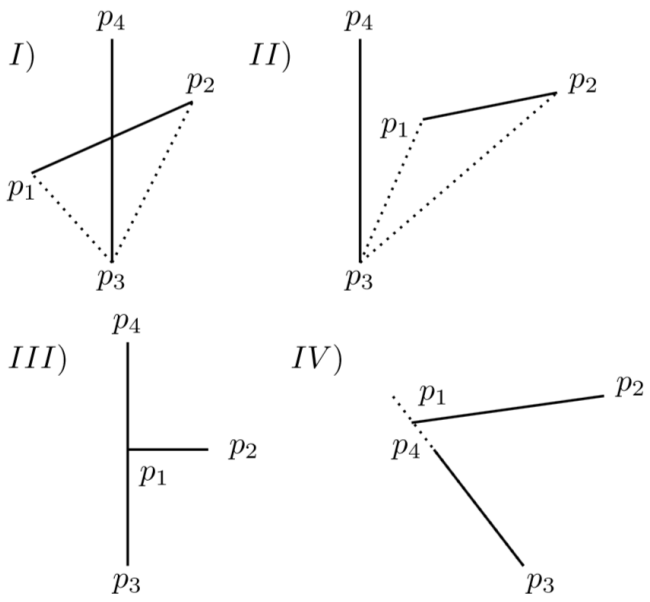
$$(p_1 - p_0) \otimes (p_2 - p_0) \begin{cases} > 0 & (I) \\ < 0 & (II) \\ = 0 & \varphi = 0 \text{ oder } \varphi = \pi \end{cases}$$

Aufeinanderfolgende Strecken: Links- oder Rechtskurve? $\measuredangle p_0 p_1 p_2$



$$(p_1 - p_0) \otimes (p_2 - p_0) \begin{cases} > 0 & \rightarrow \text{ Linkskurve} \\ < 0 & \rightarrow \text{ Rechtskurve} \\ = 0 & \rightarrow \text{ weder noch (auf einer Linie)} \end{cases}$$

Wann schneiden sich die Strecken $\overline{p_1 p_2}$ und $\overline{p_3 p_4}$?



1. $sgn(p_4 - p_3) \otimes (p_2 - p_3) \neq sgn(p_4 - p_3) \otimes (p_1 - p_3)$

2. $sgn(p_4 - p_3) \otimes (p_2 - p_3) = sgn(p_4 - p_3) \otimes (p_1 - p_3)$
   $< 0 \rightarrow$ rechts von $\overline{p_3 p_4}$, $> 0 \rightarrow$ links

3. $sgn(p_4 - p_3) \otimes (p_1 - p_3) = 0$
   $(iii) \Leftrightarrow \min(x_3, x_4) \leq x_1 \leq \max(x_3, x_4) \wedge \min(y_3, y_4) \leq y_1 \leq \max(y_3, y_4)$

Algorithm 51 DIRECTION(p, q, r)

```
1: return (r − q) ⊗ (q − p)
```

Algorithm 52 ON-SEGMENT(p, q, r)

```
1: if min(x_p, x_q) ≤ x_r ≤ max(x_p, x_q) ∧ min(y_p, y_q) ≤ y_r ≤ max(y_p, y_q) then
2:     return TRUE
3: else
4:     return FALSE
5: end if
```


Algorithm 53 SEGMENT-INTERSECT(p_1, p_2, p_3, p_4)

1: $d_1 = DIRECTION(p_3, p_4, p_1)$
2: $d_2 = DIRECTION(p_3, p_4, p_2)$
3: $d_3 = DIRECTION(p_1, p_2, p_3)$
4: $d_4 = DIRECTION(p_1, p_2, p_4)$
5: **if** $d_1 d_2 < 0 \wedge d_3 d_4 < 0$ **then return** TRUE
6: **else if** $d_1 = 0 \wedge ON\text{-}SEGMENT(p_3, p_4, p_1)$ **then return** TRUE
7: **else if** $d_2 = 0 \wedge ON\text{-}SEGMENT(p_3, p_4, p_2)$ **then return** TRUE
8: **else if** $d_3 = 0 \wedge ON\text{-}SEGMENT(p_1, p_2, p_3)$ **then return** TRUE
9: **else if** $d_4 = 0 \wedge ON\text{-}SEGMENT(p_1, p_2, p_4)$ **then return** TRUE
10: **else**
11: **return** FALSE
12: **end if**

17.2 Bestimmen der konvexen Hülle

Gegeben: Punktmenge Q
Gesucht: Konvexe Hülle $[Q]$

Es genügt: konvexes Polygon $P = \{p_0, p_1, \ldots, p_n\}$, sodass für alle $q \in Q : q = \sum_{i=1}^{n} \lambda_i p_i$, $\lambda_i \geq 0$, $\sum_{i=1}^{n} \lambda_i = 1$

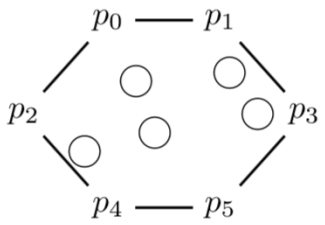


Abbildung 20: Konvexe Hülle

Voraussetzung: $\exists q_1, q_2, q_3 \in Q$ nicht kollinear, P genau die Extremalpunkte von $[Q]$

Grahamsches Scannen:
verwaltet Stack S aus Kandidaten x für P.

1. Alle $x \in Q$ kommen im Laufe des Algorithmus auf den Stack.
2. Alle $x \in Q \backslash P$ werden schließlich entfernt und kommen nie wieder auf den Stack
3. Wenn der Algorithmus terminiert, dann S „=“P, geordnet in mathematisch positiver Richtung, gelesen von unten nach oben, i.Z. $[S \sim P]$

Hilfefunktionen:
$TOP(S) \ldots$ Ausgabe des obersten Elements von S
$NEXT\text{-}TO\text{-}TOP(S) \ldots$ Ausgabe des zweitobersten Elements von S
$POP(S) \ldots$ Entfernen des obersten Elements
$PUSH(S, x) \ldots$ Legen von x auf den Stack

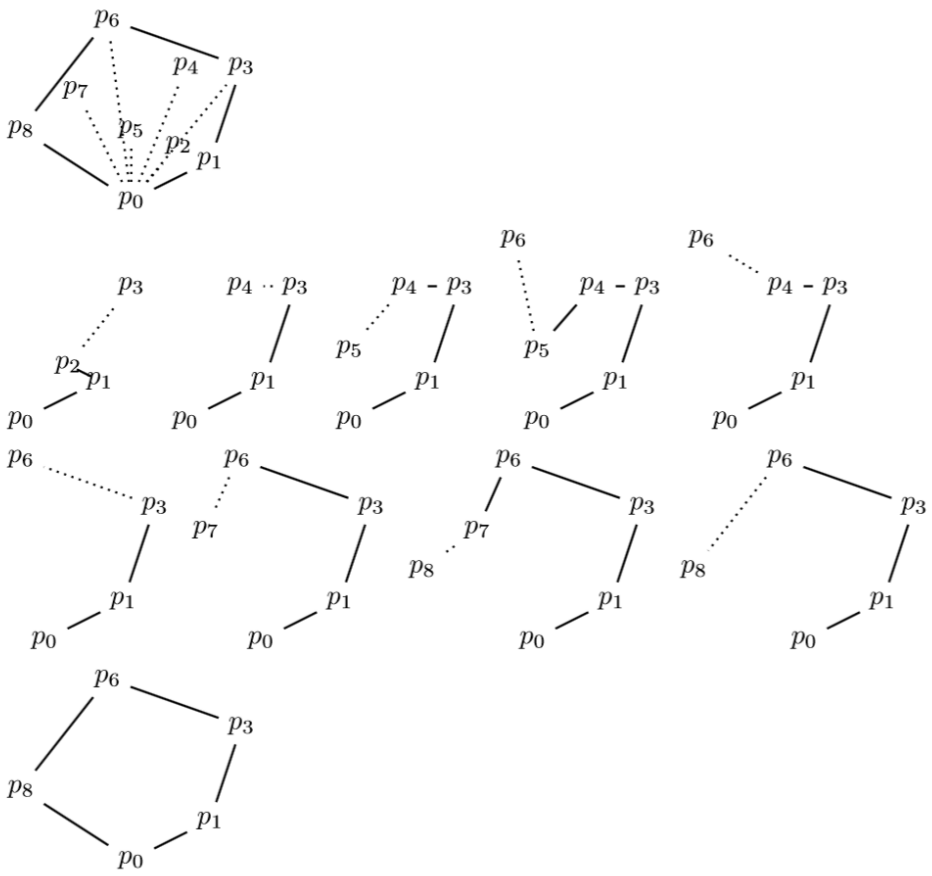
Algorithm 54 GRAHAM-SCAN(Q)

1: p_0 := Punkt in Q mit kleinster y-Koordinate, von all diesen jener mit kleinster x-Koordinate.
2: $(p_1, p_2, \ldots, p_m)$:= Punkte von Q, geordnet bzgl. Polarwinkel, genauer bzgl. Winkel zwischen $\overrightarrow{p_0 p_i}$ mit $\begin{pmatrix}1\\0\end{pmatrix} =: \varphi(p_i)$, wobei im Falle von mehreren Punkten mit gleichem Polarwinkel nur jener genommen wird, der zu p_0 maximalen Abstand hat
3: **if** $m < 2$ **then**
4: **return** Q
5: **else**
6: S := leerer Stack
7: $PUSH(S, p_0)$, $PUSH(S, p_1)$, $PUSH(S, p_2)$
8: **for** $i = 3$ to m **do**
9: **while** $NEXT\text{-}TO\text{-}TOP(S) \to TOP(S) \to p_i$ keine Linkskurve **do**
10: $POP(S)$
11: **end while**
12: $PUSH(S, p_i)$
13: **end for**
14: **return** S
15: **end if**

Laufzeit:
$n = |Q|$
Sortieren nach $\varphi(p_i)$: $\mathcal{O}(n \log n)$
$PUSH, POP, \ldots \mathcal{O}(1)$

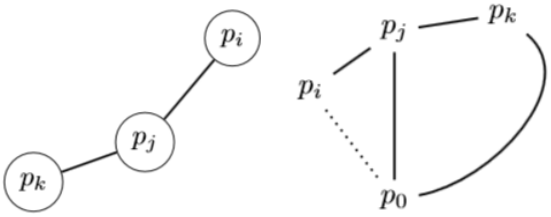
$m + 1 = n$ mal $PUSH$, höchstens n mal $POP \to \mathcal{O}(n)$
$\Rightarrow \mathcal{O}(n \log n)$

17.1 Beispiel.

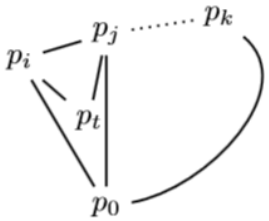


17.2 Satz. Das Grahamsche Scannen, angewendet auf eine Punktmenge Q, liefert schließlich einen Stack S mit $S \sim E$, E Extremalpunkte von Q, sodass $p_0, \ldots, p_m$ die Extremalpunkte von Q in mathematisch positiver Richtung, also gegen den Uhrzeigersinn, sind.
Beweis:
$\{p_0, p_1, \ldots, p_m\} \ldots$ nach Schritt 2 des Algorithmus.
Sei $Q_i := \{p_0, p_1, \ldots, p_i\}$
$Q \backslash Q_m$ a priori entfernte Punkte $\Rightarrow [Q_m] = [Q]$

$Q = \{p_0, p_1, \ldots, p_m\}$
E_i sei die Menge der Extremalpunkte von $[Q_i]$. Es gilt $p_0, p_1, p_i \in E_i$.
Zu zeigen: $S \sim E_m$
Schleifeninvariante: „Am Beginn jeder Iteration der for-Schleife gilt: $S \sim E_{i-1}$“
Initialisierung: $i = 3$: $S = \begin{bmatrix} p_2 \\ p_1 \\ p_0 \end{bmatrix}$, $Q_2 = \{p_0, p_1, p_2\} = E_2 \Rightarrow S \sim E_2$
Aufrechterhaltung: Am Beginn der Iteration zum Wert i: $TOP(S) = p_{i-1}$. Sei $p_j := TOP(S)$ nach Ausführen der $while$-Schleife und $p_k := NEXT\text{-}TO\text{-}TOP(S)$.
$\Rightarrow$ S sieht aus wie unmittelbar nach Ausführen der j-ten Iteration der for-Schleife $(j < i)$
$\overset{\text{IV}}{\Rightarrow} S \sim E_j$ Weiters gilt $\varphi(p_i) > \varphi(p_j)$ und $p_k \to p_j \to p_i$ ist Linkskurve, da sonst p_j entfernt worden wäre.



Nun füge p_i mit $PUSH$ hinzu.
Behauptung: $[Q_j \cup \{p_i\}] = [Q_i]$
Sei p_t ein beliebiger Punkt der durch die $while$-Schleife aus dem Stack S entfernt wurde und sei p_r der nächste Punkt unter p_t in S ($p_r = p_j$ möglich) $\Rightarrow p_r \to p_t \to p_i$ ist keine Linkskurve, sowie $\varphi(p_t) > \varphi(p_r)$.
$\Rightarrow p_t$ liegt im Dreieck $p_0 p_r p_i$ (innen oder auf $\overline{p_r p_i}$) $\Rightarrow p_t \notin E_i \to [Q_i \backslash \{p_t\}] = [Q_i]$
$P_i := \{x \in \{p_0, \ldots, p_m\} |$ eine $POP(X)$-Anweisung in $while$-Schleife während for zum Wert i durchgeführt$\}$ = die Menge aller Punkte, die in der aktuellen Iteration vom Stack entfernt wurden.
$\Rightarrow [Q_i \backslash P_i] = [Q_i]$, da obige Begründung für alle $x \in P_i$ gilt.



$[E_j] = [Q_j] \Rightarrow [Q_i] = [Q_i \backslash P_i] = [Q_j \cup \{p_i\}] = [E_j \cup \{p_i\}] = E_i \Rightarrow S \sim E_i$
Terminierung: $i = m + 1$
$\Rightarrow S \sim E_m$ ■

Algorithm 55 JARVIS-MARCH(Q)

```
1: x := Punkt in Q = {q_0, ..., q_n} mit kleinster y-Koordinate, von all diesen jener mit
   kleinster x-Koordinate
2: y := dummy point
3: p_0 := x
4: i := 0
5: while y ≠ p_0 do
6:     p_i := x
7:     y := q_0
8:     for j = 1 to |Q| − 1 do
9:         if (y = x) ∨ (p_i → y → q_j keine Linkskurve) then
10:            y := q_j
11:        end if
12:        i := i + 1
13:        x := y
14:    end for
15: end while
```

Laufzeit: $\mathcal{O}(n|E_m|)$

17.3 Bemerkung. Jarvis-March verwendet die Technik des Package-Wrapping (Gift-Wrapping, Geschenkeinpacken). man kann sich dies folgendermaßen Vorstellen: Man umhüllt die Menge beginnend bei x mit einem straffen Papierstück bis man wieder beim Ausgangspunkt angekommen ist.

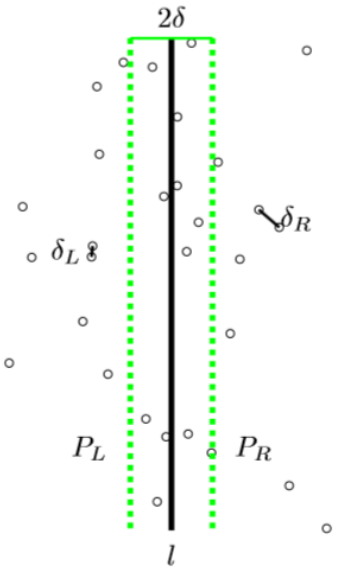
17.3 Das Nächste-Punktepaar-Problem

Gegeben: Punktmenge Q, $|Q| = n \geq 2$
Gesucht: $p_1, p_2 \in Q : ||p_1 - p_2||_2 = \min\limits_{x,y \in Q} ||x - y||_2$

Bruteforce: $\binom{n}{2}$ Punktepaare $\rightarrow \Theta(n^2)$
Divide & Conquer: $T(n) = \Theta(n \log n)$

Eingabe: $P \subseteq Q$, Felder X, Y, $X \ldots$ sortiert nach den x-Koordinaten, $Y \ldots$ sortiert nach y-Koordinaten.
Eingabe bereits sortiert, sonst $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n \log n) \overset{\text{Master-Theorem}}{\Rightarrow} T(n) = \Theta\left(n(\log n)^2\right)$
Dann: Prüfe, ob $|P| \leq 3$. Ja: Bruteforce, Nein: Divide & Conquer

Teile: bestimme vertikale Gerade l, sodass $P = P_L \cup P_R$ $|P_L| = \left\lceil \frac{|P|}{2} \right\rceil$, $|P_R| = \left\lfloor \frac{|P|}{2} \right\rfloor$
$\rightarrow X = X_L \cup X_R$, $Y = Y_L \cup Y_R$ X_L, X_R, Y_L, Y_R sortiert
Conquer: rekursiv auf P_L, P_R anwenden $\rightarrow$ minimale Distanz δ_L, δ_R, $\delta := \min(\delta_L, \delta_R)$
Kombinieren: gesuchte (p_1, p_2) ist entweder jedes mit $||p_1 - p_2||_2 = \delta$ oder $p_1 \in P_L$, $p_2 \in P_R$

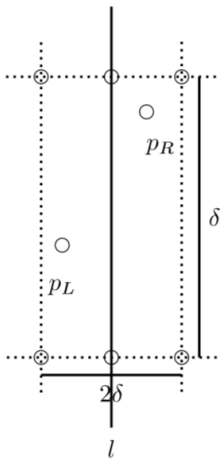


1. $Y' = Y \backslash \{x \in Y \text{ mit } d(x, l) > \delta\}$

2. $p \in Y'$, suche $x \in Y'$ mit $||x - p||_2 < \delta$ durch Vergleich von p mit den 7 nächsten nachfolgenden Punkten aus Y'

3. $\delta' = \min_{x_1, x_2 \in Y'} ||x_1 - x_2||_2$, Punktepaar (p_3, p_4) : $\quad \delta' < \delta$? Ja: *return* (p_3, p_4, δ'), Nein: *return* (p_1, p_2, δ)

Korrektheit: klar, außer: 7?
Annahme $\delta' < \delta$, $\delta' = ||p_L - p_R||$.
Wenn p_L und p_R weniger als δ voneinander entfernt sind, dann müssen sie sich in einem zentriertem $\delta \times 2\delta$ Rechteck um die Gerade l befinden. Es können sich aber höchstens nur 8 Punkte aus P innerhalb dieses Rechtecks befinden. Betrachtet man das linke $\delta \times \delta$ Rechteck, so müssen alle Punkte von P_L mindestens δ von p_L entfernt sein. Damit können höchstens 4 Punkte in dem Rechteck existieren.
Wenn man stattdessen ein $(\delta - \epsilon) \times 2(\delta - \epsilon)$ Rechteck betrachtet, reichen sogar 5 Punkte aus.



Implementierung und Laufzeit:
Hauptproblem: X_L, X_R, Y_L, Y_R, Y' sortiert
Lösung: X, Y vorsortiert $\rightarrow \mathcal{O}(n \log n)$
Umgekehrtes Mergesort Y in zwei sortierte Listen zerlegen:

```
1: Seien Y_L[1...Y.size] und Y_R[1...Y.size] neue Felder
2: Y_L.size = Y_R.size = 0
3: for i = 1 to Y.size do
4:     if Y[i] ∈ P_L then
5:         Y_L.size = Y_L.size + 1
6:         Y_L[Y_L.size] = Y[i]
7:     else
8:         Y_R.size = Y_R.size + 1
9:         Y_R[Y_R.size] = Y[i]
10:    end if
11: end for
```

$\Rightarrow T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n) \rightarrow T(n) = \Theta(n \log n)$

18 Lineare Programmierung

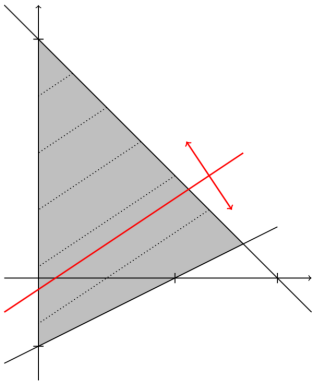
18.1 Allgemeine lineare Programme in Standard- und Schlupfform

Aufgabe:
Gegeben: (affin-) lineare Zielfunktion $f(x_1, x_2, \ldots, x_n) = a_1x_1 + \cdots + a_nx_n(+\nu)$, $a_i \in \mathbb{R}$
Nebenbedingungen: $g_i(x_1, \ldots, x_n) \leq b_i$, $i = 1, \ldots, m$.
Gesucht: $x_1, \ldots, x_n : f(x_1, \ldots, x_n)$ max! min!

18.1 Bemerkung. Es geht auch $g_i(x_1, \ldots, x_n) \geq b_i$ oder $g_i(x_1, \ldots, x_n) = b_i$.

18.2 Beispiel.

$$
\begin{aligned}
&f(, x_1, x_2) = 2x_1 - 3x_2 \text{ max!} \\
&x_1 + x_2 \leq 7 \\
&x_1 - 2x_2 \leq 4 \\
&x_1 \geq 0
\end{aligned}
$$



Graphisch: $f(x_1, x_2) = 2x_1 - 3x_2 = z$ Geraden mit Anstieg $\frac{2}{3}$

18.3 Definition. Jede Wahl $(x_1, \ldots, x_n)$, sodass alle Nebenbedingungen erfüllt sind, heißt zulässige Lösung des linearen Programms.

Im Bsp: Gebiet B beschränkt $\Rightarrow \exists \max_{(x_1,x_2)\in B} f(x_1, x_2) \to g : 2x_1 - 3x_2 = z$ verschieben bis Rand von B erreicht ist.

Allgemein: n Variablen
lineare Ungleichung $\hat{=}$ Halbraum

18.4 Definition. Der Simplex ist der zulässige Bereich, welcher durch den Schnitt der Halbräume gegeben ist.

Standardform:

$$\sum_{j=1}^{n} c_j x_j \text{ max!}$$

Nebenbedingungen:

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i,\ 1 \leq i \leq m$$
$$x_j \geq 0,\ 1 \leq j \leq n$$

In Vektorform:

$$f = \underline{c}^T \cdot \underline{x} \text{ max!}$$
$$A \cdot \underline{x} \geq \underline{b},\ \underline{x} \geq 0 \text{ (Komponentenweise)}$$

Simplex-Algorithmus
Eingabe: lineares Programm
Ausgabe: optimale Lösung

1. Beginne bei einer Ecke $\underline{x}_0$ des Simplex
2. Gehe zu benachbarter Ecke $\underline{x}_1$ (entlang Kante), sodass $f(\underline{x}_0) > f(\underline{x}_0)$
3. Terminiere, wenn $\underline{x}_i =$ lokales Maximum

lokales Maximum = globales Maximum, da f linear und B konvex.

Schlupfform:
In der Schlupfform werden alle Ungleichungen zu Gleichungen, außer die Nichtnegativitätsbedingungen.

$$x_i \geq 0$$
$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad 1 \leq j \leq m$$
$$\text{Schlupf: } s_i = b_i - \sum_{j=1}^{n} a_{ij} x_j \geq 0$$
$$\nu + \sum_{j=1}^{n} c_j x_j \text{ max!}$$
$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j,\ 1 \leq i \leq m$$
$$x_1, \ldots, x_n, x_{n+1}, \ldots, x_{n+m} \geq 0$$
$x_1, \ldots, x_n$ freie Variablen (auch Nichtbasisvariablen genannt)
$x_{n+1}, \ldots, x_{m+n}$ Schlupf- oder Basisvariablen genannt

Konversion in Standardform:

1. $\left.\begin{array}{c}\min \to \max \\ \geq \to \leq \\ \leq \to \geq\end{array}\right\} \to \cdot(-1)$

2. $x_j \geq 0$ fehlt: $x_j = x_j' - x_j''$, $x_j' \geq 0, x_j'' \geq 0 \to$ lineares Programm P'
   $\underline{x}'$ zulässige Lösung von $P' \Rightarrow x_j := x_j' - x_j'' \rightsquigarrow \underline{x}$ zulässige Lösung von P
   $\underline{x}$ zulässige Lösung von $P \Rightarrow x_j' := \underline{x}_j$, $x_j'' := 0 \to \underline{x}'$ zulässige Lösung von P für $\underline{x}_j \geq 0$, ansonsten wähle $x_j' := 0$, $x_j'' := -\underline{x}_j$ für $\underline{x}_j < 0$

3. Nebenbedingung $g(x_1, \ldots, x_n) = b \Leftrightarrow -g(x_1, \ldots, x_n) \leq -b \wedge g(x_1, \ldots, x_n) \leq b$

18.2 Probleme als lineare Programme

1. **Kürzeste Wege**:
   Gegeben: $G = (V, E)$ gerichtet, $w : E \to \mathbb{R}_0^+$, $s, t \in V$.
   Gesucht: (kürzester Weg $s \to t$), $\delta(s, t)$
   lineares Programm:
   d_t max! (siehe Bemerkung 18.5)
   $d_v \leq d_u + w(u, v) \quad \forall (u, v) \in E$
   $d_s = 0$
   $|V|$ Variablen, $|E| + 1$ Nebenbedingungen

   18.5 Bemerkung. Anstatt zu minimieren, maximieren wir.
   $\overline{d_v} := \delta(s, v) \quad \forall v \in V$
   Dann gilt: $\overline{d_v} \leq \overline{d_u} + w(u, v), \overline{d_s} = 0$
   $\overline{d_v} = \min\limits_{u:(u,v)\in E} \left(\overline{d_u} + w(u, v)\right)$
   d.h.: $\overline{d_v} = \max\{x \in \mathbb{R} \mid x \leq \overline{d_u} + w(u, v) \quad \forall (u, v) \in E\}$

2. **Maximaler Fluss**:
   Gegeben: Flussnetzwerk $G = (V, E, w, s, t)$
   Gesucht: $|\phi| = \sum\limits_{v\in\Gamma^+(s)} \phi(s, v)$ max!
   Nebenbedingungen:

$$\begin{aligned} &\phi(u, v) \leq w(u, v) \quad \forall (u, v) \in E \\ &\phi(u, v) \geq 0 \quad \forall (u, v) \in E \\ &\sum_{v\in\Gamma^-(u)} \phi(v, u) = \sum_{v\in\Gamma^+(u)} \phi(u, v) \quad \forall u \in V \backslash \{s, t\} \end{aligned}$$

$|E|$ Variablen, $2|E| + |V| - 2$ Nebenbedingungen

3. **Billigster Fluss**:
   Gegeben: Flussnetzwerk $G = (V, E, w, s, t)$, Kostenfunktion $k : E \to \mathbb{R}$
   Lineares Programm:

$$\sum_{(u,v)\in E} k(u,v)\phi(u,v) \text{ min!}$$

   Nebenbedingungen:

$$\begin{aligned}
&\phi(u,v) \leq w(u,v)\\
&\phi(u,v) \geq 0\\
&\sum_{v\in\Gamma^-(u)} \phi(v,u) = \sum_{v\in\Gamma^+(u)} \phi(u,v)\\
&\sum_{v\in\Gamma^+(s)} \phi(s,v) = d
\end{aligned}$$

   Eine Nebenbedingung mehr als vorher.

18.3 Der Simplex-Algorithmus

(i) Jeder Iteration wird eine Basislösung zugeordnet: Freie Variablen: $x_1 = x_2 = \cdots = x_n = 0$, Schlupfvariable aus den Gleichungen der Nebenbedingung bestimmen

(ii) Jede Iteration konvertiert Schlupfform in eine äquivalente Schlupfform, sodass Zielfunktionswert der neuen Schlupfform (an der Basislösung) größer ist als f (Basislösung der alten Schlupfform)

   a) Wähle freie Variable x, sodass mit wachsendem x f wächst. x wächst bis für eine Schlupfvariable y gilt: $y = 0$

   b) Rollen von x und y vertauschen (entspricht Basiswechsel)

18.6 Beispiel.

$$3x_1 + x_2 + 2x_3 \text{ max!}$$

Standardform:

$$\begin{aligned}
&x_1 + x_2 + 3x_3 \leq 30\\
&2x_1 + 2x_2 + 5x_3 \leq 24\\
&4x_1 + x_2 + 2x_3 \leq 36\\
&x_1,\, x_2,\, x_3 \geq 0
\end{aligned}$$

Schlupfform:

$$\begin{aligned}
z &= \quad 3x_1 + x_2 + 2x_3\\
x_4 &= 30 - x_1 - x_2 - 3x_3\\
x_5 &= 24 - 2x_1 - 2x_2 - 5x_3\\
x_6 &= 36 - 4x_1 - x_2 - 2x_3\\
&x_1,\, x_2,\, x_3,\, x_4,\, x_5,\, x_6 \geq 0
\end{aligned}$$

In kompakter Form:

z	x_4	x_5	x_6		x_1	x_2	x_3
1	0	0	0	0	3	1	2
0	1	0	0	30	-1	-1	-3
0	0	1	0	24	-2	-2	-5
0	0	0	1	36	-4	-1	-2

Start: Basislösung: $\underline{x} = (0, 0, 0, 30, 24, 36)$, $f(\underline{x}) = 0$
Hier ist die Basislösung eine zulässige Lösung, was nicht immer der Fall sein muss.
Nun wählen wir ein x_e mit einem positiven Koeffizienten in der Zielfunktion.
1. Iteration: Wähle x_1
Nebenbedingungen: $x_1 \leq 30$, $2x_1 \leq 24$, $4x_1 \leq 36 \rightarrow x_1 = 9$, $x_6 = 0$
Letzte Gleichung nach x_1 auflösen $\rightarrow x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}$
Einsetzen in die restlichen Gleichungen ergibt:

z	x_1	x_4	x_5		x_2	x_3	x_6
1	0	0	0	27	$\frac{1}{4}$	$\frac{1}{2}$	$-\frac{3}{4}$
0	1	0	0	9	$-\frac{1}{4}$	$-\frac{1}{2}$	$-\frac{1}{4}$
0	0	1	0	21	$-\frac{3}{4}$	$-\frac{5}{2}$	$\frac{1}{4}$
0	0	0	1	6	$-\frac{3}{2}$	-4	$\frac{1}{2}$

Basislösung: $\underline{x'} = (9, 0, 0, 21, 6, 0)$, $f(\underline{x'}) = z = 27$
2. Iteration: Wir können jetztx_2 oder x_3 wählen, aber nicht x_6. Wähle x_2.
Nebenbedingungen: $x_2 \leq 4$, $x_5 = 0$
$x_2 = 4 - \frac{8}{3}x_3 - \frac{2}{3}x_5 + \frac{1}{3}x_0$

z	x_1	x_2	x_4		x_3	x_5	x_6
1	0	0	0	28	$-\frac{1}{6}$	$-\frac{1}{6}$	$-\frac{2}{3}$
0	1	0	0	8	$\frac{1}{6}$	$\frac{1}{6}$	$-\frac{1}{3}$
0	0	1	0	4	$-\frac{8}{3}$	$-\frac{2}{3}$	$\frac{1}{3}$
0	0	0	1	18	$-\frac{1}{2}$	$\frac{1}{2}$	0

$z = 28$, $x'' = (8, 4, 0, 18, 0, 0)$
Nun sind alle Koeffizienten in der Zielfunktion negativ. Damit sind wir fertig und haben das Maximum gefunden.

Basiswechsel:
$F \ldots$ Menge der freien Variablen. $S \ldots$ Menge der Schlupfvariablen (Basisvariablen)
Eingabe: $(F, S, A, \underline{b}, \underline{c}, \nu, g, k)$
Ausgabe: $(\hat{F}, \hat{S}, \hat{A}, \underline{\hat{b}}, \underline{\hat{c}}, \hat{\nu})$ neue Schlupfform

$$z = \nu + \sum_{j \in F} c_j x_j$$
$$x_i = b_i - \sum_{j \in F} a_{ij} x_j \text{ für } i \in S$$
$$x_j \geq 0 \text{ für } j \in F \cup S$$
$$g \in S,\ k \in F\ x_g \leftrightarrow x_k \text{ danach } g \in \hat{F},\ k \in \hat{S}$$

Algorithm 56 BASIS-WECHSEL(F, S, A, b, c, ν, g, k)

1: // Berechne die Koeffizienten der Gleichung für die neue Basisvariable x_g
2: sei $\hat{A}$ eine neue $m \times n$-Matrix
3: $\hat{b}_k := \frac{b_g}{a_{gk}}$
4: **for** $j \in F \backslash \{k\}$ **do**
5: $\quad \hat{a}_{kj} := \frac{a_{gj}}{a_{gk}}$
6: **end for**
7: $\hat{a}_{kg} := \frac{1}{a_{gk}}$
8: // Berechne die Koeffizienten der übrigen Nebenbedingungen
9: **for** $i \in S \backslash \{g\}$ **do**
10: $\quad \hat{b}_i := b_i - a_{ik}\hat{b}_k$
11: $\quad$ **for** $j \in F \backslash \{k\}$ **do**
12: $\quad\quad \hat{a}_{ij} := a_{ij} - a_{ik}\hat{a}_{kj}$
13: $\quad$ **end for**
14: $\quad \hat{a}_{ig} := -a_{ik}\hat{a}_{kg}$
15: **end for**
16: // Berechne die Zielfunktion
17: $\hat{\nu} := \nu + c_k\hat{b}_k$
18: **for** $j \in F \backslash \{k\}$ **do**
19: $\quad \hat{c}_j := c_j - c_k\hat{a}_{kj}$
20: **end for**
21: $\hat{c}_g := -c_k\hat{a}_{kg}$
22: // Berechne die neue Menge der Basisvariablen und freien Variablen
23: $\hat{F} := F \backslash \{k\} \cup \{g\}$
24: $\hat{S} := S \backslash \{g\} \cup \{k\}$
25: **return** $(\hat{F}, \hat{S}, \hat{A}, \hat{b}, \hat{c}, \hat{\nu})$

18.7 Bemerkung. Die Zeilen 3-7 berechnen die Koeffizienten für die neue Gleichung für x_k, indem sie nach x_k auflöst:

$$x_g = b_g - \sum_{j \in F} a_{gj}x_j \Rightarrow x_k = \frac{1}{a_{gk}}\left(b_g - \sum_{j \in F \backslash \{k\}} a_{gj}x_j - x_g\right)$$

Die Zeilen 9-14 berechnen die neuen Koeffizienten für die anderen Gleichungen, indem

sie x_k substituiert:

$$x_i = b_i - \sum_{j \in F \setminus \{k\}} a_{ij} x_j - a_{ik} x_k$$
$$x_k = \hat{b}_k - \sum_{j \in F \setminus \{k\}} \hat{a}_{kj} x_j - \hat{a}_{kg} x_g$$

Schließlich aktualisieren die Zeilen 7-21 die Zielfunktion auf die gleiche Weise:

$$z = \nu + \sum_{j \in F \setminus \{k\}} c_j x_j + c_k x_k$$

18.8 Lemma. Sei $a_{gk} \neq 0$. Betrachte Aufruf $BW(F, S, A, \underline{b}, \underline{c}, \nu, g, k)$ mit Ergebnis $(\hat{F}, \hat{S}, \hat{A}, \underline{\hat{b}}, \underline{\hat{c}}, \hat{\nu}, \hat{g}, \hat{k})$, $\overline{\underline{x}}$ sei Basislösung nach dem Aufruf.
Dann gilt:

1.) $\overline{x_j} = 0 \quad \forall j \in \hat{F}$

2.) $\overline{x_k} = \frac{b_g}{a_{gk}}$

3.) $\overline{x_i} = b_i - a_{ik}\hat{b}_k \quad \forall i \in \hat{S} \setminus \{k\}$

Beweis:
Basislösung setzt alle freien Variablen auf $0 \Rightarrow$ 1.)
Wenn wir alle freien Variablen auf 0 setzen gilt $x_i = \hat{b}_i \quad \forall i \in \hat{S}$. Wegen $x_i = \hat{b}_i - \sum_{j \in \hat{F}} \hat{a}_{ij} x_j = \hat{b}_i \overset{\text{(Zeile 10)}}{=} b_i - a_{ik}\hat{b}_k$ folgt 3.)
$x_k = \hat{b}_k \overset{\text{Zeile 3}}{=} \frac{b_g}{a_{gk}} \Rightarrow$ 2.) ■

Annahme: Wenn die Funktion $INIT\text{-}SIMPLEX$ aufgerufen wird und die Basislösung nicht zulässig ist, soll die Funktion mit „unlösbar“ terminieren. Ansonsten gibt sie eine Schlupfform mit zulässiger Basislösung zurück.

Algorithm 57 SIMPLEX(F, S, A, b, c , ν)

```
1: (F, S, A, b, c, ν) := INIT-SIMPLEX(A, b, c)
2: Sei Δ ein neuer Vektor mit m Einträgen
3: while ∃j ∈ F : c_j > 0 do
4:     Wähle k ∈ F mit c_k > 0
5:     for i ∈ S do
6:         if a_ik > 0 then
7:             Δ_i := b_i / a_ik
8:         else
9:             Δ_i := ∞
10:        end if
11:    end for
12:    Wähle g ∈ S, sodass Δ_g minimal
13:    if Δ_g == ∞ then
14:        return „unbeschränkt“
15:    else
16:        (F, S, A, b, c, ν) := BASIS-WECHSEL(F, S, A, b, c, ν, g, k)
17:    end if
18: end while
19: for i = 1 to n do
20:    if i ∈ S then
21:        x̄_i := b_i
22:    else
23:        x̄_i := 0
24:    end if
25: end for
26: return (x̄_1, x̄_2, ..., x̄_n)
```

Korrektheit:

1. Falls $SIMPLEX$ terminiert:
   a) Ausgabe einer neuen Schlupfform mit zulässiger Basislösung
   b) lineares Programm unbeschränkt
2. Algorithmus terminiert
3. Ausgegebene Lösung ist optimal

4. ($INIT\text{-}SIMPLEX$)

18.9 Lemma. Gegeben ist ein lineares Programm in Standardform $(A, \underline{b}, \underline{c})$. Falls $SIMPLEX$ terminiert:

a) $\Rightarrow$ Falls eine Lösung ausgegeben, dann ist es eine zulässige Lösung

b) $\Rightarrow$ lineares Programm unbeschränkt

Beweis:
Schleifeninvariante für die *while*-Schleife: Am Beginn jeder Iteration gilt:

1. Schlupfform ist äquivalent zur Schlupfform von $INIT\text{-}SIMPLEX$
2. $\forall i \in S : b_i \geq 0$
3. Basislösung ist zulässig

Initialisierung: 1. und 3.: klar
Ad 2.: Basislösung: $x_i = b_i \, \forall i \in S \overset{\text{Lösung zulässig}}{\Rightarrow} x_i \geq 0 \Rightarrow b_i \geq 0$

Aufrechterhaltung: während *while*-Schleife: $x_g \leftrightarrow x_k$ durch $BASIS\text{-}WECHSEL$.
$BASIS\text{-}WECHSEL$ gibt äquivalente Schlupfform zurück $\Rightarrow$ 1.)

Ad 2.: $b_i \geq 0 \, \forall i \in S$ am Beginn der *while*-Schleife
vor Basiswechsel: b_i, a_{ij}, S, nachher: $\hat{b}_i, \hat{a}_{ij}, \hat{S}$
$\hat{b}_k \geq 0$, denn $b_g \geq 0$, $a_{gk} > 0$ („$if\ a_{ik} > 0 \ldots$“, „Wähle $g \ldots$“)
Sei $i \in S \backslash \{g\}$. Dann gilt nach Zeile 10 in $BASIS\text{-}WECHSEL$:
$\hat{b}_i = b_i - a_{ik}\hat{b}_k = b_i - a_{ik}\frac{b_g}{a_{gk}}$

1. Fall: $a_{ik} > 0$: g so gewählt, dass $\frac{b_g}{a_{gk}} \leq \frac{b_i}{a_{ik}} \, \forall i \in S$
   $\Rightarrow \hat{b}_i \geq 0$
2. Fall: $a_{ik} \leq 0 \Rightarrow \hat{b}_i \geq 0$, da a_{gk}, $b_g > 0$

Ad 3.: Zu zeigen: Alle $x_i \geq 0$. Für $i \in F$: klar, da diese auf 0 gesetzt werden
Sei $i \in S$: $x_i = \hat{b}_i - \sum_{j \in F} \hat{a}_{ij} x_j = \hat{b}_i \overset{(2)}{\geq} 0$

Terminierung: Zwei Möglichkeiten:

1. *while*-Bedingung verletzt: Basislösung zulässig, wird ausgegeben

2. return „unbeschränkt“: $\forall i \in S : \Delta_i = \infty$, $\forall i \in S : a_{ik} \leq 0$

Betrachte Lösung: $\overline{x}$:

$$\overline{x}_i = \begin{cases} \infty & \text{, für } i = k \\ 0 & \text{, für } i \in F \backslash \{k\} \\ b_i - \sum\limits_{j \in F} a_{ij} \overline{x}_j & \text{, für } i \in S \end{cases}$$

$\Rightarrow \overline{x}_i \geq 0 \quad \forall i \in F$

Sei $i \in S : \overline{x}_i = b_i - \sum\limits_{j \in F} a_{ij} \overline{x}_j = b_i - a_{ik} \overline{x}_k \geq 0$, da $b_i \geq 0$, $a_{ik} \leq 0$ und $\overline{x}_k = \infty$

$c_k > 0 \Rightarrow z = \nu + \sum\limits_{j \in F} c_j \overline{x}_j = \nu + c_k \overline{x}_k = \infty$

Ad *while*-Bedingung verletzt: $z = \nu + \sum\limits_{j \in F} c_j x_j$, alle $c_j < 0 \Rightarrow$ Maximum ■

Terminierung des $SIMPLEX$-Algorithmus:
z-Wert ist monoton wachsend, möglicherweise aber nicht strikt. Dieses Phänomen nennt man Kreisen.

18.10 Beispiel.

$$\begin{aligned} z &= x_1 + x_2 + x_3 \\ x_4 &= 8 - x_1 - x_2 \\ x_5 &= x_2 - x_3 \end{aligned}$$

$$\begin{aligned} &x_k := x_1 \Rightarrow x_g = x_4 \\ &z = 8 + x_3 - x_4 \\ &x_1 = 8 - x_2 - x_4 \\ &x_5 = x_2 - x_3 \\ &z = 8 \end{aligned}$$

$$\begin{aligned} &x_k := x_3 \Rightarrow x_g = x_5 \rightarrow z = 8 \text{ bleibt} \\ &z = 8 + x_2 - x_4 - x_5 \\ &x_1 = 8 - x_2 - x_4 \\ &x_3 = x_2 - x_5 \end{aligned}$$

18.11 Lemma. $(A, \underline{b}, \underline{c})$ Standardform, S gegeben $\Rightarrow$ Schlupfform ist eindeutig bestimmt

Beweis:
Annahme: Es existieren 2 verschiedene Schlupfformen

$$F = \{1, 2, \ldots, m+n\} \backslash S$$
$$z = \nu + \sum_{j \in F} c_j x_j$$
$$x_i = b_i - \sum_{j \in F} a_{ij} x_j \quad \forall i \in S$$

$$z = \nu' + \sum_{j \in F} c'_j x_j$$
$$x_i = b'_i - \sum_{j \in F} a'_{ij} x_j \quad \forall i \in S$$

$$\Rightarrow 0 = (b_i - b'_i) - \sum_{j \in F} \left(a_{ij} - a'_{ij}\right) x_j \quad i \in S$$
$$\Leftrightarrow \sum_{j \in F} a_{ij} x_j = (b_i - b'_i) + \sum_{j \in F} a'_{ij} x_j \quad i \in S$$
$$\Rightarrow b_i = b'_i,\ a_{ij} = a'_{ij}$$
analog: $\nu = \nu'$, $c_j = c'_j$

■

18.12 Lemma. $SIMPLEX$ terminiert nach höchstens $\binom{n+m}{n}$ Iterationen oder nie.
Beweis:
Anzahl der Möglichkeiten S zu wählen ist $\binom{n+m}{n}$. Wenn $SIMPLEX$ mehr als $\binom{n+m}{n}$ Iterationen ausführt, muss er kreisen und terminiert daher nie. ■

Regel von Bland: Wähle immer eine Variable mit kleinstmöglichem Index als x_k. Dann lässt sich beweisen, dass $SIMPLEX$ immer terminiert.

18.13 Satz. Falls $INIT\text{-}SIMPLEX$ eine zulässige Basislösung ausgibt und die Regel von Bland eingehalten wird, dann gilt für den $SIMPLEX$-Algorithmus:

1. Das lineare Programm ist unbeschränkt, oder
2. man erhält die korrekte Lösung nach höchstens $\binom{n+m}{n}$ Iterationen.

Stichwortverzeichnis

Pseudocodeverzeichnis

Literatur

[CLR+13] Cormen, Thomas H. ; Leiserson, Charles E. ; Rivest, Ronald ; Stein, Clifford ; Molitor, Paul: Algorithmen - Eine Einführung -. überarbeitete und aktualisierte Auflage. München : Oldenbourg Verlag, 2013. – ISBN 978–3–486–74861–1

