Vorlesungsmitschrift

Höhere Algorithmik

gelesen von Prof. Dr. Günter Rote

Tobias Höppner

Wintersemester 2014/2015

Inhaltsverzeichnis

1 Einführung (Vorlesung 1 am 17.10.)

1.1 Organisatorisches

Mitschrift wird von Studenten erstellt.

Korrekturfarbe für Gummipunkte: Grün!

Voraussetzungen

- O-Notation
- Turing-Maschine
- Sortieralgorithmen
- Schubfachprinzip
- Gauß-Nummer
- Harmonische Reihe

1.2 Kuchen teilen

Problem: Ein Kuchen soll unter zwei Personen aufgeteilt werden.
Zwei Lösungsideen:
- perfektes Teilen
- einer teilt den Kuchen und der andere sucht sich eine Hälfte aus.

Was passiert, wenn jemand die Teile des Kuchens unterschiedlich bewertet? (z.B. Kirsche auf einer Seite, viel Sahne auf der anderen Seite)
Perfektes teilen bedeutet, dass jemand *für sich* perfekt teilt. (nach seinem Maßstab)
Ziel: Fairness Jeder will $\frac{1}{n}$ des Kuchens nach ihrem Maßstab. ($n = \#$Personen)

1.2.1 1. Algorithmus (für 2 Personen)

1. Erste teilt
2. Zweite sucht aus

Der Algorithmus ist toll, aber es gibt zu viele Schritte. Daher wollen wir den Algorithmus verbessern.
Ziel: möglichst wenige Schritte.

1.2.2 2. Algorithmus (für 3 Personen)

Anton, Berta und Clara:

1. Anton teilt $\frac{1}{3}|\frac{2}{3}$
2. Berta teilt $\frac{\frac{2}{3}}{2}|\frac{\frac{2}{3}}{2}$
3. Clara sucht aus.
4. Anton sucht aus.

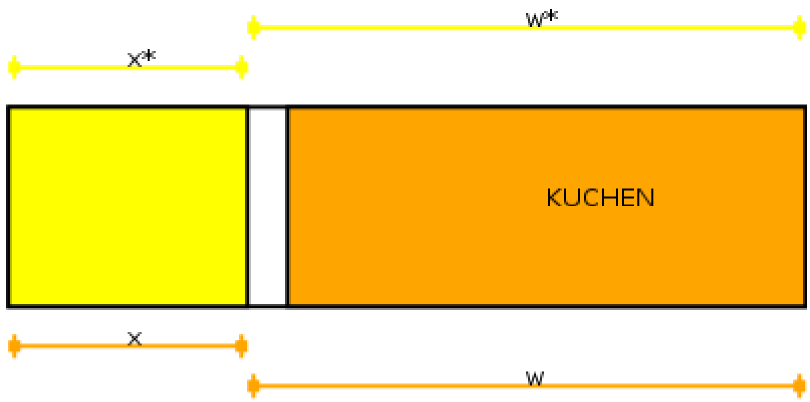
   Fall 1: Clara nimmt eines der rechten Stücke $\Rightarrow$ Anton nimmt linkes Stück.

Fall 2: Clara nimmt linkes Stück.
Schubfachprinzip: eines der rechten Stücke ist mindestens $\frac{1}{3}$

5. Berta):

1.2.3 3. Teilen und Trimmen

1. Anton teilt:



2. Berta:

   Fall 1: Berta denkt $x \leq \frac{1}{3}$

   Fall 2: Berta denkt $x > \frac{1}{3} \Rightarrow$ Trimmen

3. Clara darf sich entscheiden:

   Fall 1: will x^* dann Algorithmus 1. für den Rest

   Fall 2: will x^* nicht.
   $\Rightarrow w^* \geq \frac{2}{3}$ für Clara und Anton

1.2.4 4. Teilen mit bewegtem Messer

Man nimmt ein Messer und jede Person sagt einfach Stop, wenn die *perfekte Wahl* für die Person getroffen ist.
$\#$Schritte $= n - 1$

1.2.5 5. Simuliertes bewegtes Messer

- Jeder macht bei $\frac{1}{n}$ eine Markierung
- der/die Linkeste bekommt das Stück
  $\#$Schritte $= n + (n-1) + ...3 + 2 = \theta(n^2)$ (Gauß-Nummer)

1.2.6 6. Simuliertes Messer + Zufall

Wie 5., aber

1. Reihenfolge zufällig
2. nur neue Linkeste Markierung werden gemacht

3. $T(n) = \#$erwartete Markierungen
$= \underbrace{\frac{1}{n}}_{\text{Erwartete Anzahl der letzten Markierung}} + \underbrace{T(n-1)}_{\text{Erwartete Anzahl von Markierungen aller Anderen.}}$

4. $T(n) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} = \theta(\log n)$ (harmonische Reihe)

5. Gesamtlaufzeit $\leq n * O(\log n) = O(n * \log n)$

1.2.7 7. Divide & Conquer

n Personen
n Markierungen bei $\frac{k}{n}$
$\#$Schritte im Worst Case $T(n) = n + 2$

1.2.8 8. Divde & Conquer + Zufall

(erwartete) Laufzeit pro Teilen $\theta(\log n)$ also insgesamt $\theta(n)$

2 Einführung Teil 2 (Vorlesung 2 am 20.10.)

2.1 Ziele der Vorlesung

- Algorithmen nach den wichtigsten Entwurfsprinzipien entwerfen:
  - Devide and Conquer
  - dynamisches Programmieren
  - bound and bound
  - greedy-Algorithmen
- Algorithmen mit Analysetechniken analysieren im Bezug auf Laufzeit und Speicherbedarf (Stromverbrauch)
  - randomisierte Analyse
  - amortisierte randomisierte Analyse
  - Rekursionsgleichungen
- Vergleich und Beurteilung von Algorithmen nach Einsatzzweck
- Theorie der NP Vollständigkeit verstehen und einfache Vollständigkeitsbeweise führen

(Stromverbrauch ist zunehmend wichtig, aber nicht Teil der Vorlesung. Allgemein sind Algorithmen mit weniger Laufzeit besser.)

2.2 Rechnermodelle

2.2.1 Turing-Maschine

Eine Turing-Maschine ist ein theoretisches Modell. Es handelt sich um ein unendliches Band mit Symbolen aus einem endlichen Alphabet mit endlichem Zustandsraum. In jedem Schritt wird ein Symbol gelesen, das Band entsprechend der Eingabe beschrieben und der Zustand verändert. Prinzipiell ist alles mit einer Turing-Maschine berechenbar, jedoch teilweise sehr umständlich, weil immer nur ein Symbol gelesen werden kann.

2.2.2 Registermaschine (RAM - random access machine)

RAM ist auch als random access memory als Arbeitsspeicher bekannt

Eine RAM funktioniert nach einem ähnlichen Prinzip wie moderne Rechner arbeiten. Es gibt eine potentiell unendliche (unbeschränkte) Anzahl von Registern `R0, R1, R2, ...` wobei jedes Register eine ganze Zahl enthalten kann. Die Programmiersprache ist ähnlich wie Assembler.

1. Befehle

Zuweisung `R4 = R17`

Rechenbefehl `R1 = R2 + R3`

```
R1 = R2 - R3
R1 = R2 * R3
R1 = R2 / R3
```

Operanden der Befehle

1. Register `R17`
2. direkte Operanden (Zahlen) `250`
3. indirekte Adressen: `(R1)`

den Inhalt des Registers, dessen Nummer in Register R1 steht.

2. Sprünge

```
GOTO x
IF R_i = 0 THEN GOTO x

GZ R1, label ;if R1 is greater 0, goto label
```

Es sind nur die drei Vergleichsoperationen GLZ: < 0 , GGZ: > 0 , GZ: = 0 erlaubt!

x ist eine Sprungmarke im Programm.

```
loop:
  \\ some commands
  GOTO loop
```

3. HALT

Ein Programm endet immer mit `HALT`

Ein- und Ausgabe

Eingabe: `R0 = n`= die Länge der Eingabe `R1, R2, ... Rn`. Alle andere Zellen sind auf `0` initialisiert.
Ausgabe steht am Ende im Speicher!

2.2.3 Berechnung der Laufzeit

a) Einheitskostenmaß (EKM)

Jede Operation dauert eine Zeiteinheit.
unfair, weil es Operationen gibt, die offensichtlich komplizierter sind.

b) logarithmisches Kostenmaß (LKM)

Laufzeit = Summe der Längen aller vorkommenden Adressen und Operanden.

$$l(x) = \lfloor \log_2 \max\{|x|, 1\} \rfloor + 1$$

$$\texttt{R2} = (\texttt{R0}) + 250$$
$$\text{...Kosten} = l(2) + l(0) + \underbrace{l(\texttt{R0})}_{\text{Adresse}} + \underbrace{l((\texttt{R0}))}_{\text{Operand}} + \underbrace{l(250)}_{\text{Operanden}}$$

Das LKM ist gerechter, als das EKM.

Im EKM kann man schwindeln:
Operationen auf langen Daten können in einem Schritt erledigt werden.
Andererseits ist das EKM näher an einem tatsächlichen Prozessor. Sofern die Operanden in ein Wort eines konventionellen Speichers (64 Bit) passen.
Abschätzung: LKM $\leq O($EKM . $l($längster vorkommender Operand oder Adresse))
Wenn die größten vorkommenden Zahlen nicht zu groß sind, dann ist das EKM realistisch.
LKM ist fairer, wenn es um sehr unterschiedliche Operanden geht (verschieden lang)

2.3 Laufzeit eines Algorithmus

Man muss den möglichen Eingaben eine Länge zuordnen.
x.. Eingabe $L(x)$
Bsp. n Zahlen $x_1, x_2, ..., x_n$ sortieren: $L = \underline{n}$
Bsp. Multiplikation von langen Zahlen x, y: $L = \#$ Bits in der Eingabe.
Bsp. Lösen eines linearen Gleichungssystems: $Ax = bA \in \mathbb{Z}^{n \times x} b \in \mathbb{Z}^n x \in \mathbb{Q}^n$
Länge der Eingabe: n^2
Gauß-Elimination $O(n^3)$ Zeit, erfordert Rechnen mit rationalen Zahlen.
Man kann Zeigen, dass die Länge der Zähler und Nenner in den Zwischenergebnissen höchstens n-Mal ($\leq n$) ist, wenn man Brüche immer kürzt.
Laufzeit im LKM: $O(n^4, l(\text{größte Eingabezahl}))$

Tendenziell kompliziertes Beispiel, um zu illustrieren, dass LKM nicht immer leicht zu berechnen ist.

Was ist die Laufzeit eines Algorithmus?

$$T(x) = \text{Laufzeit des Algorithmus bei Eingabe} x$$

$$(AnalyseimschlimmstenFall).T(n) = \max\{T(x) | L(x) = n\}$$

Andere Möglichkeiten

Analyse im Durchschnitt, Erwartungswert der Laufzeit
Benötigt eine Wahrscheinlichkeitsverteilung auf der Menge der Eingaben.

3 Rechnermodelle (Fortsetzung) (Vorlesung 3 am 24.10.)

3.1 Warum nicht die Turingmaschine?

Die Registermaschine ist näher am heutigen Rechnermodell. Die Turingmaschine ist viel primitiver.
Satz:

a) Ein Alogrithmus, der auf einer Registermaschine Laufzeit $T(n)$ im logarithmischen Kosteneinheitsmaß hat, kann auf einer Turingmaschine in Laufzeit $O((T(n))^3)$ simuliuert werden.

b) Ein Alogirhtmus mit Laufzeit $U(n)$ auf einer Turingmaschine kann mit Laufzeit $O(U(n) \log U(n))$ auf einer Registermaschine im LKM simuliert werden.

zu b) In Zeit $U(n)$ kann die Maschine höchstens die Felder $-U(n)... + U(n)$ beschreiben. Adressen sind durch $2U(n)$ beschränkt jeden Schritt der TM kann in konstant vielen Operationen der Registermaschine simuliert werden.
$\rightarrow O(\log U(n)$

zu a) Speicherinhalt auf dem Band notieren.
i : (Inhalt von Register i).($i + 1$:Inhalt von Register($i + 1$). ...
Register mit Inhalt 0 können weggelassen werden. Register werden in natürlicher Reihenfolge aufgeschrieben. Alle Zahlen binär oder dezimal (nach Belieben).

Die Länge des Bandes $= L$ ist durch $T(n)$ beschränkt.
Jede Adresse, jede Registereinheit wurde bei der letzten Benutzung in voller Länge bei $T(n)$ berücksichtigt.

3.2 Elementare Operationen

1. Adresse im Speicher suchen; (Adresse steht im linken Zwischenbereich)
2. entsprechenden Inhalt zwischen Speicher und Zwischenbereich übertragen
3. Rechenoperationen im Zwischenbereich

```
R2 = (R17)
```

Jede Stelle die verglichen wird, erfordert im schlimmsten Fall ein Wandern über das gesamte Band.

Operation 1 dauert $O(L^2)$ Schritte, wobei L die Länge des Bandes ist.

Operation 2 ist ähnlich. Gegebenenfalls muss man den rechten Teil des Bandinhalts verschieben (Um eine Stelle verschieben dauert $O(L)$ Zeit, $\leq O(L^2)$ insgesamt).
Operation 3 $\leq O(L^2)$

$O(L^2)$ für 1 Schritt der Registermaschine $= O(T(n))^2$

3.3 Teile und Herrsche

(eng. divide and conquer) (lat. divide et impera)

1. Zerlege das Problem P in Teilprobleme $P_1, P2, ..., P_k$ (typischerweise $k = 2$)
2. Löse die Teilprobleme rekursiv.
3. Füge die Teillösung zur Lösung von P zusammen.

3.3.1 Beispiel A: Quicksort

1. Wahl eines Pivotelementes a:
   Zerlegung in Elemente $\underbrace{< a}_{\text{Teilproblem}}, = a, > a$

3. Teilfolgen aneinanderhängen.

Bei Quicksort ist der erste Schritt aufwändiger, bei Mergesort der letzte Schritt.

3.3.2 Beispiel B: Mergesort (Sortieren durch Verschmelzen)

1. Zerlegung in 2 gleich große Teile

3. Verschmelzen der beiden sortierten Teillisten.

Laufzeit $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + \Theta(n)$

n gerade $T(n) = 2T(\frac{n}{2}) + \Theta(n)$

Lösung $T(n) = O(n \log n)$

3.3.3 Analysemöglichkeiten

I. Lösung <u>erraten</u> und durch vollständige Induktion beweisen.

II. Wiederholtes einsetzen auf der rechten Seite:

$$\begin{aligned}
T(n) &\leq 2T(\frac{n}{2}) + cn \quad (c > 0) \\
T(\frac{n}{2}) &\leq 2T(\frac{n}{4}) + c * \frac{n}{2} \\
T(n) &\leq 2(2T(\frac{n}{4}) + c\frac{n}{2}) + cn \\
&= 4 \underbrace{T(\frac{n}{4})}_{=2T(\frac{n}{8})+c\frac{n}{4}} + cn + cn \\
&\leq 8T(\frac{n}{8}) + cn + cn + cn \\
&\leq 2^k T(\frac{n}{2^k}) + k.c.n
\end{aligned}$$

Annahme $n = 2^l$ ist eine Zweierpotenz $l = \log_2 n$

$$T(n) = \underbrace{2^l}_{n} \underbrace{T(1)}_{\text{konst.}} + \underbrace{l}_{\log_2 n} .c.n = O(n \log n)$$
$$= O(n) + O(n \log n)$$

nur gültig für Zweierpotenzen.

Möglichkeit a) n auf die nächste $n' = 2^l$ aufrunden.
$n \leq n' < 2n$
Sortieren von n Elementen kann nicht länger dauern als Sortieren von n' Elementen.
(zu beweisen! z.B. mit vollst. Indunktion anhand der Rekursion)
$T(n) \leq T(n') = O(n' \log n') = O(2n. \log(2n)) = O(n \log n)$✓

Möglichkeit b) Als Inspiration, um auf die Vermutung $O(n \log n)$ zu bekommen. Beweis mit Methode I.

III. Rekursionsbaum $\lfloor \frac{\lfloor \frac{n}{2} \rfloor}{2} \rfloor = \lfloor \frac{n}{4} \rfloor$ Laufzeit: 2^l Probleme konstanter Größe. $T(1), T(2) \leq c'$

$$\begin{aligned}
\text{Ebene 0}: &\leq \Theta(n) \\
\text{Ebene 1}: &\leq 2\Theta(\lceil \frac{n}{2} \rceil) \\
\text{Ebene 2}: &\leq 4\Theta(\lceil \frac{n}{4} \rceil) \\
\Theta(n) &\leq c.n \\
\text{Summe} &\leq cn + 2c\lceil \frac{n}{2} \rceil + 4c\lceil \frac{n}{4} \rceil + ... + 2^{l-1}c\lceil \frac{n}{2^{l-1}} \rceil + 2^l c' \\
&\leq cn + 2c(\frac{n}{2} + 1) + 4c(\frac{n}{4} + 1) + ... \\
&= \underbrace{cn + cn + ... + cn}_{\text{l-mal}} + \underbrace{2c + 4c + 8c + ... + 2^{l-1}c}_{(2^l-2)c} + 2^l c'
\end{aligned}$$

Datum der Vorlesung : 27.10.2014
Autor : Hinnerk van Bruinehsen
email : h.v.bruinehsen@fu-berlin.de

Höhere Algorithmik - 4. Vorlesung

Bestimmung des Maximums und des Minimums von n Zahlen

Problemstellung: Gegeben sind die Zahlen $a_1, ..., a_n$. Gesucht werden das Maximum sowie das Minimum von diesen zahlen.
Will man entweder nur das Maximum oder nur das Minimum dieser Zahlen bestimmen, vergleicht man die erste Zahl mit der zweiten. Je nach gesuchtem Ergebnis muss man entweder den größeren oder den kleineren der beiden Werte mit dem nächsten Wert vergleichen. Das ganze wird fortgesetzt, bis alle Zahlen miteinander verglichen wurden.
Aus diesem Algorithmus folgt, dass zur Bestimmung des Maximums allein $n-1$ Vergleiche ausreichen. Genauso reichen zur Bestimmung des Minimums $n-1$ Vergleiche.
In der Summe sind dies $2n-2$ Vergleiche. Die asymptotische Laufzeit liegt in $\mathcal{O}(n)$.
Anschaulich ist auch sofort klar, dass es unmöglich ist, eine bessere als lineare Laufzeit zu erreichen, da sämtliche Zahlen betrachtet werden müssen. Daher ist dies einer der wenigen Fälle in dieser Vorlesung, in der die Konstante betrachtet und verbessert werden soll.
Optimierung: Für $n \geq 2$ kann das Maximum kann nicht gleichzeitig das Minimum sein (und umgekehrt). Hieraus kann gefolgert werden, dass nur $2n-3$ Vergleiche benötigt werden.

Teile und herrsche

Wir betrachten die Teilfolgen L (links) und R (rechts) mit $\lfloor \frac{n}{2} \rfloor$ und $\lceil \frac{n}{2} \rceil$ Elementen.
Das maximale Elemement der linken Teilfolge L sei l_{max}, das minimale l_{min}. Analog dazu seien r_{max} das maximale und r_{min} das minimale Element der rechten Teilfolge R.

Bestimme $l_{min}, l_{max}, r_{min}, r_{max}$

$$T(n) = \text{ Anzahl der Vergleiche}$$

$$\begin{aligned}
&T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 2 \\
&T(1) = 0 \\
&T(2) = 1 \\
&T(3) = 3 = 2 + 1 + 0 \\
&T(4) = 4 = 2 + 1 + 1 \underline{\leq 2n - 3}
\end{aligned}$$

Zur Bestimmung des gesammten Maximums benötigen wir zwei weitere Vergleiche: Das maximale Element von der Gesammtfolge ist das Maximum von l_{max} und r_{max}, das minimale Element der Gesamtfolge ist das Minimum von l_{min} und r_{min}.

Analyse falls n eine Zweierpotenz ist ($n = 2^k$):

$$T(n) = 2T(\frac{n}{2}) + 2$$

<u>Ansatz:</u> $T(n) = An + B$ (lineare Funktion)
Durch Einsetzen unseres Ansatzes erhalten in die Rekursionsgleichung folgt:

$$\begin{aligned} An + B &= 2(A\frac{n}{2} + B) + 2 \\ &= An + 2B + 2 && |(-An - B) \\ 0 &= B + 2 && |(-2) \\ B &= -2 \end{aligned}$$

Bestimmung von A durch Einsetzen von $B = -2$ in den Ansatz:

$$\begin{aligned} T(n) &= An + B && \text{für den Fall } T(2) = 1 \text{ folgt:} \\ T(2) &= A2 + B = 1 \\ T(2) &= A2 - 2 = 1 && (+2) \\ 2A &= 3 && (/2) \\ A &= \frac{3}{2} \end{aligned}$$

Für $A = \frac{3}{2}$ und $B = -2$ erfüllt $A \cdot n + B$ also die Rekursion. Wir erhalten also als Lösung für den Fall $n = 2^k$ (n ist Zweierpotenz):

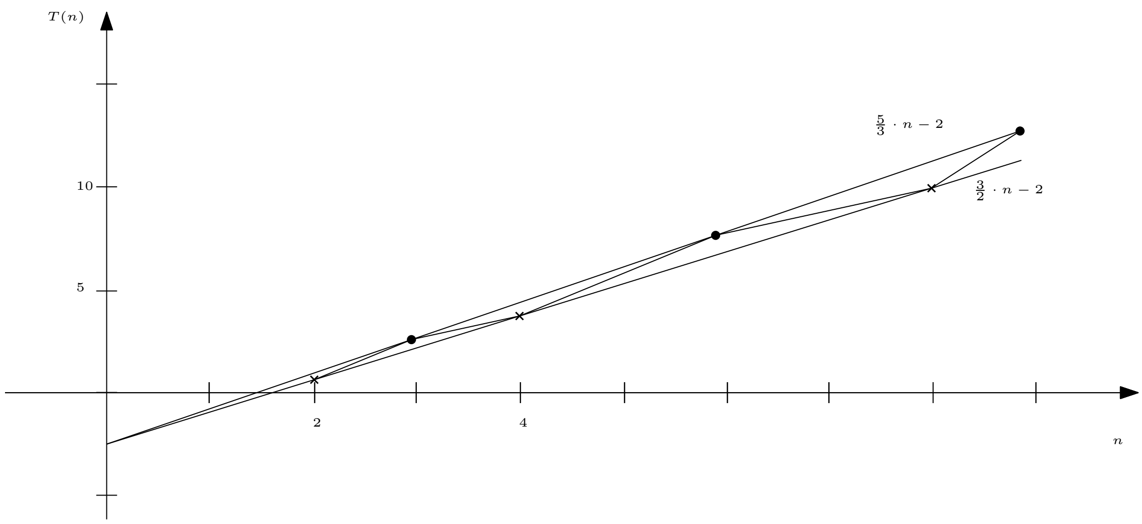
$$T(n) = \frac{3}{2}n - 2$$

Verschiebung im Wertebereich

Eine leicht zu lösende Rekursion hat zum Beispiel die Form: $h(n) = 2h(\frac{n}{2}) = 4h(\frac{n}{4}) \Rightarrow h(n) = \underline{an}$
Schwieriger ist es, wenn die Gleichung einen Störfaktor enthält: $f(n) = 2f(\frac{n}{2}) + \underbrace{2}_{\text{Störfaktor}}$

$$
\begin{aligned}
f(n) &= 2f(\frac{n}{2}) + 2 && |(+2) \\
\Leftrightarrow f(n) + 2 &= 2f(\frac{n}{2}) + 4 && \text{(2 ausklammern)} \\
\Leftrightarrow f(n) + 2 &= 2(f(\frac{n}{2}) + 2) \\
\text{definiere } g(n) &= f(n) + \underbrace{2}_{\text{Durch Ansatz}} \\
g(n) &= 2g(\frac{n}{2}) && \text{additiver Störfaktor ist weg.} \\
g(n) &= f(n) + c \\
f(n) &= 2f(\frac{n}{2}) + 2 \\
f(n) &= g(n) - c \\
g(n) - c &= 2(g(\frac{n}{2}) + 2 = 2g(\frac{n}{2}) - 2c + 2 \\
-c &= -2c + 2 \\
c &= 2
\end{aligned}
$$

Einschub:



Beispiel: (Fibonacci-Folge mit Störfaktor)

$$
\begin{aligned}
a_n &= a_{n-1} + a_{n-2} + 3 && |(+3) \\
(a_n + 3) &= (a_{n-1} + 3) + (a_{n-2} + 3) && \text{Fibonacci um 3 verschoben}
\end{aligned}
$$

Verschiebung im Definitionsbereich

Ähnlich, wie die Verschiebung im Wertebereich funktioniert die Verschiebung im Definitionsbereich.

Beispiel:

$$g(n) = 2\left(g(\lfloor\frac{n+3}{2}\rfloor)\right) \qquad \text{substituiere } n = m+3$$
$$g(m+3) = 2\left(g(\lfloor\frac{m+6}{2}\rfloor)\right)$$
$$g(m+3) = 2\left(g(\lfloor\frac{m}{2}+\frac{6}{2}\rfloor)\right)$$
$$g(m+3) = 2\left(g(\lfloor\frac{m}{2}+3\rfloor)\right)$$
$$g(m+3) = 2\left(g(\lfloor\frac{m}{2}\rfloor+3)\right)$$

Wir setzen $h(n) = g(n+3)$ und erhalten:

$$h(n) = 2\left(h(\lfloor\frac{n}{2}\rfloor)\right)$$

Auf diese Art haben wir den Störfaktor innerhalb des Funktionsaufrufs beseitigt und könnten jetzt regular mit der Bearbeitung dieser Aufgabe weitermachen.

Erweiterung auf nicht-2er Potenzen

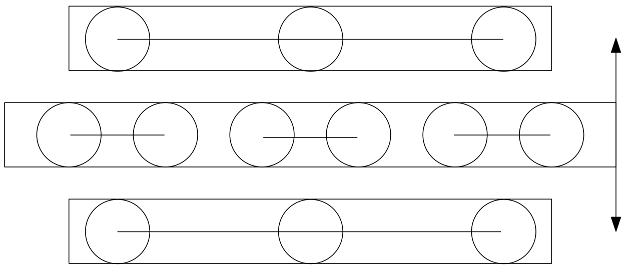


Abbildung 1: Vergleiche zum Finden von Maxima und Minima

Sei M eine n elementige Menge. Zur Vereinfachung gehen wir davon aus, dass n eine gerade Zahl ist. Wir bilden nun $\frac{n}{2}$ Paare und vergleichen diese miteinander. Hierfür benötigen wir $\frac{n}{2}$ Vergleiche.
Wir teilen die Elemente in zwei $\frac{n}{2}$ elementige Untermengen (ähnlich, wie bei Mergesort), von denen eine die Maxima, die andere die Minima aus den vorangegangenen Vergleichen enthält. Beide Untermengen haben nun eine ungerade Anzahl an Elementen. Wir vergleichen wieder paarweise die Maxima und die Minima. Dafür benötigen wir jeweils $\frac{n}{2} - 1$ Vergleiche:

$$\left.\begin{array}{l}\frac{n}{2}\\ \frac{n}{2}-1\\ \frac{n}{2}-1\end{array}\right\} 3\frac{n}{2}-2$$

Insgesamt benötigen wir also $\frac{3}{2}n - 2$ Vergleiche - und damit genauso viele wie für den Fall dass n als Zweierpotenz darstellbar ist. Das Verfahren funktioniert nach dem Bottom-Up Prinzip.

Multiplikation von zwei n-stelligen Zahlen

Gegeben ist eine Basis B, z.B. $B = 2$ (binär) oder $B = 10$ (dezimal) oder $B = 2^{32}$

$$x = (x_{n-1}x_{n-2}...x_1x_0)_B = \sum_{i=0}^{n-1} x_i B^i$$

32 Bit	32 Bit	32 Bit	32 Bit	32 Bit

Abbildung 2: 32 Bit Blöcke, auf diesen Schulmethode anwenden

$y = (y_{n-1}y_{n-2}...y_1y_0)_B = \sum_{i=0}^{n-1} y_i B^i$

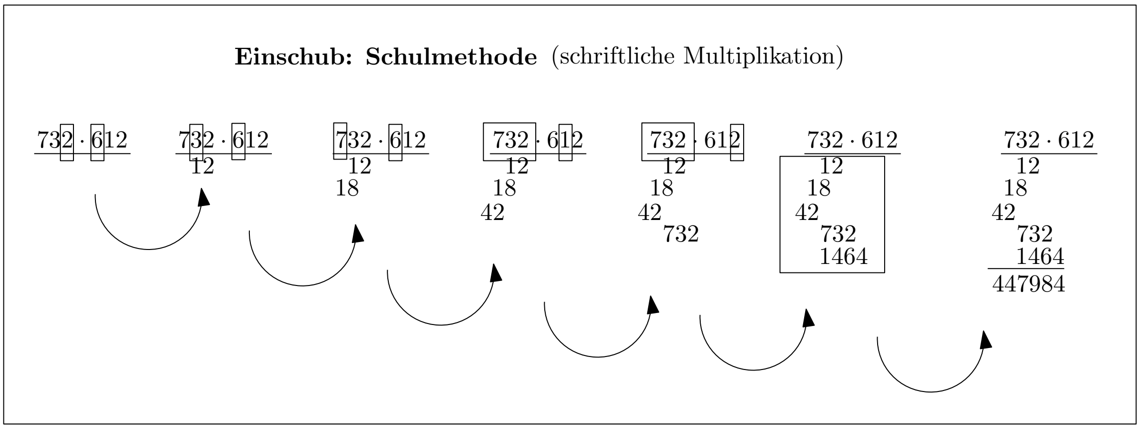


Abbildung 3: Schulmethode (ohne Übertrag)

Schulmethode: Multipliziere jedes x_i mit jedem y_j und addiere alle Produkte an die geeignete Stelle (wie in Abbildung 3).

$$x \cdot y = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} x_i y_j \, B^{i+j} \text{ Laufzeit in } \mathcal{O}(n^2)$$

Teile und herrsche (Basis $B = 2$)

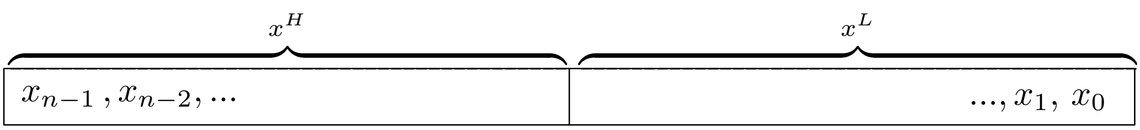


Abbildung 4: Teile und herrsche

Wie in Abbildung 4 zu sehen, sei x eine n Bit lange Zahl. Zur Vereinfachung nehmen wir an, dass n gerade ist.
Wir teilen x in zwei Teile, wobei ein Teil die höherwertigen Bits und der andere Teil die niedrigwertigen Bits enthält. Den Teil mit den höherwertigen Bits nennen wir x^H, den mit den niedrigerwertigen Bits nennen wir x^L.
Dann gilt $x = x^H \cdot 2^{\frac{n}{2}} + x^L$.
Außerdem sei y eine entsprechend gewählte zweite Zahl für die gilt:
$y = y^H \cdot 2^{\frac{n}{2}} + y^L$

$$x^H := (x_{n-1}x_{n-2}...x_{\frac{n}{2}})_2 \qquad \frac{n}{2} \text{ höherwertige Bits}$$

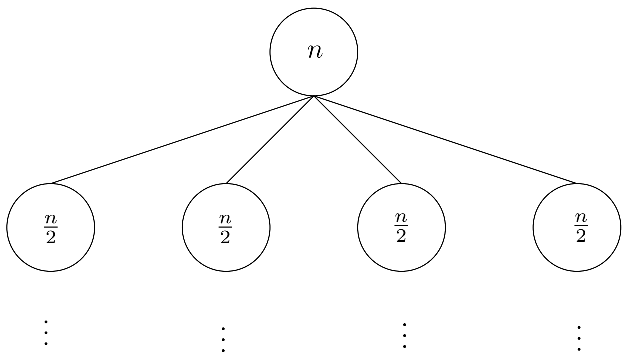
$$x^L := (x_{\frac{n}{2}-1}...x_0)_2 \qquad \frac{n}{2} \text{ niederwertige Bits}$$

$$x = x^H \cdot \underbrace{2^{\frac{n}{2}}}_{\text{Linksshift umi}2^{\frac{n}{2}}\text{ Bits}} + x^L$$

$$\begin{aligned} xy &= (x^H \cdot 2^{\frac{n}{2}} + x^L)(y^H \cdot 2^{\frac{n}{2}} + y^L) \\ &= x^H y^H + (x^H y^L + x^L y^H)2^{\frac{n}{2}} + x^L y^L \end{aligned}$$

$$T(n) = 4T(\frac{n}{2}) + \mathcal{O}(n)$$

Diese Rekursion ist dargestellt in Abbildung 5.



	1 Problem mit n	$c \cdot n$
	4 Probleme mit $\frac{n}{2}$	$4c\frac{n}{2} = 2cn$
	16 Probleme mit $\frac{n}{4}$	$16c\frac{n}{4} = 4cn$
	4^i Probleme mit $\frac{n}{2^i}$	$2^i cn$
$n = 2^k$	4^k Probleme mit 1 Bit	$2^n cn \cdot 4^k c = cn^2$

Abbildung 5: Rekursionsbaum - Schulmethode

$\Rightarrow$ Laufzeit: $T(n) = \mathcal{O}(n^2)$
Somit ist bringt „Teile und Herrsche"in diesem Fall keine Verbesserung. Grund hierfür ist, dass wir vier Teilbäume haben.
Um eine Verbesserung zu erzielen, müssen wir die Anzahl der Teilbäume auf drei reduzieren.

Algorithmus von Karatsuba

Satz: Es existiert ein Algoritmus, mit dem die Multiplikation zweier n-stelliger Zahlen in weniger als $\mathcal{O}(n^2)$ möglich ist.
Ein Algorithmus, der diesen Satz erfüllt ist der Algorithmus von Karatsuba.
Der Algorithmus von Karatsuba ist ein schneller Multiplikationsalgorithmus. Er reduziert für die Multiplikation zweier n-stelliger Zahlen die Anzahl der nötigen einstelligen Multiplikationen im Allgemeinen auf höchstens $3n^{\log_2 3}$. Für n die ein Vielfaches von zwei sind sogar exakt auf $n^{\log_2 3}$. Damit ist er schneller als die klassische Schulmethode und erfüllt die Forderung aus dem vorangegangenen Abschnitt.
Der zugehörige Rekursionsbaum ist in Abbildung 6 dargestellt.

1. $z_1 = (x^L + x^H) \cdot (y^L + y^H) = x^L y^L + x^H y^L + x^L y^H + x^H y^H$ (1 Multiplikation)
2. $z_2 = x^L y^L$ (1 Multiplikation)
3. $z_3 = x^H y^H$ (1 Multiplikation)
4. $z_4 = z_1 - z_2 - z_3$
5. $xy = \underbrace{z_3 2^n}_{(*)} + z_4 2^{\frac{n}{2}} + z_2$

(*) Diese Multiplikation kann durch shiften sehr effizient gemacht werden!

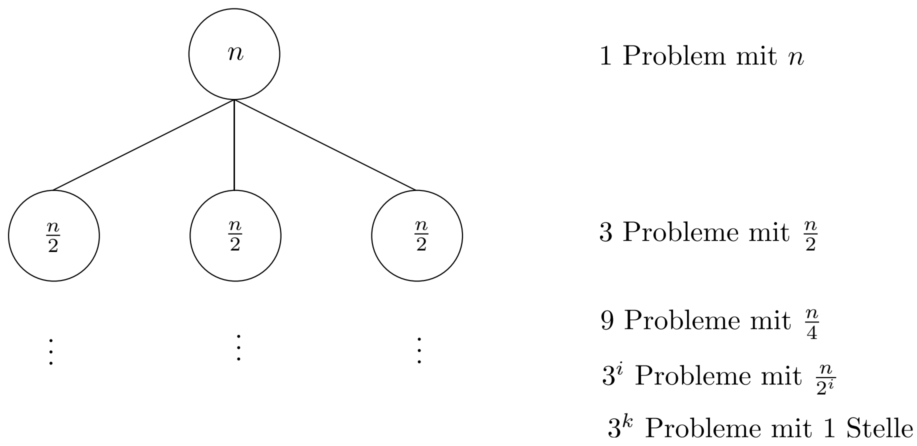


Abbildung 6: Rekursionsbaum - Algorithmus von Karatsuba

Summiert man den Aufwand pro Ebene auf, erhält man für Ebene i den Aufwand $(l\frac{3}{2})^i \cdot c \cdot n$. Summiert man die Ebenen auf, erhält man als Gesamtaufwand:

$$
\begin{aligned}
&= \sum_{i=0}^{k} \left(\frac{3}{2}\right)^i \cdot c \cdot n \\
&= c \cdot n \cdot \sum_{i=0}^{k} \left(\frac{3}{2}\right)^i
\end{aligned}
$$

Da diese Formel eine geometrische Reihe beschreibt, können wir eine endliche Partialsumme wie folgt berechnen:

$$\sum_{i=0}^{k} \left(\frac{3}{2}\right)^i = \frac{\left(\frac{3}{2}\right)^{k+1} - 1}{\frac{3}{2} - 1} = \mathcal{O}\left(\left(\frac{3}{2}\right)^k\right)$$

Für den Gesamtaufwand folgt hieraus:

$$
\begin{aligned}
\Rightarrow c \cdot n \cdot \sum_{i=0}^{k} \left(\frac{3}{2}\right)^i &= \mathcal{O}\left(c \cdot \cancel{n} \cdot \frac{3^k}{\cancel{2}^k}\right) \qquad \text{(Gilt, da } n = 2^k \text{)} \\
&= \mathcal{O}\left(3^k\right)
\end{aligned}
$$

Da $3^k = 3^{\log_2 n} = n^{\log_2 3}$ (Logarithmengesetzte), folgt:

$$\mathcal{O}\left(n^{\gamma}\right) \text{, mit } \gamma = \log_2 3 \approx 1,7$$

Teile-und-herrsche-Rekursionen

Die Strategie „Teile-und-herrsche"zerlegt ein Problem $T(n)$ in a Teilprobleme der Größe $\frac{n}{b}$. Hinzu kommt der Aufwand für das Zerlegen in die Teilprobleme und Zusammenfügen derselbigen. Allgemein kann man also folgende Formel dafür angeben:

$$T(n) = aT(\frac{n}{b}) + f(n)$$

MASTER-Theorem

Das Master-Theorem, auch Hauptsatz der Laufzeitfunktionen, kann bei vielen rekursiven Funktionen, wie sie beispielsweise bei vielen Divide and Conquer Algorithmen auftreten, eine schnelle Einordnung in Laufzeitklassen ermöglichen.

Anzahl Probleme	Größe	Aufwand	
1	n	$f(n)$	n^4
a	$\frac{n}{b}$	$af(\frac{n}{b})$	$n^4 \cdot \frac{a}{b^4}$
a^2	$\frac{n}{b^2}$	$a^2 f(\frac{n}{b^2})$	$n^4 \cdot (\frac{a}{b^4})^2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
a^k	$\frac{n}{b^k}$	$a^k f(\frac{n}{b^k})$	$n^4 \cdot (\frac{a}{b^4})^k$

$$k = \log_b n$$

$\frac{n}{b^k}$ konstant, z.B. wenn $f(n) = n^4$

3 Fälle:

1. fallende geometrische Reihe: ($\frac{a}{b^4} < 1$): $T(n) = \mathcal{O}(f(n))$
2. wachsende geometritsche Reihe: ($\frac{a}{b^4} > 1$): $T(n) = \mathcal{O}(a^k) = \mathcal{O}(n^{\log_b a})$
3. konstant: $\frac{a}{b^4} = 1$: $T(n) = \mathcal{O}(n^{\log_b a} \log n)$

4 Rekursion (Fortsetzung) (Vorlesung 5 am 31.10.)

4.1 Motivation Master-Theorem

$$T(n) = \underbrace{T(\frac{n}{b})}_{T(\lfloor \frac{n}{b} \rfloor)+\dots+T(\lceil \frac{n}{b} \rceil)} * a + f(n)$$

Für Probleme $\leq n_0$ wird das Problem irgendwie direkt gelöst.
Startbedingung: $1 \leq T(n) \leq M$ für $n \leq n_0$

In der Praxis muss man natürlich irgendwann das n_0 ausrechnen und kann nicht beliebig lange aufteilen.
Die Konstanten $a \geq 1$ und $b > 1$ müssen erfüllt sein und außerdem müssen wir fordern:

$$\begin{aligned}
\lceil \frac{n}{b} \rceil &\leq n - 1 \text{ für } n > n_0 \\
\Leftrightarrow \frac{n}{b} &\leq n - 1 \\
n(1 - \frac{1}{b}) &\geq 1 \\
n &\geq \frac{b}{b-1} \\
\Rightarrow n_0 &\geq \frac{b}{b-1}
\end{aligned}$$

sonst werden die Probleme nicht kleiner und die Rekursion kann nicht gelöst werden.

$$n \log_b n \text{ Elemente} \begin{cases} 1 \text{ Problem der Größe } n & \text{Aufwand } 1 f(n) n^k \text{ Annahme } f(n) = n^k \\ 2 \text{ Probleme der Größe } \frac{n}{b} & \text{Aufwand } a * f(\frac{n}{b}) a (\frac{n}{b})^k \\ 3 \text{ Probleme der Größe } \frac{n}{b^2} & \text{Aufwand } a^2 * f(\frac{n}{b^2}) a^2 (\frac{n}{b})^k \\ \vdots & \vdots \end{cases}$$

Beispiel: Mergesort

$$\begin{aligned}
a = b &= 2 \\
\gamma = \log_2 2 &= 1
\end{aligned}$$

4.2 Master-Theorem für divide and conquer-Rekursion

$$\begin{aligned}
a \geq 1, b > 1, M, n_0 &\geq 1 (\frac{n_0}{b} \leq n_0 - 1) \\
f(n), T(n) &\text{Funktionen auf den natürlichen Zahlen} \\
f(n) &\geq 0
\end{aligned}$$

Es gelten die Rekursionsbedingungen

$$\begin{aligned}
T(n) &\leq aT(\lceil \frac{n}{b} \rceil) + f(n) && (n > n_0) \\
T(n) &\geq aT(\lfloor \frac{n}{b} \rfloor) + f(n) && (n > n_0) \\
1 &\leq T(n) \leq M
\end{aligned}$$

Dann definieren wir den kritischen Exponenten

$$n = \log a > 0$$

(-) Wenn $f(n) = \mathcal{O}(n^{\gamma-\epsilon})$ für ein $\epsilon > 0$, dann
$T(n) = \Theta(n^{\gamma})$

(=) Wenn $f(n) = \Theta(n^{\gamma})$ ist, dann
$T(n) = \Theta(n^{\gamma} \log n)$

(+) Wenn $f(n) = \Theta(n^{\gamma+\epsilon})$ für ein $\epsilon > 0$ ist oder wenn die Reularitätsbedingung erfüllt ist
$\exists c <^{1}$:
(*) $a.f(\rceil \frac{n}{b})\lceil < c.f(n)$ für alle $n > n_0$
dann gilt: $T(n) = \Theta(f(n))$

4.2.1 Bemerkungen

1. Wenn f monoton ist, dann gelten die Schlussfolgerungen auch für beliebig gemischtes Auf- und Abrunden.
2. Mit (*) kann man auch Funktionen wie $f(n) = 2^n$ oder $f(n) = 2^{\sqrt{n}}$ erfassen.
3. $\Omega(n^{\gamma+\epsilon})$ im Fall (+) reicht leider nicht.
4. $f(n) = n \log n, \gamma = 1$ wird nicht erfasst.

4.3 Beweis: Master-Theorem

a.) Wir betrachten die oberen Schranken für die Fälle (-) und (=)

(a) Ersetze $f(n)$ durch die oberen Schranke $\underline{u}.n^k$
$f(n) \leq u.n^k$ Finde eine Funktion $P(n)$ mit $(***)P(n) \geq aP(\lceil \frac{n}{b} \lceil) + un^k$ für $n \geq n_0$
und $P(n) \geq M$ für $n \geq n_0$
Dann ergibt sich durch vollständige Induktion: $T(n) \leq P(n)$
Basis: $(n \leq n_0)$

$$\begin{aligned} T(n) &\leq aT(\lceil \frac{n}{b} \rceil) + f(n) \leq \text{(I.V.)} \\ &\leq aP(\lceil \frac{n}{b} \rceil) + f(n) \\ &\leq aP(\lceil \frac{n}{b} \rceil) + un^k \leq P(n) \end{aligned}$$

(b) Verschiebung des Definitionsbereiches, um $\lceil\rceil$ los zu werden.
$v = \frac{b}{b-1} \Rightarrow -\frac{v}{b} = 1 - v$
$P(n) = T'(n - v)$ bzw. $T'(n) = P(n + v)$
$T' ist jetzt auf \mathbb{R}_{>0}$ definiert.
Wir bestimmten dann T' so, dass
$(**)T'(n) = aT'(\frac{n}{b}) + u'n^k$ (u ist eine Konstante)

Behauptung: aus $(**)$ folgt $(***)$, falls T' monoton wächst

$$\underbrace{P(n)} \geq aP(\lceil\frac{n}{b}\rceil) + un^k$$
$$\text{L.S.} = P(n) = T'(n-v) = aT'(\frac{n}{b} - \frac{v}{b}) + u'(n-v)^k$$
$$\begin{aligned}\text{R.S.} &= aP(\lceil\frac{n}{b}\rceil) + un^k \\ &= aT'(\lceil\frac{n}{b}\rceil - v) + un^k \\ &< aT'(\frac{n}{b} + 1 - v) + un^k \\ &= aT'(\frac{n-v}{b}) + un^k\end{aligned}$$

Jetzt müssen wir nur noch u' so wählen, dass $u'(n-v)^k \geq un^k$ für $n \geq n_0 u' \geq u\frac{n_0^k}{(n_0-v)^k}$

Lösen von $(**)$ durch Ansatz:

Fall (-) $k = \gamma - \epsilon$: $T'(n) = Dn^\gamma + En^k$ Einsetzen in $(**)$

$$\begin{aligned}Dn^\gamma + En^k &= aD(\frac{n}{b})^\gamma + aE(\frac{n}{b})^k + u'n^k \\ &= Dn^\gamma \underbrace{\frac{a}{b^\gamma}}_{1} + n^k(aE\frac{1}{b^k} + u')\end{aligned}$$

$$E(1 - \frac{a}{b^k} = u', E = \frac{u'}{1 - \frac{a}{b^2}})$$
$$E(1 - \frac{b^\gamma}{b^{\gamma-\epsilon}}) = u'$$
$$E(1 - b^\epsilon) = u'$$
$$\underline{E} = \frac{-u'}{b^\epsilon - 1} < 0$$

D ist noch frei: Wähle D groß genug, dass $P(n) = T'(n-v) = D(n-v)^\gamma + E(n-v)^k \geq M$ für $n \leq n_0$ ist.

Fall (=)

$$T'(n) = Dn^\gamma + En^\gamma \log_b n$$
$\cdots \Rightarrow E = u'$, D bleibt frei. - D groß genung.

Ergebnis im Fall (-) $T(n) \leq D(n-v)^\gamma + E(n-v)^{\gamma-k} = \mathcal{O}(\backslash^\gamma)$

Ergebnis im Fall (=) $= \mathcal{O}(\backslash^\gamma \log \frac{\backslash}{\gamma})$

5 Master Theorem (Fortsetzung) (Vorlesung 6 am 3.11.)

5.1 Beweis Fortsetzung

Fall (+)

$$T(n) \leq T(\lceil \frac{n}{b} \rceil) + f(n)$$
$$T(n) \geq T(\lfloor \frac{n}{b} \rfloor) + f(n)$$
$$f(n) = \Theta(n^{\gamma+\epsilon})$$
$$\gamma = \log_b a$$
$$\text{oder: } \forall n > n_0 : \quad a.f(\lceil \frac{n}{b} \rceil) < c.f(n)$$

$c < 1$ ist eine Konstante

$$\Rightarrow T(n) = \Theta(f(n))$$

Beweis (Induktion)

untere Schranke $T(n) \geq f(n)$ (aus der Rekursion) $\Rightarrow T(n) = \Omega(f(n))$
obere Schranke: Ansatz: $T(n) \leq D.f(n)$
Versuch eines Beweises durch Induktion.
n_0 groß genug machen, dass $\frac{n_0}{b} \leq n_0 - 1 \Rightarrow \frac{n}{b} \leq n - 1 \forall n \geq n_0$
$\Rightarrow \lceil \frac{n}{b} \rceil < n$ Induktion kann funktionieren.
Induktionsschritt: $n \geq n_0$ für $i < n$ sei $T(i) \leq D.f(i)$ schon bewiesen.

$$\begin{aligned} T(n) &\leq aT(\lceil \frac{n}{b} \rceil) + f(n) \\ &\leq a.D.f(\lceil \frac{n}{b} \rceil) + f(a) \quad \text{nach I.V.} \\ &\leq D.cf(n) + f(n) \quad \text{Regularitätsbedingung} \\ &\leq D.f(n) \end{aligned}$$

$$\underbrace{Dc + 1 \leq D}_{\text{notwendig}}$$
$$\leftrightarrow D(1-c) \leftrightarrow D \geq \frac{1}{1-c}$$

Induktionsbasis: Wähle D groß genug, dass $T(i) \leq Df(i)$ für $i = 1, 2, ..., n_0 - 1$ gilt.
(Voraussetzung: $f(i) > 0$)
$D = \max\{\frac{T(1)}{f(1)}, \frac{T(2)}{f(2)}, \ldots, \frac{T(n_0)}{f(n_0)}, \frac{1}{1-c}\}$
2. Fall: $f(n) = \Theta(n\gamma + \epsilon), \epsilon > 0$

Obere Schranke (a) Ersetze $f(n)$ durch $u.n^{\gamma+\epsilon}$
Beweise, dass $f(n) = u.n^{\gamma+\epsilon}$ die Regularitätsbedingung erfüllt. (zunächst ohne Aufrunden, weil leichter).
$a.f(\frac{n}{b}) < c.f(n)$ L.S. $= a.u.(\frac{n}{b})^{\gamma+\epsilon} = \frac{a.un^{\gamma+\epsilon}}{b^{\gamma}.b^{\epsilon}}$
R.S. $= c.u.n^{\gamma+\epsilon}$

n_0 so groß wählen, dass $\frac{(\frac{n}{b}+1)^{\gamma+\epsilon}}{(\frac{n}{b})^{\gamma+\epsilon}}$ nahe genug bei 1 ist, sodass die L.S. immer noch $< cf(n)$ ist.

$\Leftarrow (1+\frac{b}{n_0})^{\gamma+\epsilon} < b^{\epsilon} \leftarrow n_0$ groß genug wählen.

5.2 Zählen von Fehlständen (Inversion)

Ein Fehlstand ist ein Paar $a_i > a_j$ mit $i > j$.
$(7, 3, 17, 12, 16, 20) = (a_1, \ldots, a_n)$
$0 \leq \#$Fehlstände $\leq \binom{n}{2}$

5.2.1 Divide and Conquer - oder - Warum Mergesort so wichtig ist!

Fehlstände können zwischen linker und rechter Hälfte leicht bestimmt werden, wenn man die beiden sortierten Listen verschmelzt.
$(15610)(2479)$ Anzahl der Fehlstände = Anzahl der Fehlstände links + Anzahl der Fehlstände rechts
$F((a_1, \ldots, a_n) \ldots$ Ausgabe: Sortierte Liste$(b_1, \ldots, b_n), k$ wobei $k = \#$Fehlstände

```
if n=0:return (a_1),0
n'=\lfloor \frac{n}{2} \rfloor; n'' = n-n'
(b_1,...,b_n),F_L = F(a_1,...,a_n')
(c1,...,c_n''),F_R = F(a_{n'+1},...,a_n)
k = F_L + F_R
i = j = 1;
for l = 1,2,...,n
  if (b_i \leq c_j or j = n'' +1) and i \leq n'
    d_l = b_i; k = k + (j-1)
    i ++
  else
    d_l = c_j
    j++
  return (d_1,...,d_n),k
```

5.2.2 Variante

Länge des Fehlstands ist $j - i(a_i > a_j, j > i)$
$p =$ Gesamtlänge alle Fehlstände; wir brauchen zusätzlich zu jeden Element die Position in der ursprünglichen Liste.
Eingabe: $a_1, \ldots, a_n$... Ausgabe ist $(b_1, \ldots, b_n), (q1, \ldots, q_n), k, p$
q_i ist die Position von b_i in der Liste $(a_1, \ldots, a_n)$... (q_i) ist eine Permutation von $(1, \ldots, n)$
Rekursive Aufrufe....
$(b_1, \ldots, b_n), (q1, \ldots, q'_n), F_L, P_L$ = rekursiv $(c_1, \ldots, c_n), (r1, \ldots, r''_n), F_R, P_R$ =

```
i = j = 1
for l = 1,...,n
  if i <= n' and (j = n''+1 or b_i \leq c_j)
    d_l = b_i
    s_l = q_i
    k = k + j -1
    // eckige klammer rechts neben die oberen 3 ausdrücken
    p = p + (j-1) (n'-q_i)+T
    // ende
    i++
  else
    d_l = c_j
    s_l = r_j + n'
```

```
    // eckige Klammer rechts neben der beiden oberen ausdrücke:
    T = T + r_j
    // ende
    j ++
return (d_1,...,d_n)(s_1,...,s_n),k,p
```

5.2.3 Laufzeit

Nach Master-Theorem:

$$\begin{aligned} T(n) = T(\lfloor \frac{n}{2} \lfloor) + T(\rceil \frac{n}{2} \lceil) + \Theta(\underbrace{n}_{n^{\gamma}(=)}) \\ a = 2, b = 2, \gamma = \log_2 2 = 1 \\ T(n) = \Theta(n \log n) \end{aligned}$$

Oft teilt das Problem auf, dass man Größen in zwei (ungefähre) gleich große Teile zerlegen möchte, einen Teil mit den kleineren Werten, und einen Teil mit den größeren Werten.
Der **Median** (= $\frac{n}{2}$) - größtes Element ist der ideale Trennungspunkt.

6 Median (Vorlesung 7 am 7.11.)

6.1 Bestimmung des k-kleinsten Elements (Medians)

Eingabe: $(a_1, a_2, \ldots, a_n)$ Liste mit Werten, k, $1 \leq k \leq n$
Bestimme das k-kleinste Element in sortierter Reihenfolge.
Sortierte Reihenfolge $a^{(1)} \leq a^{(2)} \leq \cdots \leq a^{(n)}$
Gesucht ist $a^{(k)}$; $k =$ Stelle in der sortierten Reihenfolge heißt der **Rang** des Elements
Beispiel: $(\underline{4}, 2, 1, 7, 9)$ Rang von $a_r 4$ ist 3.
Das Element, das in der Mitte steht heißt der Median.
Oft hat man versucht das Element in der Mitte zu bestimmen, in dem man alle Werte aufsummiert und dann durch die Anzahl der Werte teilt. Das Ergebnis sollte dann der Mittelwert sein. Das Problem sind allerdings Werte, die im Verhältnis zu allen anderen deutlich größer sind (bsp. $(1, 2, 3, 4, 2, 3, 9000)$), weil sie den Mittelwert ungünstig verschieben, sodass er keinen Sinn ergibt.

Der **Median** kann wie folgt bestimmt werden:
für ungerade n

$$a^{\frac{n+1}{2}}$$

für gerade n

$$\frac{1}{2}(a^{\frac{n}{2}} + a^{\frac{n}{2}+1})$$

6.2 Quickselect

Algorithmus: `Quickselect(k,l)` mit $l = (a_1, ..., a_n)$

1. Wähle Pivotelement a
2. Zähle, wie viele Elemente $<, =, > a$ sind. Der Rang von a ist zwischen $n_< + 1$ und $n_< + n_=$
3. **if** $k \leq n_<$ **then** `Quickselect(k,l_kleiner)`, wobei $|l_<| = n_<$ und `l_kleiner` enthält die Elemente $< a$.
4. **if** $k > n_< + n_=$ **then** `Quickselect(k-nkleiner-ngleich, l_groesser)`
5. return a

6.2.1 Laufzeit

Laufzeit im schlimmsten Fall: Pivotelement immer das kleinste Element oder größte.
Liste wird nur um 1 kleiner in jeder Rekursion $\rightarrow \Theta(n^2)$

Laufzeit im besten Fall:

- Rang(a)$= k$, keine Rekursion notwendig $\rightarrow \Theta(n)$ (GLÜCK!)

- Teilung in der Mitte: $n_<, n_> \leq \frac{n}{2} : T(n) = T(\frac{n}{2}) + \Theta(n)$ Ein bisschen Mastertheorem:

$$
\begin{aligned}
T(n) &= 1 * T(\frac{n}{2}) + \Theta(n) \\
a &= 1 \\
b &= 2 \\
f(n) &= \Theta(n^1) 1 > 0 \\
\gamma &= \log_b a = 0 \rightarrow \text{ Fall(+) } \Rightarrow T(n) = \Theta(f(n)) = \Theta(n)
\end{aligned}
$$

Alternative (Einsetzen:)

$$
T(n) = \Theta(n) + \Theta(\frac{n}{2}) + \Theta(\frac{n}{4}) \cdots = \Theta(n)
$$

Der Algorithmus ist also stark davon abhängig welches Pivotelement wir wählen. Ideal wäre es den Median zu finden. Da wir aber hier versuchen den Median zu finden ist das ein Zirkelschluss. Dabei muss es nicht mal genau das Element genau in der Mitte sein, es reicht, wenn es nahe genug dran ist.

6.3 randomisiertes Quickselect

Wähle a zufällig aus der Liste.
Rang(a) ist gleich verteilt auf $1, 2, ..., n$.

6.3.1 Laufzeit

Analyse der erwarteten Laufzeit:
Wir nennen den Aufruf von Quickselect erfolgreich, wenn:

$$
\begin{aligned}
n_< + n_= &= \frac{1}{4}n \\
n_> + n_= &= \frac{1}{4}n
\end{aligned}
$$

in der obersten Aufrufebene ist.

$$
[\frac{1}{4}n \leq \text{rang}(a) \leq \frac{3}{4}n]
$$

wenn (a) eindeutig ist.
Wahrscheinlichkeit(erfolgreich) $\geq \frac{1}{2}$
Bei einem erfolgreichen Aufruf wird die Liste auf höchstens $\frac{3}{4}n$ reduziert.
$T(n) =$erwartete Laufzeit.

$$
\begin{aligned}
T(n) &\leq E(\#\text{Läufe bis zu einem erfoglreichen Lauf}).\mathcal{O}(n) + T(\frac{3}{4}) \\
&= \frac{1}{p} \text{ wobei } p = \frac{1}{2} \text{ die Erfolgswahrscheinlichkeit ist.} \\
&= \leq 2
\end{aligned}
$$

$$T(n) \leq T(\frac{3}{4}n) + \mathcal{O}(n) \Rightarrow T(n) = \mathcal{O}(n)$$

6.4 Quickselect nach Blum, Floyd, Pratt, Rivest, Tarjan (1973)

Determinitische Auswahl in $\mathcal{O}(n)$ Zeit.

1. Falls $n \leq n_0$, sortiere
2. Andernfalls zerlege Folge in 5er-Gruppen und bestimme in jeder Gruppe den Median $m_1, m_2, ...m_{\lfloor \frac{n}{5} \rfloor}$
3. Bestimme den Median m^* dieser Mediane rekursiv.
4. Wähle das Pivotelement $a := m^*$ und verfahre weiter wie bei Quickselect.

6.4.1 Laufzeit

Welche Aussagen treffen jetzt auf $n_< + n_=$ und $n_> + n_=$zu?

$$
\begin{aligned}
n_< + n_= &\geq 3\frac{\lfloor \frac{n}{5} \rfloor}{2} \\
n_> + n_= &\geq 3\frac{\lfloor \frac{n}{5} \rfloor}{2} \\
n_< &= n - (n_< + n_=) \\
&= n - \frac{3}{2} * \lfloor \frac{n}{5} \rfloor
\end{aligned}
$$

Annahme $n = 5l$

$$
\begin{aligned}
n_< &\leq n - 0,3 = 0,7n \\
n &= 5l + i \qquad\qquad |i = 0,1,2,3,4 \quad l = \lfloor \frac{n}{5} \rfloor \\
n_< &\leq n - \frac{3}{2}l = n - \frac{3}{2}(\frac{n-i}{5}) \\
&= n - \frac{3}{10}n + \frac{3}{10}i \\
&\leq \frac{7}{10}n + \frac{12}{10} \leq \frac{7}{10}n + 3 \\
n_< &\leq \frac{7}{10}n + 3
\end{aligned}
$$

Behauptung: $T(n) = \mathcal{O}(n)$
Beweis: Annahme:

$$T(n) \leq \mathcal{C}n + T(\lfloor \frac{n}{5} \rfloor) + T(\lfloor 0,7n \rfloor + 3) \text{ für } n \geq 100$$

Behauptung $T(n) \leq \mathcal{C}'n$, wenn $\mathcal{C}' \geq 20C$ ist und $\mathcal{C}'$ so groß ist, dass $T(n) \leq \mathcal{C}'n$ für $n \geq 100$ ist.
Beweis mit vollständiger Induktion: $n \geq 100$ geht!
für $n > 100$:

$$
\begin{aligned}
T(n) &\leq \mathcal{C}n + T(\lfloor \frac{n}{5} \rfloor) + T(\lfloor 0,7n \rfloor + 3) \\
&\leq \mathcal{C}n + \mathcal{C}' \frac{n}{5} + \mathcal{C}' * 0,7n + \mathcal{C}' 3 \\
&\leq \mathcal{C}' \frac{n}{20} \\
&\leq \mathcal{C}' n(0,05 + 0,2 + 0,7) + \mathcal{C}'.3 \\
&= \mathcal{C}'(0,95n + 3) \leq \mathcal{C}'n
\end{aligned}
$$

$$
\begin{aligned}
0,95n + 3 &\leq n \\
3 &\leq n.0,05 \quad (n \geq 100 \rightarrow n0,05 \geq 5)
\end{aligned}
$$

7 Das Rucksackproblem **(Vorlesung 8 am 10.11.)**

Gegeben sind n Gegenstände.
Jeder Gegenstand hat einen Wert w und ein Gewicht g_i.
Es gibt eine Gewichtsschranke G.

Problem

Finde eine Teilmenge mit möglichst großem Wert und Gesamtgewicht $\leq G$

Beispiel n = 5, G = 12

i	1	2	3	4	5
g_i	4	3	5	2	6
w_i	7	8	6	3	9

Formulierung mit Variablen:

$$\begin{aligned} \text{maximiere} \quad & \sum_{i=1}^{n} x_i w_i && |x_i \text{ gibt an, ob Gegenstand ausgewählt wird} \\ \text{unter} \quad & \sum_{i=1}^{n} x_i g_i \leq G \\ & x_i \in \{0,1\} \end{aligned}$$

$\rightarrow 2^n$ Möglichkeiten.

- ganzzahliges RP: $x_i \in \mathbb{N}$
- gebrochenes RP: $0 \leq x_i \leq 1$

7.1 Lösung: Dynamisches Programmierung / Optimierung

Löse das Problem durch *systematisches Lösen von Teilproblemen.*
Große Teilprobleme werden auf kleinere zurückgeführt, die man schon vorher gelöst hat.

Teilprobleme?

Betrachte nur die ersten i Gegenstände.
zusätzlich: muss man das zulässige Gesamtgewicht variieren.

$$\begin{aligned} f(i,b) &= \text{optimaler Wert mit den ersten } i \text{ Gegenständen und das Gesamtgewicht } \leq b \\ &= \max\{\sum_{j=1}^{i} w_j x_j | \sum_{j=1}^{i} g_j x_j \leq b, x_j \in \{0,1\}\} \end{aligned}$$

$$\begin{aligned} f(i,b) &= \max\{f(i-1,b), f(i-1,b-g_i) + w_i, \text{falls } b \geq g_i\} \\ &= f(i-1,b), \text{falls } g_i > b \end{aligned}$$

Lösung mit Tabelle: $f(i,b)$ mit $i = 0, ..., n$ und $b = 0, ..., G$

g_i		4	3	5	2	6
i	0	1	2	3	4	5
$b = 0$	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	8	8	8	8
4	0	7^+	8	8	8	8
5	0	7	8	8	11	11
6	0	7	8	8^-	11	
7	0	7	15^+	15^-	15	
8	0	7	15	15	15	
9	0	7	15	15	18	
10	0	7	15	15	18	
11	0	7	15	15	18	
12	0	7	15	21^+	21^-	21^-

Mit $^+$ markierte Einträge in der Tabelle werden zur optimalen Gesamtlösung hinzugefügt.
$x_5 = 0 \rightarrow x_4 = 0 \rightarrow x_3 = 1 \rightarrow x_2 = 1 \rightarrow x_1 = 1$ Tabelle liefert $f(5,12) = 21 = f(n,G)$ den Wert der Optimallösung.
Um die Lösung selbst zu finden, müssen wir zurückverfolgen, wie dieser Wert zustande gekommen ist.

Zurückverfolgen der Lösung

a) man merkt sich bloß die Tabelle und rechnet beim Zurückgehen jeden Eintrag noch einmal nach. (Programmieraufwand)

b) man speichert sich schon beim Berechnen Zusatzinformationen, wie der Wert zustande gekommen ist. (viel zusätzlicher Speicheraufwand)

7.1.1 Laufzeit und Speicherbedarf

$\Theta(nG) =$ Größe der Tabelle =Speicherbedarf
Der Speicher lässt sich auf $\Theta(G)$ reduzieren (allerdings verliert man die Möglichkeit der Rücknachvollziehbarkeit)

7.2 Dynamische Programmierung

- Definition der Teilprobleme
  nicht eindeutig vorgegeben.

- Rekursion (+ Anfangsbedingungen)
  Variante mit Gesamtgewicht $= b$
  ($f(i,b) = -\infty$ falls es keine Lösung gibt.)
  (Rekursion bleibt unverändert, Anfangsbedingung ändert sich. Optimallösung in der ganzen Spalte suche)

- systematisches Ausfüllen(Zeilen- oder Spaltenweise) der Tabelle aller Teilprobleme

- Rückverfolgen der Lösung

7.3 Die Tabelle als Netzwerk

Betrachte die Tabelle als gerichteten Graphen. Jeder Eintrag = 1 Knoten.
Vorgänger = Einträge, von denen der Knoten abhängt.
Kantengewicht = Wert, der in Rekursion addiert und Knotenwert = $f(i, b)$ = Längster Weg von der linken oberen Ecke $(0, 0)$ zum Knoten (i, b).
Sehr oft lässt sich eine DP-Rekursion als Wegeproblem in einem azyklischen Graphen modellieren. (kürzeste / längste Wege von einer Ecke zur anderen)

7.4 Speicheroptimierung

Der Speicher lässt sich optimieren(?) in dem man einen Faktor $\log n$ zur Laufzeit hinzufügt.
(unklar...)

8 Dynamische Programmierung (Fortsetzung)(Vorlesung 9 am 14.11.)

8.1 Gewichtete Intervallauswahl

Gegeben: n Intervalle $[a_1, b_1), [a_2, b_2), ..., [a_n, b_n)$ mit Gewichteten $w1, ..., w_n$
Gesucht: Disjunkte Intervalle mit dem größten Gesamtgewicht.
Teilprobleme: Betrachte nun Intervalle, die in $(-\infty, x)$ enthalten sind, $x \in \mathbb{R}$. Für x reicht es, die Intervallendpunkte zu betrachten.

8.1.1 Vorverarbeitung (Normalisierung)

Sortiere alle Intervallendpunkte, ändere die vorkommenden Werte in $1, 2, 3, ..., 2n, \Rightarrow\ x \in \{0, 1, 2, ..., 2n\}$. Das Problem wird dadurch nicht verändert.
$f(i) =$ größtes Gewicht einer Menge disjunkter Intervalle die in $(-\infty, i)$ enthalten sind.
$f(i) = \max\{f(i-1), \max\{f(a_k) + w_k |$Intervalle k mit $b_k = i\}\}$
$f(0) = 0$
Berechne $f(1), f(2), .., f(n)$ mit der Rekursionsformel. $f(m)$ ist die Optimale Lösung.

8.1.2 Laufzeit

Jedes Intervall $[a_k, b_k)$ kommt genau $1x$ in der Rekursion auf der rechten Seite vor.

8.1.3 Algorithmus

1.($\mathcal{O}(n \log n)$) Sortieren und Umnummerieren der Endpunkte

2.($\mathcal{O}(n)$) Erstelle für $i = 1, .., m$ eine Liste L_i der Intervalle k mit $b_k = i$, Initialisiere alle $L_i = \emptyset$

```
for k = 1,..n
  L_b_k.append(k)
```

3.) Rekursion:

```
f(i) := max(f(i-1),max(f(a_k)+w_k)|k\in L_i)
```

f wird in einem Feld gespeichert.

8.2 Rundreiseproblem (Traveling Salesperson Problem[TSP])

Gegeben ist ein gerichteter Graph mit Kantengewichten.
Gesucht ist ein Kreis, der jeden Knoten genau einmal besucht und geringste Gesamtlänge hat (Hamiltonkreis).
Mögliche Lösungen: Startknoten beliebig fixieren.
$(n-1)(n-2)(n-3) * ... * 2 * 1 = (n-1)!$
falls der Graph vollständig ist.
Teilprobleme $(T, i) T \subseteq \{1, ..., n\}, i \in T, 1 \in T$
$f(T, i) =$ der kürzeste Weg von 1 nach i der genau die Knoten in T besucht.

8.2.1 Rekursion

$$
\begin{aligned}
f(T,i) &= \underbrace{\min}_{j \in T-\{i\}, j, i \in E, j \neq 1} (f(T-\{i\},j)c_{ji}) && \text{, für } |T| \geq 3 \\
f(\{1,i\},i) &= c_{1i} (\text{bzw. } \infty \text{, falls } 1i \notin E) \\
\text{OPT} &= \min_{j \neq 1, jn \in E} (f(\{1,...,n\},j) + c_{jn})
\end{aligned}
$$

Wieviele Teilprobleme gibt es?

$$\#\text{Teilprobleme} \leq 2^n.n$$

$2^n - 1$ Teilmengen T
Zu T gibt es $|T| - 1 Teilprobleme(T,i)$ ($\sum((_n)k)(k-1)$)
Jedes Teilproblem benötigt $\mathcal{O}(n)$ Zeit (eigentlich $\mathcal{O}(|T|)$)
Insgesamt $\mathcal{O}(2^n n^2)$ Laufzeit
Speicher $\mathcal{O}(2^n n)$
(exponentiell viel besser als $\mathcal{O}((n-1)!)$)

8.2.2 1. Möglichkeit

Tabelle mit $2^{n-1} \times n$ Einträgen. Teilprobleme werden z.B. niht wachsendem $|T|$ gelöst.(Andere Möglichkeit: T als $(n-1)$-stellige Binärzahl darstellen, in nummerischer Reihenfolge lösen.) Wichtig: $T \leq S$ und T vor S lösen.

2. Möglichkeit

Tabellieren (Memoization)
Top-down-Berechnung rekursiv nach Bedarf mit Speicher, der schon berechneten Ergebnisse.
Initialisieren der Tabelle M auf -1 (Annahme $c_{ij} \geq 0$)

```
def f (T,i):
  if M[T,i] != -1: return M[T,i]
  berechne E = f(T,i) nach der Rekursionsgleichung rekursiv.
  M[T,i] = E
  return E
```

Man kann sich überlegen, dass genau die Teilprobleme gelöst und gespeichert werden, für die es einen Weg von i nach 1 gibt, der gewanderte Knoten $\{1,...,n\} - T$ als Zwischenknoten besucht. Wenn der Graph wenige Knoten enthält, dann können das viel weniger als $2^n n$ Teilprobleme sein.

Verwendung einer Hashtabelle für M

In der Praxis sind RRP mit bis zu 10.000 Ständen bis zur Optimalität lösbar und größere genügend gut approximierbar. Ein Ansatz ist branch-and-bound (Systematisches Durchsuchen von Lösungsbäumen)

9 Dynamische Programmierung (Fortsetzung)(Vorlesung 10 am 17.11.)

9.1 Optimale Triangulierung eines konvexen Polygons

$P = (p_1, p_2, ..., p_n)|p_i \in \mathbb{R}^2 s$ ein Polygon in der Ebene. (konvex: Alle Innenwinkel $< 180°$)

9.1.1 Triangulierung

Zerlegung einer Polygonfläche in Dreiecke durch Diagonale Strecken $p_i p_j$, die im inneren verlaufen. Diagonalen dürfen sich nicht kreuzen. (erster Vorverarbeitungsschritt für viele geometische Algorithmen)
kürzeste Triangulierung = kleinste Gesamtlänge aller Diagonalen
Teilprobleme: f_{ij} =kürzeste Triangulierung des Polygons $p_i, p_{i+1}, ..., p_j$ für $1 \leq i < j \leq n$
Also... $j = i + 2$... Dreieck.
$j = i + 1$... Zweieck?!
Rekursion!

$$\begin{aligned}
&f_{ij} \to \text{ besteht aus einem Dreieck} p_i p_k p_j && | i < k < j \\
&\quad \text{optimale Triangulierungen } (p_i, ..., p_k) \text{ und } (p_k, ..., p_j) \\
&f_{ij} = \min\{f_{ik} + f_{kj} + ||p_i - p_k|| + ||p_k - p_j|| \quad | \; i < k < j\} \\
&\quad 1 \leq j, j \leq n, j \geq i + 2
\end{aligned}$$

Damit die Formel auch für $k = +1$ oder $k = j - 1$ stimmt, müssen wir $f_{i,i+1} = -||p_i - p_{i+1}||$ setzen.
Beispiel: $f_{3,5} = f_{34} + f_{45} + ||p_3 - p_4|| + ||p_4 - p_5|| = 0$ Endergebnis = f_{1n}

9.1.2 Laufzeit

$\binom{n}{2} = \mathcal{O}(n^2)$ Teilprobleme, jedes Teilproblem benötigt $\mathcal{O}(n)$ Zeit.
$\to \mathcal{O}(n^3)$ Laufzeit, $\mathcal{O}(n^2)$ Speicher.

9.2 Isotone Regression

Zum Vergleich lineare Regression. Messwerte (x_i, y_i) gesucht ist eine Gerade $y = ax + b$, sodass $\sum_{i=1}^{n} |y_i(ax_i + b)|$ oder $\sum(y_i - (ax_i + b))^2$
Gegeben ist eine Folge von Messwerten: $a_1, a_2, ..., a_n \in \mathbb{R}$
Gesucht ist eine monoton wachsende Folge: $x_1 \leq x_2 \leq ... \leq x_n$, die $\sum_{i=1}^{n} w_i * |x_i - a_i|$ minimiert.($w_i > 0$ sind gewichtete Daten.)

9.2.1 Teilprobleme

$$\begin{aligned}
&f_k(z) = \min\{\sum_{i=1}^{k} w_i|x_i - a_i| : x_1 \leq x_2 \leq ... \leq x_k = z\} && k = 0, ..., n; z \in \mathbb{R} \\
&f_k(z) = \min\{f_{k-1}(x) : x \leq z\} + w_k|z - a_k| && k \geq 1, z \in \mathbb{R} \\
&f_0(z) = 0 \quad \text{für alle } z
\end{aligned}$$

Beispiel

$$a_1 = 5, w_1 = 1,3$$
$$f_1(z) = \min\{f_0(x)|x \leq z\} + 1.3|z-5|$$
$$f_2(z) = \underbrace{\min\{f_1(x)|x \leq z\}}_{g_1(z)} + 0,7 * |z-1|$$
$$g_2(z) = \min\{f_1(x)|x \leq z\}$$

Lemma

(a) $f_k(z)$ ist eine stückweise lineare konvexe Funktion

(b) Die Knicke liegen an einr Telmenge der Eingabewerte $a_1, ..., a_k$

(c) Die Steigung des ersten Stücks ist $-w_1 - w_2 - ... - w_k$

(d) Die Steigung des letzten Stücks ist w_k

Beweis durch Induktion.
Wie kommt man von f_{k1} auf g_{k-1}? Lösche alle aufsteigende Stücke und ersetze sie durch ein horizontales Stück.
Möglichkeiten der Speicherung einer stückweise linearen Funktion: Koordinaten der Knicke $+$ Steigung des ersten und letzten Astes $\rightarrow \mathbb{O}(n)$ Werte.
$\rightarrow$ Addition zwei solcher Funktionen: $\mathcal{O}(n)$

9.2.2 Optimallösung

Wie bestimmt man die Optimallösung?
Das Minimum von $f_{k-1}(z)$ sei an der Stelle p_{k-1}
Optimalwert x^*_{k-1}, wenn x^*_k gegeben ist.
$x^*_{k-1} = \min\{p_k, x^*_k\}$

10 Dynamische Programmierung (Fortsetzung)(Vorlesung 11 am 21.11.)

Nachtrag: Isotone Regression

$$f_k(z) = \min\{\sum_{i=1}^{k} w_i|x_i - a_i| : x_1 \leq x_2 \leq ... \leq x_{k-1} \leq x_k = z\}$$

$$f_k(z) = \underbrace{\min\{\ f_{k-1}(\underbrace{x}_{x_{k-1}})|\underbrace{x}_{x_{k-1}} \leq \underbrace{z}_{x_k}\}}_{g_{k-1}(z)} + w_k|a_k - z|$$

$g_{k-1}(z)$: 2 Fälle:

$z \leq p_{k-1} \Rightarrow\ g_{k-1}(z) = f_{k-1}(z)$; der optimale Wert von $x = x_{k-1}$ bei gegeben Wert von $z = x_k$ ist z selbst.

$z \geq p_{k-1} \Rightarrow$ der optimale Wert von x ist p_{k-1} $g_{k-1}(z) = f_{k-1}(p_{k-1})$

Wenn x_k^* der optimale Wert von x_k ist, dann ist der optimale Wert von x_{k-1}:

$$x_{k-1}^* = \min\{x_k^*, p_{k-1}\}$$

Darstellung einer stückweise linearen stetigen Funktion durch Differenzen von Steigungen.
Knick an der Stelle x_0 mit Knickwert $s' - s$
Bei Addition einer linearen Funktion bleibt der Knickwert unverändert! (und Knickstelle)

Die Funktion wird durch eine Folge von Knickwerten dargestellt. Jeder Knick ist ein Paar (Knickstelle, Knickwert)
Darstellung ...

10.1 Algorithmus

Übergang von f_{k-1} zu g_{k-1}:
(x, Δ) sei der rechteste Knick:

```
while s - \Delta \geq 0:
  s:=s-\Delta
  lösche den rechtesten Knick
p_{k-1}:=x (min von f_{k-1} gefunden)
neuen Knickwert des rechtesten Knicks = \Delta-s
s:=0
```

Übergang von g_{k-1} zu f_k:

```
Füge einen zusätzlichen Knick (a_k, 2_w_k) ein.
```

Die Knicke können als Prioritätswarteschlange Q gespeichert werden. nach Schlüsselwert x geordnet.

```
Q = empty, s=0
for k = 1,...,n:
  Q.insert((a_k,2w_k))
  s = s + w_k
  (x,\Delta) := Q.findmax()
  while s - \Delta \geq 0:
    s = s - \Delta
    Q.deletemax()
```

```
    (x,\Delta) := Q.findmax()
  p_k = x
  ersetze \Delta in Q.findmax() durch \Delta-s
  s = 0
(Berechnung der Optimallösung)
x_n = p_n
for k = n-1, n-2, ... , 1
  x_k = min(x_{k-1}, p_k)
```

11 Der optimale Suchbaum

Schlüssel:

```
  AARON, p_1
  KLAUS, p_3
  DIETER, p_2
```

Mögliche Suchbäume:

Schlüssel $x_1 < x_2 < ...x_n$
Anfragehäufigkeiten $p_1, p_2, ..., p_n$ für die Schlüssel
und $q_0, q_1, ..., q_n$ für die Intervalle zwischen den Schlüsseln

Mittlere gewichtete Weglänge:

$$= \sum_{i=1}^{n} p_i \underbrace{(\text{Tiefe des Knotens mit Schlüssel} i)}_{\#\text{ Vergleiche } -1} + \sum_{i=0}^{n} q_i \underbrace{(\text{ Tiefe des Blattes, dass dem entsprechenden Intervall entspricht})}_{\#\text{Vergleiche}}$$

soll minimiert werden.

Ansatz: q_i entspricht dem Intervall (x_i, x_{i+1}) $x_0 = -\infty, x_{n+1} = +\infty$
Ein Teilbaum in einem Suchbaum entspricht einem Interval (x_i, x_j mit $0 \leq i \leq j \leq n+1$)
(Der Baum wird genau dann betreten, wenn der gesuchte Schlüssel in diesem Intervall liegt.)

11.1 Teilprobleme

$f(i,j)$ $0 \leq i \leq j \leq n+1$ optimaler Suchbaum für Schlüssel $x_{i+1}, ..., x_{j-1}$ und Häufigkeiten $p_{i+1}, ..., pj - 1$

11.2 Rekursion

$$f(i,j) = \min\{f(i,k) + f(k+1,j)\} + q_i + q_{i+1} + ... + q_{j-1} + p_{i+1} + p_{i+2} + \\ ... + p_{j-1} - p_k \qquad |i+1 \leq k \leq j-1, 0 \leq i, j \leq n+1, j \geq$$

11.3 Anfangswerte

$$f(i, i+1) = 0, i = 0, ..., n$$

$f(2,4) = f(2,3) + f(3,4) + q_2 + q_3 + p_3 - p_3$

11.4 Gesamtlösung

$f(0, n+1)$

11.5 Laufzeit

$\mathcal{O}(n^2)$ Teilprobleme.
$\mathcal{O}(n)$ Ausdrücke, über die minimiert wird.
Die Summe $q_1 + ... + p_{i-1}$ ist fest, für jedes Teilproblem nur einmal ausrechnen $\rightarrow \mathcal{O}(n)$ Zeit.
$\Rightarrow \mathcal{O}(n^3)$ Zeit, $\mathcal{O}(n^2)$.

12 Dynamische Programmierung (Einen hab ich noch!)(Vorlesung 12 am 24.11.)

12.1 Dynamische Programmierung - eine Zusammenfassung

Bisher gelöste Probleme mit dynm. Prog.:

1. Rucksackproblem
2. optimale Triangulierung $\mathcal{O}(n^3)$
3. Suchbaum $\mathcal{O}(n^3)$
4. CYK (zweites Semester GTI) - Coche, Younger, Kasamy
   Eine Grammatik ist in Chomsky Normalform (CNF), wenn jede jede Formel einer Grammatik in ein Terminalsymbol "mündet".
   Ziel ist es für ein gegebenes Wort zu prüfen, ob es von der gegeben CNF Grammatik erzeugt werden kann.
   $f(i,j) = \{v|v \rightarrow^s xa_i, ..., a_j\}|a_i, a_j \in \Sigma^*$

12.2 Editierabstand

Wir nehmen an, dass wir folgendes Wort auf der Tastatur getippt haben:
A g o r h y t m u s
Aber eigentlich wollten wir Pfannku... ähm **Algorithmus** schreiben.
Wie kommen wir jetzt von Agorhytmus zu Algorithmus?

12.2.1 Problem

Gegeben: zwei Wörter $A = a_1, ..., a_m$ und $B = b_1, ..., b_n$ aus Σ^*
Wie können wir A in B durch folgende Operationen verwandeln?

1. löschen eines Buchstabens (Kosten k_L)
2. einfügen eines Buchstabens (k_E)
3. Buchstabe u durch x ersetzen (δ_{ux})

Gesucht ist die billigste Folge von Operationen wie A in B umgewandelt wird.
Die **Kosten** k_L, k_E, δ_{ux} sind der **Editierabstand**.
Vor allem in der Bioinformatik ist das ein sehr wichtiges Problem auf sehr großen Datenmengen.

12.2.2 Teilprobleme

$f(i,j) =$ Editierabstand zwischen $a_1, ..., a_i$ und $b_1, ..., b_j$

12.2.3 Rekursion

$$f(i,j) = \min\{f(i-1,j) + k_l, f(i,j-1) + k_E, f(i-1,j-1) + \delta_{a_i,b_j}\} \quad |1 \leq i \leq m, 1 \leq j \leq n$$

12.2.4 Startwerte

Konvention: $\delta_{xx} = 0 \forall x \in \Sigma$

$$
\begin{aligned}
f(i,0) &= f(i-1,0) + k_L = i * k_L \\
f(0,0) &= 0 \\
f(0,j) &= jk_E
\end{aligned}
$$

12.2.5 Graph

Graph mit $(m+1)(n+1)$ Knoten. Jeder Knoten hat 3 Vorgänger (außer am Rand). Editierabstand ist der kürzeste Weg von $(0,0)$ zu (m,n)

12.2.6 Annahmen

Zwei aufeinanderfolgende Änderungen lohnen sich nicht: x→y→u (Dreiecksungleichung x: $\delta_{xy} + \delta_{yu} \geq \delta_{xu}$)
Löschen und Einfügen statt Ändern ist in diesem Algorithmus vorgesehen. Falls $\delta_{xy} \leq k_L + k_E$

12.2.7 Speicherreduktion

auf $\mathcal{O}(m+n)$ durch devide & conquer.

1. Zur Berechnung der <u>Kosten</u> $f(m,n)$ ist nun $\mathcal{O}(m+n)$ Speicher notwendig: Es genügt, zwei aufeinanderfolgende Spalten im Speicher zu halten.
2. Abstände zum Zielknoten:
$$g(i,j) = \min\{g(i+1,j) + k_L, g(i,j+1) + k_E, g(i+1,j+1) + \delta_{a_{i+1},b_{j+1}}\}$$
3. Optimallösung =
$$\min\{f(i, \lfloor\frac{n}{2}\rfloor), g(i, \lfloor\frac{n}{2}\rfloor)|i = 0..m\}$$
4. Es sei i_0 das Optimum in 3.
Bestimme rekursiv den kürzesten Weg von $(0,0)$ zu $(i_0, \lfloor\frac{n}{2}\rfloor)$ und von $(i_0, \lfloor\frac{n}{2}\rfloor)$ zu (m,n)

⇒ Speicherbedarf $\mathcal{O}(m+n)$

12.3 Laufzeit (inkl. Speicherreduktion)

Rekursion: $T(m,n) = O(m,n) + T(i_0, \lfloor\frac{n}{2}\rfloor) + T(m - i_0, n - \lfloor\frac{n}{2}\rfloor)$
n sei eine zweier Potenz:

$$
\begin{aligned}
T(n,m) &\leq cmn + T(i, \frac{n}{2}) + T(m-i, \frac{n}{2}) \\
&\leq cmn + \min\{T(i, \frac{n}{2}) + T(m-i, \frac{n}{2})|0 \leq i \leq m\}
\end{aligned}
$$

Behauptung: $T(m,n) \leq c'mn$; Beweis durch Induktion
Einsetzen: R.S.

$$cmn + c'i * \frac{n}{2} + c'(m-i)\frac{n}{2}$$
$$= cmn + \frac{c'}{2}mn \leq c'mn$$

wähle $c' = 2c$, dann gehts.

13 Gierige Algorithmen

Optimierungsalgorithmen, die eine Folge von Entscheidungen kurzsichtig treffen und später nicht mehr rückgängig machen.

13.1 Beispiel Rucksack

Rucksackproblem $g_1, ..., g_n G$ mit $w_1, ..., w_n$

1. Wählt die wertvollsten Gegenstände zuerst, bis Rucksack voll ist.
   NICHT OPTIMAL!

g_i	10	1	1	$\dots$	10
w_i	100	99	99	$\dots$	

2. Sortiere Gestenstände $\frac{w_1}{g_1} \geq ... \geq \frac{w_n}{g_n}$ und wähle gierig in dieser Reihenfolge. **NICHT OPTIMAL!**

g_i	8	5	5	$\dots$	G = 10
w_i	9	5	5	$\dots$	
$\frac{w_i}{g_i}$	$\frac{9}{8}$	1	1		

Bemerkung: Algorithmus 2. ist optimal für das gebrochene Rucksackproblem!
Wir sehen, dass Greedy-Algorithmen nicht immer die beste Lösung für gewisse Probleme sind.

13.2 Umgewichtete Intervallauswahl

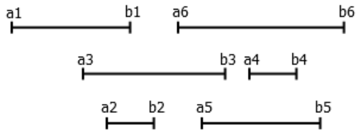
$[a_1, b_1)...[a_n, b_n)$, keine Gewichte w_i
Wähle eine möglichst große **Anzahl** von undisjunkten Intervallen aus.

1. Wähle kürzeste Intervalle zuerst. (NICHT OPTIMAL)
2. Intervalle von links nach rechts. (NICHT OPTIMAL)
3. Intervall, das am wenigsten andere überlappt.
4. sortiert nach Endzeitpunkt b_i

14 Greedy Algorithmen (Vorlesung 13 am 28.11.)

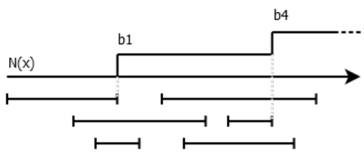
14.1 Intervallauswahl nach Endzeitpunkten

Greedy-Algorithmus Sortiere die Intervalle nach Endzeitpunkt $b_1 \leq b_2 \leq ... \leq b_n$



r = $-\infty$ für $i = 1, ..., n$
Wenn Intervalle $[a_i, b_i)$ keine ausgewähltes Intervall überlappt(`if a_i \geq r`), wähle es aus (`r = b_i`).
Wir müssen uns den rechtesten Punkt merken, um zur optimalen Lösung zu kommen... r = der rechte Endpunkt der bisher ausgewählten Intervalle.

14.1.1 Beweis

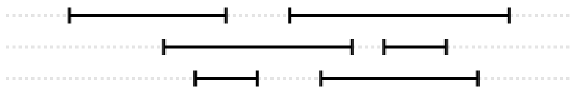


$\mathcal{N}(x) = \#$der gewählten Intervalle innerhalb $(-\infty, x)$ bei der Greedylösung.
$\mathcal{N}^*(x) = \#$der gewählten Intervalle bei einer beliebigen anderen Lösung.
Wir wollen zeigen, dass $\mathcal{N} \geq \mathcal{N}^*$ für alle $x. \mathcal{N}(\infty) \geq \mathcal{N}(\infty)$
$\mathcal{N}(x)$ kann sich nun an einem Punkt b_i ändern.
Beweis mit Induktion nach $i. \mathcal{N}(b_i) \geq \mathcal{N}^*(b_i)$
Induktionsbehauptung: $\mathcal{N}(-\infty) = \mathcal{N}^*(-\infty) = 0$
Annahme: $\mathcal{N}^*(x)$ macht an der Stelle b_i einen Sprung: $\mathcal{N}^*(b_i) = \mathcal{N}^*(b_{i-1}) + 1$.
Die andere Lösung wählt das Intervall $[a_i, b_i)$ aus.
$\mathcal{N}^*(b_i) = \mathcal{N}^*(a_i) + 1 \leq \mathcal{N}(a_i) + 1$
Im Intervall $[a_i, b_i)$ hat der Greedy Algorithmus ebenfalls ein Intervall gewählt, das zu den bisherigen Intervallen disjunkt ist. (spätestens das Intervall $[a_i, b_i)$ ist so ein Kandidat. $\Rightarrow \mathcal{N}(b_i) \geq \mathcal{N}(a_i) + 1 \leq \mathcal{N}(b_i)$
Annahme (Fall2): $\mathcal{N}^*(x)$ macht keinen Sprung bei b_i

Im Vergleich zum Algorithmus der dynamischen Programmierung werden hier Vereinfachungen vorgenommen.

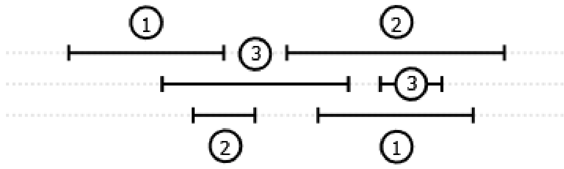
14.2 Variante

Wir müssen alle Intervalle akzeptieren und sie verschiedenen Maschinen zuordnen, wenn sie sich überlappen.
Gesucht ist die minimale Anzahl von Maschinen.
Beispiel: Es sind 3 Maschinen notwendig.
Untere Schranke, Anzahl Intervalle die einem gemeinsamen Punkt x enthalten.



14.3 Greedy Algorithmus

Betrachte die Intervalle aufsteigend vom Startpunkt $a_1 \leq a_2 \leq ... \leq a_n$
Ordne jedes Intervall $[a_i, b_i)$ der Maschine mit der kleinsten Nummer j zu, die frei ist.

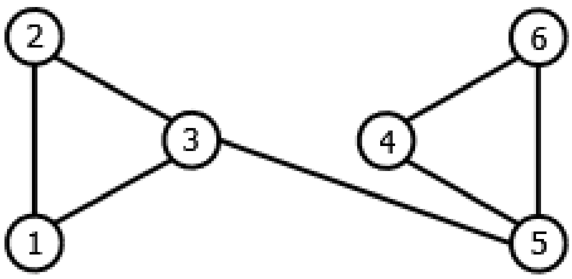


14.3.1 Beweis

Behauptung: Wenn die Maschine j zugeordnet wird, ist der Punkt a_i in mindestens j Intervallen enthalten.
Die Maschinen $1, 2, ..., j-1$ sind belegt und dabei in anderen Intervallen enthalten. plus Intervall $[a_i, b_i)$

14.4 Interpretation als Graphenproblem

Intervalle $\rightarrow$ Knoten
überlappende Intervalle $\rightarrow$ Kanten
Der entstehende Graph ist ein Intervallgraph. Ein Durchschnittsgraph von Intervallen. Teilmen-

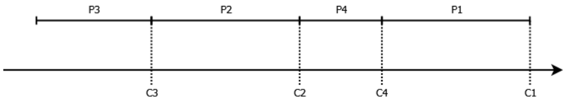


ge von Intervallen, die sich nicht überlappen $\rightarrow$ unabhängige (Knoten-)Menge.
Überlappungsfreie Zuordnung von Maschinen $\rightarrow$ Graphenfärbung (chromatische Zahl ψ)
Punkt x, der in mehreren Intervallen enthalten ist $\rightarrow$ Clique. (vollständiger Teilgraph)
$\leftarrow$ gilt in Intervallgraphen aber nicht in allgemeinen Durchschnittsgraphen.
Die Cliquenzahl (Größe der größten Clique) nennen wir ω
Ausserdem gilt für Intervallgraphen $\omega = \psi$ - sonst gilt im allgemeinen Durschnittsgraphen $\omega \leq \psi$

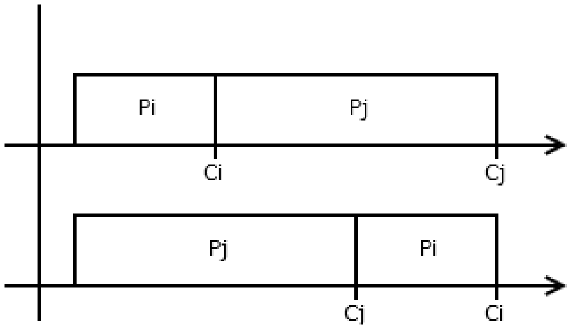
14.5 Zeitplanung(Scheduling)

Man hat n-Aufträge und jeder Auftrag hat eine Bearbeitungszeit p_i und einen Termin d_i(deadline, due date) und die Aufträge müssen jetzt sequentiell abgearbeitet werden.

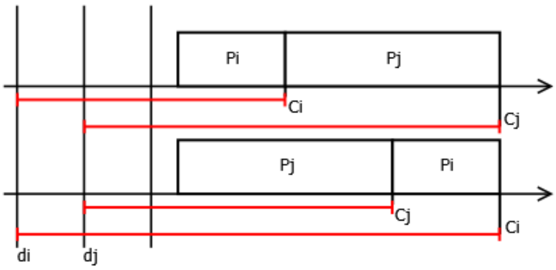
Gesucht ist eine Reihenfolge in der die Aufträge bearbeitet werden.



Aus der Reihenfolge ergibt sich anschließend eine Abschlusszeit C_i, wann der Auftrag fertig ist.
Die Verspätung $L_i = \max\{C_i, d_i, 0\}$
Wir wollen die maximale Verspätung $\max\{L_i | i = 1, ..., n\}$ minimieren.
Was passiert, wenn man zwei benachbarte Aufträge vertauscht.
Wir vergleichen: $\max\{C_j, C_i - d_i, 0\}$



$\max\{C'_j, C'_i - d_i, 0\}$
Es gilt: $C_j = C'_i > Ci, C'_j$
Annahme, beide p's sind verspätet... Behauptung: wenn $d_i \leq d_j$ ist, dann ist die Reihenfolge ij



mindestens so gut wie die Reihenfolge ji.
$\max\{C_{max} - d_j, C_i - d_i\} \leq \max\{C_{max} - d_i, C'_j - d_j\}$
Wissen: $-d_j \leq -d_i$

14.5.1 Beweis

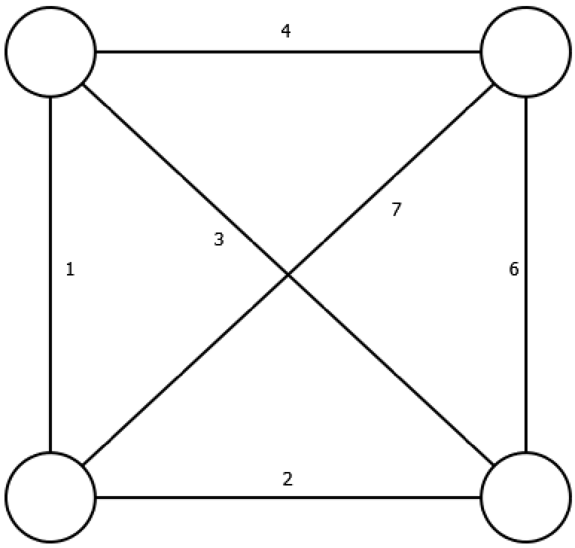
$C_{max} - d_j \leq C_{max} - d_i$
$C_i - d_i \leq C_{max} - d_i$
$\max\{C_{max} - d_j, C_i - d_i\} \leq C_{max} d_i \leq \max\{C_{max} - d_i, C'_j - d_j\}$

14.6 EDD-rule (earliest due date rule

Bearbeite die Aufträge in der Reihenfolge der Termine d_i.
Beweis der Optimalität durch ein Austauschargument.

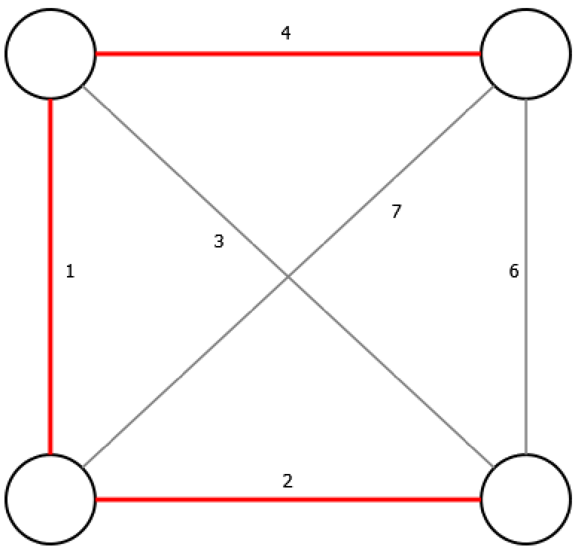
15 Der Klassiker für Greedy Algorithmen: minimal SPT

Gegeben ist ein zusammenhängender ungerichteter Graph mit Kantengewichten ≥ 0.
Gesucht ist ein Spannbaum, der alle Knoten enthält mit kleinstem Gesamtgewicht.
Beispiel: siehe Bild:



15.1 Algorithmus von Kruskal

Betrachte die Kanten in der Reihenfolge nach Gewicht. Wähle die Kante aus, wenn sie mit den bisher gewählten Kanten keinen Kreis bildet.
Dieser Algorithmus hat zu einem grundlegenden Umdenken und weitreichenden Verallgemei-



nerungen geführt. Makroide. Wenn man eine beliebige Matrix nimmt und man betrachtet die unabhängigen Spaltenmengen, dann bilden die ein Makroid. Diese bilden dann eine Basis der Matrix.