Erinnerung: das Knapsack-Problem

- Das 0-1-Knapsack-Problem:
  - Wähle aus n Gegenständen, wobei der i-te Gegenstand den Wert v_i und das Gewicht w_i besitzt
  - Maximiere den Gesamtwert bei vorgegebenem Höchstgewicht W
    - w_i und W sind Ganzzahlen
    - "0-1": jeder Gegenstand muß komplett genommen oder dagelassen werden
- Abwandlung: fraktionales Knapsack-Problem (*fractional KP*)
  - Man kann Anteile von Gegenständen wählen
  - Z.B.: Goldbarren beim 0-1-Problem und Goldstaub beim fraktionalen Problem

Das fraktionale Rucksack-Problem

- Gegeben: n Gegenstände, i-ter Gegenstand hat Gewicht w_i und Wert v_i, Gewichtsschranke W
- Gesucht:

$$q_1, \ldots, q_n \in [0,1] \text{ mit } \sum_{i=1}^{n} q_i w_i \leq W$$

$$\text{und maximalem Wert } \sum_{i=1}^{n} q_i v_i$$

Der Algorithmus

```
# Input: Array G enthält Instanzen der Klasse Item
#        mit G[i].w = Gewicht und G[i].v = Wert
def fract_knapsack( G, w_max ):
    sortiere G absteigend nach (G[i].v / G[i].w)
    w = 0          # = bislang "belegtes" Gesamtgewicht
    for i in range( 0, len(G) ):
        if G[i].w <= w_max - w:
            q[i] = 1
            w += G[i].w
        else:
            q[i] = (w_max - w) / G[i].w
            break          # w_max ist ausgeschoepft
    return q
```

- Aufwand: $O(n) + \underbrace{O(n \log n)}_{\text{Sortieren}}$

Korrektheit

- Hier nur die Beweisidee:
- Der Greedy-Algorithmus erzeugt Lösungen der Form
  (1, …, 1, q_i, 0, …, 0)
- Ann.: es existiert eine bessere Lösung (q'_1, …, q'_m);
  diese hat die Form
  (1, …, 1, q'_j, …, q'_k, 0, …, 0)
- Zeige, daß man aus dieser Lösung eine andere Lösung (q''_1, …, q''_m) konstruieren kann, die dasselbe Gewicht hat, aber größeren Wert, und mehr 1-en oder mehr 0-en

- Achtung: Der Greedy-Algorithmus funktioniert nicht für das 0-1-Rucksack-Problem
- Gegenbeispiel: $W = 50$

Gegenstand	Gewicht	Wert	v_i / w_i
1	10	60	6
2	20	100	5
3	30	120	4

  - Greedy $\rightarrow 1 \times G_1 + 1 \times G_2$
  - Optimal $\rightarrow 1 \times G_2 + 1 \times G_3$

Elemente der Greedy-Strategie

- Die Greedy-Choice-Eigenschaft: global optimale Lösung kann durch lokal optimale (greedy) Auswahl erreichet werden
- Die optimale (Greedy-)Unterstruktur:
  - Eine optimale Lösung eines Problems enthält eine optimale Lösung eines Unterproblems, und diese Teillösung besteht aus 1 Element weniger als die Gesamtlösung
  - M.a.W.: es ist möglich zu zeigen: wenn eine optimale Lösung A Element s_i enthält, dann ist $A' = A \setminus \{s_i\}$ eine optimale Lösung eines kleineren Problems (ohne s_i und evtl. ohne einige weitere Elemente)
    - Bsp. ASP: $A \setminus \{k\}$ ist optimale Lösung für S'

- Optimale (Greedy-)Unterstruktur (Fortsetzung):
  - Man muß nicht alle möglichen Unterprobleme durchprobieren (wie bei Dyn.Progr.) — es genügt, das "nächstbeste" Element in die Lösung aufzunehmen, wenn man nur das richtige lokale(!) Kriterium hat
    - Bsp. ASP: wähle "vorderstes" kompatibles f_i
  - Möglicherweise ist eine Vorbehandlung der Eingabe nötig, um die lokale Auswahlfunktion effizient zu machen
    - Bsp. ASP: sortiere Aktivitäten nach Endzeit f_i

Allgemeine abstrakte Formulierung des Greedy-Algos

- Die Auswahl-Funktion basiert für gewöhnlich auf der Zielfunktion, sie können identisch sein, oft gibt es mehrere plausible Fkt.en

```
# C = Menge aller Kandidaten
# select = Auswahlfunktion
S = []          # = Lösung = Teilmenge von S
while not solution(S) and C != []:
    x = Element aus C, das select(x) maximiert
    C = C \ x
    if feasible(S,x):     # is x compatible with S?
        S += x
if solution(S):
  return S
else:
  return "keine Lösung"
```

- Beispiel ASP:
  - Zielfunktion = Anzahl Aktivitäten (Max gesucht)
  - Auswahl-Funktion = kleinstes $f_i \in S'$

Scheduling in (Betriebs-) Systemen

- Ein einzelner Dienstleister (ein Prozessor, ein Kassierer in einer Bank, usw.) hat n Kunden zu bedienen
- Die Zeit, die für jeden Kunden benötigt wird, ist vorab bekannt: Kunde i benötigt die Zeit t_i, $1 \le i \le n$
- Ziel:
  - Gesamtverweildauer

$$T = \sum_{i=1}^{n} (\text{ Zeit im System für Kunde } i)$$

aller Kunden im System minimieren

Beispiel

- Es gibt 3 Kunden mit $t_1 = 5$, $t_2 = 10$, $t_3 = 3$

Reihenfolge			T		
1	2	3	5 + (5 + 10) + (5 + 10 + 3)	= 38	
1	3	2	5 + (5 + 3) + (5 + 3 + 10)	= 31	
2	1	3	10 + (10 + 5) + (10 + 5 + 3)	= 43	
2	3	1	10 + (10 + 3) + (10 + 3 + 5)	= 41	
3	1	2	3 + (3 + 5) + (3 + 5 + 10)	= 29	← optimal
3	2	1	3 + (3 + 10) + (3 + 10 + 5)	= 34	

Algorithmenentwurf

- Betrachte einen Algorithmus, der den optimalen Schedule Schritt für Schritt erstellt
- Angenommen, nach der Bedienung der Kunden $i_1, \ldots, i_m$ ist Kunde j an der Reihe; die Zunahme von T auf dieser Stufe ist

$$t_{i_1} + \ldots + t_{i_m} + t_j$$

- Um diese Zunahme zu minimieren, muß nur t_j minimiert werden
- Das legt einen einfachen Greedy-Algorithmus nahe: füge bei jedem Schritt denjenigen Kunden, der die geringste Zeit benötigt, dem Schedule hinzu (→ "*shortest job first*")
- Behauptung: Dieser Algorithmus ist immer optimal

Optimalitätsbeweis

- Sei $I = (i_1, \ldots, i_n)$ eine Permutation von $\{1, 2, \ldots, n\}$
- Werden die Kunden in der Reigenfolge I bedient, dann ist die Gesamtverweildauer für alle Kunden zusammen

$$\begin{aligned} T &= t_{i_1} + (t_{i_1} + t_{i_2}) + (t_{i_1} + t_{i_2} + t_{i_1}) + \cdots \\ &= n t_{i_1} + (n-1) t_{i_2} + (n-2) t_{i_1} + \cdots \\ &= \sum_{k=1}^{n} (n - k + 1) t_{i_k} \end{aligned}$$

- Jetzt nehmen wir an, daß in I zwei Zahlen *a*, *b* gefunden werden können, mit $a < b$ und $t_{i_a} > t_{i_b}$
- Durch Vertauschung dieser beiden Kunden ergibt sich eine neue Bedienungsreihenfolge I', die besser ist, weil

$$T(I) = (n-a+1)t_{i_a} + (n-b+1)t_{i_b} + \sum_{\substack{k=1 \\ k \neq a,b}}^{n} (n-k+1)t_{i_k}$$

$$T(I') = (n-a+1)t_{i_b} + (n-b+1)t_{i_a} + \sum_{\substack{k=1 \\ k \neq a,b}}^{n} (n-k+1)t_{i_k}$$

$$\begin{aligned} T(I) - T(I') &= (n-a+1)(t_{i_a} - t_{i_b}) + (n-b+1)(t_{i_b} - t_{i_a}) \\ &= (b-a)(t_{i_a} - t_{i_b}) \\ &> 0 \end{aligned}$$

Reihenfolge	1	...	a	...	b	...	n
bedienter Kunde	i_1	...	i_a	...	i_b	...	i_n
Bedienzeit	t_{i_1}	...	t_{i_a}	...	t_{i_b}	...	t_{i_n}
nach Austausch und von t_{i_a} und t_{i_b}	↓		✕		✕		↓
Bedienzeit	t_{i_1}	...	t_{i_b}	...	t_{i_a}	...	t_{i_n}
bedienter Kunde	i_1	...	i_b	...	i_a	...	i_n

Das SMS-Aufkommen der letzten Jahre

- $1.5 \cdot 10^{12}$ SMS (text messages) im Jahr 2009 allein in den USA
  - Annualized Total Wireless Revenues = $ 152.6 Milliarden [USA, 2009]
- $23 \cdot 10^{9}$ text messages in one week in China around (Chinese) New Year 2010
- $9.6 \cdot 10^{9}$ text messages in UK im Dez. 09

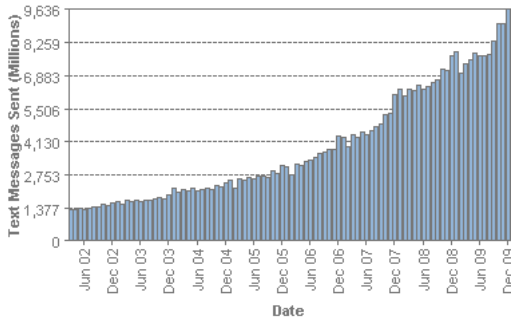
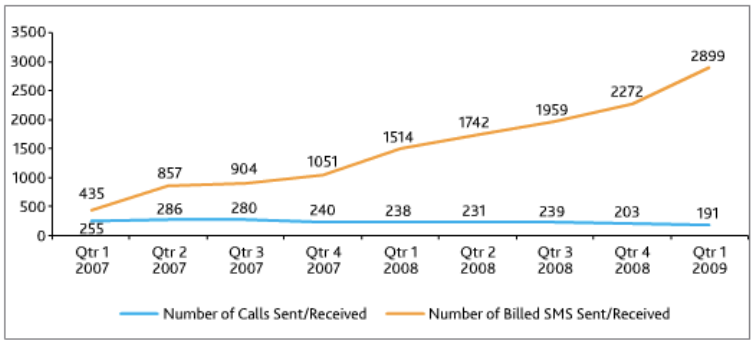


Figure 12: Average Number of Monthly Texts and Phone Calls—U.S. Mobile Teens 13–17



Source: The Nielsen Company

- Fazit: eine gute Kodierung ist sehr wichtig
  - "Gut" = Kodierung, die Bandbreite spart

Kodierung und Kompression

- Kodierung = Konvertierung der Darstellung von Information in eine andere Darstellung
  - Beispiel: Schriftzeichen sind ein Code, ASCII ein anderer Code
- Eine (von vielen) Aufgaben der Kodierung: platzsparende Speicherung von Information (Kompression)
- *Fixed-length code* = konstante Zahl von Bits pro Zeichen (z.B. ASCII)
- Idee der Huffman-Kodierung: Kodiere einzelne Zeichen durch Code-Wörter mit variabler Länge (*variable-length code*)
  - Häufig auftretende Eingabewörter werden mit einem kurzen Code dargestellt
  - Seltene Eingabewörter bekommen ein längeres Code-Wort

Beispiel Morsecode

- Morsecode kodiert Buchstaben in Abhängigkeit der Häufigkeit

A	B	C	D	E	F	G	H	I	J	K	L	M
•–	–•••	–•–•	–••	•	••–•	––•	••••	••	•–––	–•–	•–••	––

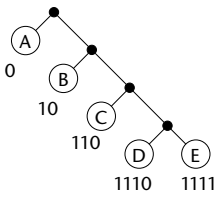
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
–•	–––	•––•	––•–	•–•	•••	–	••–	•••–	•––	–••–	–•––	––••

- Alphabet = drei Zeichen: •, –, □ (Pause)
- HALLO → •••• □ •– □ •–•• □ •–•• □ –––
- Die Pause ist ein wichtiges Zeichen:
  •••••–•–•••–••––– → EEEEETETEEETEETTT
  •••••–•–•••–••––– → ISTRERAM
  •••••–•–•••–••––– → HRBXM
  •••••–•–•••–••––– → IIAAIAIO … usw.
- Frage: gibt es Codes ohne Pause-Zeichen (= Leerzeichen), die trotzdem eindeutig decodiert werden können? (und kurz sind!)

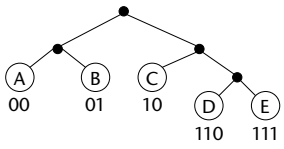
Eindeutige Codes

- Definition der Fano-Bedingung:
  Eine Kodierung besitzt die Fano-Eigenschaft ⇔
  kein Code-Wort ist Präfix eines anderen Code-Wortes
- Solche Codes heißen *prefix codes*
- Idee: betrachte Eingabewörter als Blätter eines Baumes
- Ein Code-Wort entspricht dem Pfad von der Wurzel zum Blatt
  - Verzweigung zum linken Sohn → „0“
  - Verzweigung zum rechten Sohn → „1“
- Erfüllt offensichtlich die Fano-Bedingung
- Länge eines Code-Wortes = Tiefe des entsprechenden Blattes
  - Ziel ist also: häufige Zeichen möglichst weit oben im Baum ansiedeln

- Beispiel: Σ = {A, B, C, D, E}, Text = `AABAACDAAEABACD`



00100011011100011110100110 1110
→ 30 bits



000001000010110000011100010010110
→ 33 bits

- Bemerkung: jeder beliebige Baum mit M Blättern kann benutzt werden, um jeden beliebigen String mit M verschiedenen Zeichen zu kodieren:
  - Die Darstellung als Binär-Baum mit einer Annotation der Zweige mit "0" bzw. "1" garantiert, daß kein Code-Wort Präfix eines anderen ist
  - Somit läßt sich die Zeichenfolge unter Benutzung des Baumes auf eindeutige Weise decodieren
  - Der Algorithmus: bei der Wurzel beginnend bewegt man sich entsprechend den Bits der Zeichenfolge im Baum abwärts; jedesmal, wenn ein Blatt angetroffen wird, gebe man das zu diesem Knoten gehörige Zeichen aus und beginne erneut bei der Wurzel

Herleitung der Huffman-Kodierung

- Lemma: Ein optimaler Baum zur Kodierung ist ein voller Baum, d.h., jeder innere Knoten hat genau 2 Kinder
- Beweis durch Widerspruch (Übungsaufgabe)

→ Sei Σ das Alphabet, dann hat der Baum $|\Sigma|$ viele Blätter (klar) und $|\Sigma|$-1 innere Knoten (s. Kapitel über Bäume)

- Gegeben: String $S = s_1 \ldots s_n$ über Σ.
  Sei $H_S(c)$ = absolute Häufigkeit des Zeichens c im String S
  Sei $d_T(c)$ = Tiefe von c im Baum T = Länge des Codewortes von c
- Anzahl Bits zur Kodierung von S mittels Code T ist

$$B_T(S) = \sum_{c \in \Sigma} H_S(c) d_T(c)$$

- Im folgenden: arbeite mit der mittleren Codewort-Länge, gemessen über alle möglichen (sinnvollen!) Texte der Sprache (z.B. Deutsch)

$$B(T) = \frac{1}{\text{Länge aller Texte}} B_T(\text{alle Texte})$$

- Anders geschrieben:

$$B(T) = \sum_{c \in \Sigma} h_S(c) d_T(c)$$

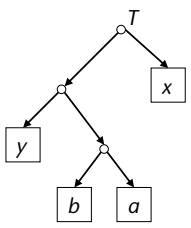
wobei $h_S(c)$ = relative Häufigkeit des Zeichens c in der betrachteten Sprache S

- Ziel: Baum T bestimmen, so daß $B(T)$ = min

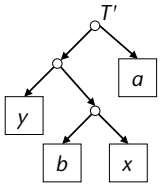
- Lemma:

Seien $x, y \in \Sigma$ die Zeichen mit minimaler Häufigkeit in S.
Dann existiert ein optimaler Kodierungs-Baum für Σ,
in dem x und y Brüder sind
und auf dem untersten Level.

- Beweis durch Widerspruch:
  - Idee: ausgehend von irgend einem optimalen Baum, konstruiere neuen Baum, der auch optimal ist und die Bedingungen erfüllt
  - Ann.: T ist optimaler Baum, aber x, y sind nicht auf max. Level → es ex. 2 Blätter a, b, die Brüder sind, mit $d_T(a) = d_T(b) \geq d_T(x), d_T(y)$



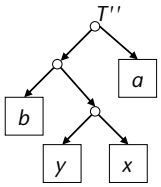
- Vertausche Blätter x und a → ergibt einen Baum T', der besser ist:



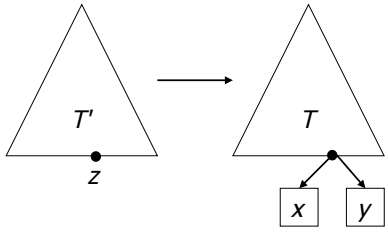
$$
\begin{aligned}
& B(T) - B(T') \\
&= \sum_{c \in \Sigma} h(c) d_T(c) - \sum_{c \in \Sigma} h(c) d_{T'}(c) \\
&= h(x) d_T(x) + h(a) d_T(a) - h(x) d_{T'}(x) - h(a) d_{T'}(a) \\
&= h(x) d_T(x) + h(a) d_T(a) - h(x) d_T(a) - h(a) d_T(x) \\
&= \big(h(a) - h(x)\big)\big(d_T(a) - d_T(x)\big) \\
&\geq 0
\end{aligned}
$$

- Analog: y und b vertauschen ergibt T'' mit

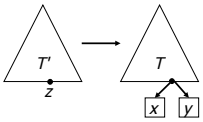
$$B(T'') \leq B(T')$$



- Lemma:
  Seien $x, y \in \Sigma$ Zeichen mit minimaler Häufigkeit.
  Setze $\Sigma' := \Sigma \setminus \{x, y\} \cup \{z\}$. (z ist ein neues "erfundenes" Zeichen.)
  Definiere h auf Σ' wie auf Σ und setze $h(z) := h(x) + h(y)$.
  Sei T' ein optimaler Baum (bzgl. Codelänge) über Σ'.
  Dann erhält man einen optimalen Baum T für Σ aus T', indem man das Blatt z in T' ersetzt durch einen inneren Knoten mit den beiden Kindern x und y.



- Beweis:
  - Zunächst $B(T)$ mittels $B(T')$ ausdrücken:



$$\forall c \in \Sigma \setminus \{x, y\} : d_T(c) = d_{T'}(c)$$

$$d_T(x) = d_T(y) = d_{T'}(z) + 1$$

$$h(x)d_T(x) + h(y)d_T(y) = \big(h(x) + h(y)\big)\big(d_{T'}(z) + 1\big)$$
$$= h(z)d_{T'}(z) + h(x) + h(y)$$

$$B(T) = B(T') + h(x) + h(y)$$

  - Ann.: T ist nicht optimal → es gibt (optimales) T'' mit $B(T'') < B(T)$
    - T'' optimal → x, y sind Brüder auf unterstem Level in T'' (gemäß erstem Lemma)
    - Erzeuge Baum T''' aus T'', in dem x, y und deren gemeinsamer Vater ersetzt werden durch ein neues Blatt z, mit $h(z) = h(x) + h(y)$
    - $B(T''') = B(T'') - h(x) - h(y) < B(T) - h(x) - h(y) = B(T')$
    - Also wäre schon T' nicht optimal gewesen → Widerspruch!

Der Huffman-Code

- Idee: erzeuge einen optimalen Code-Baum mit zwei Schritten
  1. Sortiere die Zeichen nach ihrer relativen Häufigkeit
  2. Erzeuge den Baum „bottom-up“ durch wiederholtes Verschmelzen der beiden seltensten Zeichen
- Beispiel: Σ = {A, B, C, D, E}

1. Ermittlung der Häufigkeiten und Sortierung

Zeichen	*h*
A	8/15
B	2/15
C	2/15
D	2/15
E	1/15

2. Aufbau des Baums

(rel. Häufigkeiten sind mit 15 erweitert)

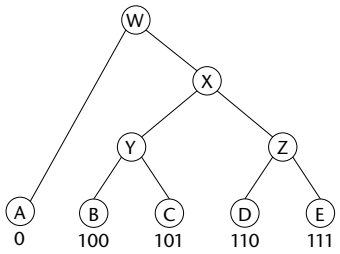
A: 8 B: 2 C: 2 D: 2 E: 1

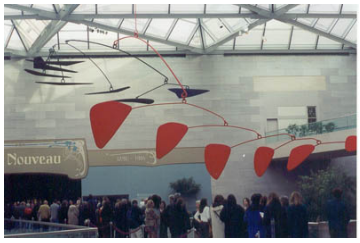
A: 8 Z: 3 B: 2 C: 2

A: 8 Y: 4 Z: 3

A: 8 X: 7

W: 15

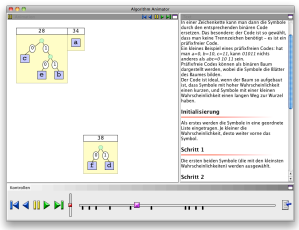




Der Algorithmus zur Erzeugung des Codes

```
# C = Alphabet, jedes c∈C hat c.freq
# q = P-Queue, sortiert nach c.freq
füge alle c∈C in q ein
while q enthält noch mehr als 1 Zeichen:
    x = extract_min( q )
    y = extract_min( q )
    erstelle neuen Knoten z mit Kindern x und y
    z.freq = x.freq + y.freq
    q.insert( z )
return extract_min(q)    # = Wurzel des Baumes
```

(Anmerkung: dies ist ein Greedy-Algorithmus)



http://graphics.ethz.ch/teaching/former/-infotheory0506/Downloads/-Applet_work_fullscreen.html

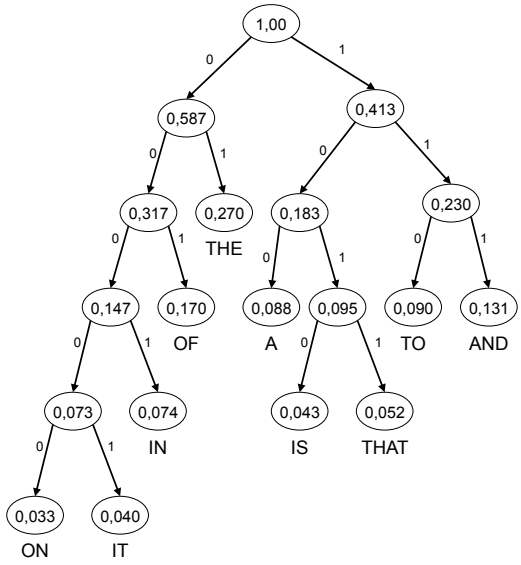
Gesamt-Algorithmus zum Kodieren eines Textes

1. Aufbau des Code-Baums (muß man nur 1x machen)
   - Häufigkeit beim Durchlaufen des Text-Korpus ermitteln
   - Heap für die Knoten, geordnet nach Häufigkeit
   - Code-Baum konstruieren (greedy)
2. Erzeugen der Code-Tabelle (dito)
   - Traversierung des Baumes
3. Kodieren des Textes
   - Look-up der Codes pro Zeichen in der Code-Tabelle
4. Optional:
   - Erzeuge für jeden Text eine eigene Code-Tabelle (Schritt 1 und 2)
   - Übertrage die Code-Tabelle mit dem Text
   - Lohnt sich nur, falls Länge des Textes << Länge der Code-Tabelle

Dekodierung mit Huffman-Codes

- Aufbau des Code-Baums
  - Ergibt binären Baum (linkes Kind = "0", rechtes Kind = "1")
- Dekodieren der Bitfolge (= kodierter Text)
  - Beginne an der Wurzel des Code-Baums
  - Steige gemäß der gelesenen Bits im Baum ab
  - Gib bei Erreichen eines Blattes das zugehörige Zeichen aus und beginne von vorne

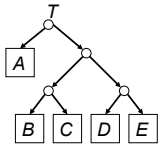
Beispiel: Kodierung ganzer Wörter



Codierungseinheit	W'keit des Auftretens	Code-Wert	Code-Länge
the	0,270	01	2
of	0,170	001	3
and	0,137	111	3
to	0,099	110	3
a	0,088	100	3
in	0,074	0001	4
that	0,052	1011	4
is	0,043	1010	4
it	0,040	00001	5
on	0,033	00000	5

Effiziente Übertragung der Code-Tabelle

- Beobachtungen:
  - Genaue Lage eines Zeichens im Baum ist egal! Wichtig ist nur die Tiefe! (und, natürlich, eine konsistente Beschriftung der Kanten)
  - Umarrangieren liefert Baum, in dem Blätter von links nach rechts tiefer werden
  - Codes der Zeichen ändern sich, nicht aber die Optimalität
- Darstellung des Baumes:
  - Gib Anzahl Zeichen mit bestimmter Länge an, gefolgt von diesen Zeichen, aufsteigend nach Länge
  - Beispiel:
    - 1 Zeichen der Länge 1, 0 der Länge 2, 4 der Länge 3
    - 01, A, 00, , 04, B, C, D, E
  - Damit lässt sich der ursprüngliche Baum wieder genau rekonstruieren

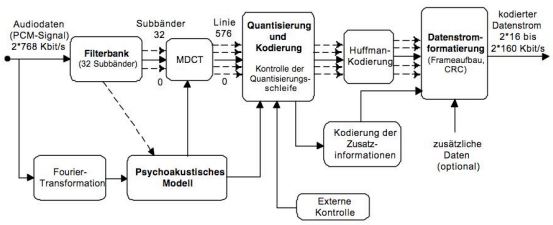


Eigenschaften

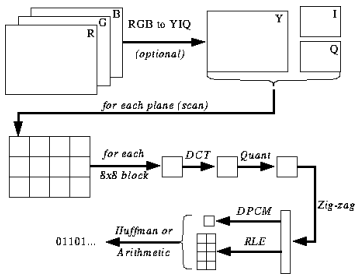
- In gewissem Sinn(!) optimaler, d.h. minimaler, Code
  - Voraussetzung u.a.: Vorkommen der Zeichen ist unabhängig voneinander; die Zeichen werden immer einzeln kodiert
- Keine Kompression für zufällige Daten
  - Alle Zeichen habe annähernd gleiche Häufigkeit
- Erfordert i.A. zweimaliges Durchlaufen des Textes
  - Ermöglicht kein „*Stream Processing*“
- Das Wörterbuch muß i.A. zusätzlich gespeichert werden
  - Für große Dateien vernachlässigbar
  - Nicht aber für kleine

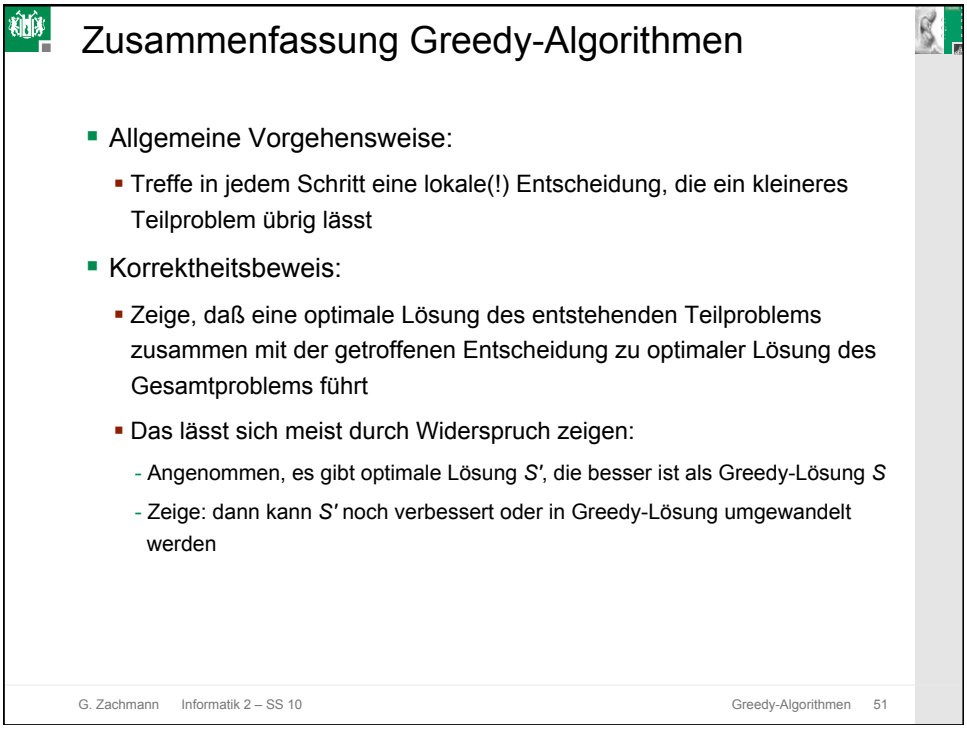
Anwendungsbeispiele der Huffman-Kodierung

- MP3-Kompression:



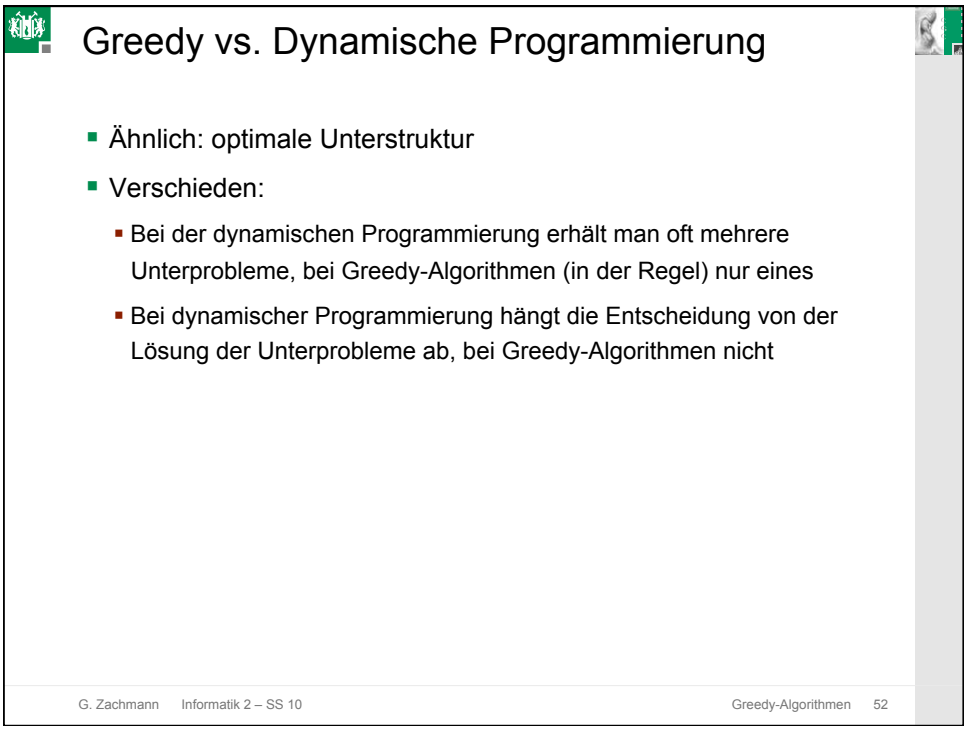
- JPEG-Kompression:





Zusammenfassung Greedy-Algorithmen

- Allgemeine Vorgehensweise:
  - Treffe in jedem Schritt eine lokale(!) Entscheidung, die ein kleineres Teilproblem übrig lässt
- Korrektheitsbeweis:
  - Zeige, daß eine optimale Lösung des entstehenden Teilproblems zusammen mit der getroffenen Entscheidung zu optimaler Lösung des Gesamtproblems führt
  - Das lässt sich meist durch Widerspruch zeigen:
    - Angenommen, es gibt optimale Lösung S', die besser ist als Greedy-Lösung S
    - Zeige: dann kann S' noch verbessert oder in Greedy-Lösung umgewandelt werden



Greedy vs. Dynamische Programmierung

- Ähnlich: optimale Unterstruktur
- Verschieden:
  - Bei der dynamischen Programmierung erhält man oft mehrere Unterprobleme, bei Greedy-Algorithmen (in der Regel) nur eines
  - Bei dynamischer Programmierung hängt die Entscheidung von der Lösung der Unterprobleme ab, bei Greedy-Algorithmen nicht