Algorithmen und Datenstrukturen 2

Lerneinheit 3: Greedy Algorithmen

Prof. Dr. Christoph Karg

Studiengang Informatik
Hochschule Aalen

Sommersemester 2016

10.5.2016

Einleitung

Diese Lerneinheit widmet sich der Verarbeitung von Optimierungsproblemen mittels Greedy Algorithmen.

Im Detail werden folgende Fragen behandelt:

- Was versteht man unter einem Optimierungsproblem?
- Wo steckt die Schwierigkeit bei Lösungssuche von derartigen Problemen?
- Was versteht man unter einer Greedy Strategie?

Optimierungsprobleme

Eigenschaften

- Jede Problemstellung besitzt (in der Regel exponentiell) viele Lösungen
- Jede Lösung besitzt einen Wert bzw. Betrag
- Eine Lösung ist optimal, falls ihr Wert minimal/maximal ist

Aufgabe: Finde eine optimale Lösung

Beachte: Es gibt in der Regel mehrere optimale Lösungen

Unterscheidung:

- Minimierungsproblem $\rightsquigarrow$ Suche einer minimalen Lösung
- Maximierungsproblem $\rightsquigarrow$ Suche einer maximalen Lösung

Greedy Algorithmen

- Ein Greedy Algorithmus ist eine Strategie zur Bearbeitung von Optimierungsproblemen
- Vorgehensweise: Konstruiere eine Lösung durch schrittweises Erweitern einer Teillösung
- Strategie: Wähle die Erweiterung, die momentan am besten ist
- Ein Greedy Algorithmus liefert eine optimale Lösung, falls die lokale Erweiterung auch global optimal ist

Das Raumbelegungsproblem

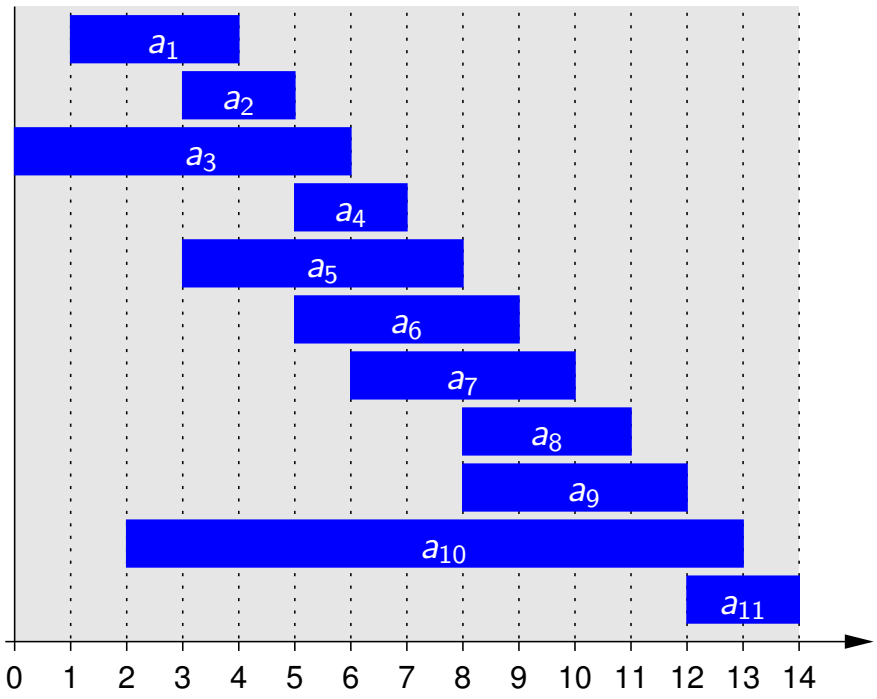
- Gegeben: Menge von Aktivitäten $S = \{a_1, \ldots, a_n\}$
- Jede Aktivität a_i besitzt
  - ▷ eine Startzeit s_i und
  - ▷ eine Endzeit f_i

  wobei $0 \leq s_i < f_i < \infty$.
- Zwei Aktivitäten a_i und a_j sind kompatibel, falls sie nicht gleichzeitig den Raum belegen. Formal:

$$s_i \geq f_j \quad \text{oder} \quad s_j \geq f_i$$

- Aufgabe: Finde eine maximale Teilmenge $A \subseteq S$ von paarweise kompatiblen Aktivitäten

Beispiel



Beispiel (Forts.)

Aktivitätenübersicht in tabellarischer Form:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

Kompatible Aktivitäten:

$$\{a_3, a_9, a_{11}\}, \{a_1, a_4, a_8, a_{11}\}, \{a_2, a_4, a_9, a_{11}\}$$

Inkompatible Aktivitäten:

$$\{a_3, a_4, a_{11}\}, \{a_1, a_4, a_7, a_{11}\}, \{a_3, a_4, a_9, a_{11}\}$$

Idee hinter dem Algorithmus

Voraussetzung: Aktivitäten sind nach aufsteigender Endzeit sortiert:

$$f_1 \leq f_2 \leq \ldots \leq f_n$$

Idee: Konstruktion einer Lösung durch schrittweise Erweiterung einer Teillösung A

Greedy Strategie: Erweitere die Teillösung A um ein Element a mit folgenden Eigenschaften:

- a ist kompatibel zu allen Elementen in A
- a hat die früheste Endzeit aller zu A kompatiblen Aktivitäten

Algorithmus

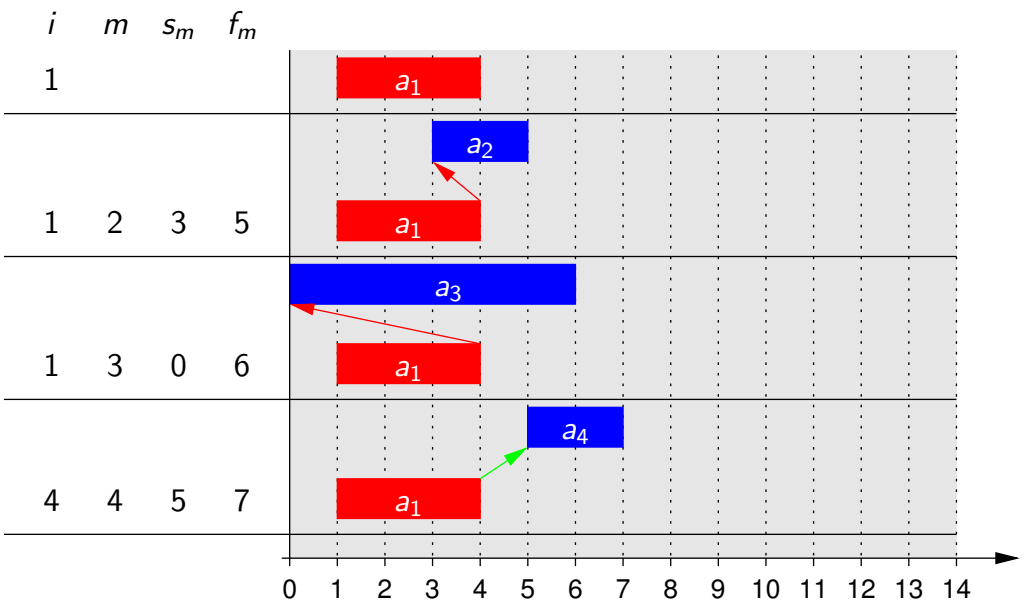
ActivitySelector($\{a_1, \ldots, a_n\}$)
Input: Aktivitäten mit aufsteigend sortierter Endzeit
Output: Maximale Lösung A

```
1 A := {a_1};
2 i := 1;
3 for m := 2 to n do
4   if (s_m ≥ f_i) then
5     A := A ∪ {a_m};
6     i := m;
7 return A;
```

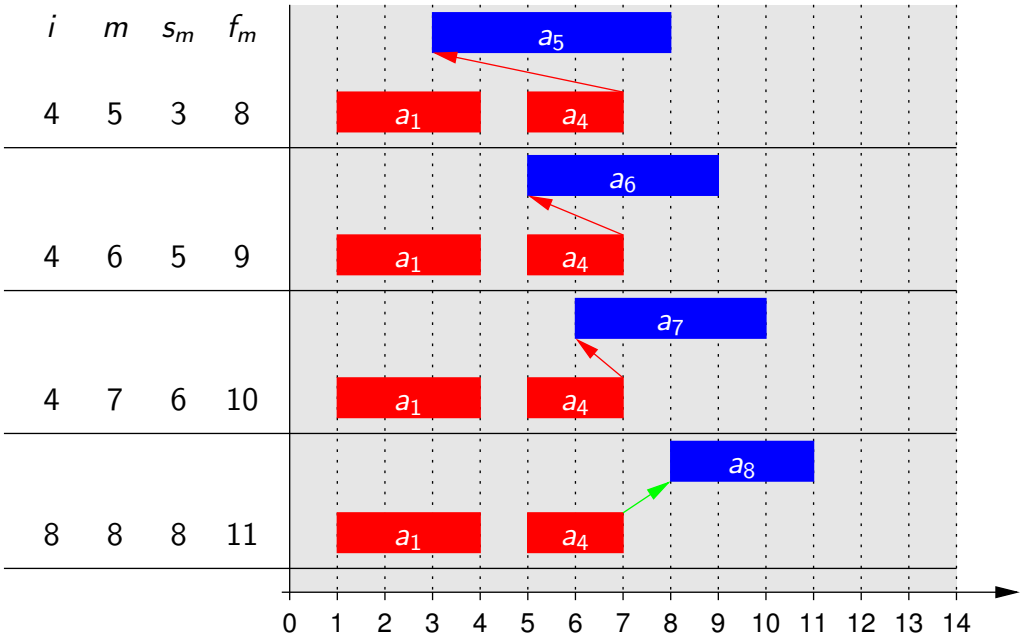
Laufzeit:

- $O(n)$ (bei bereits sortierten Werten)
- $O(n \log_2 n)$ (bei unsortierten Werten)

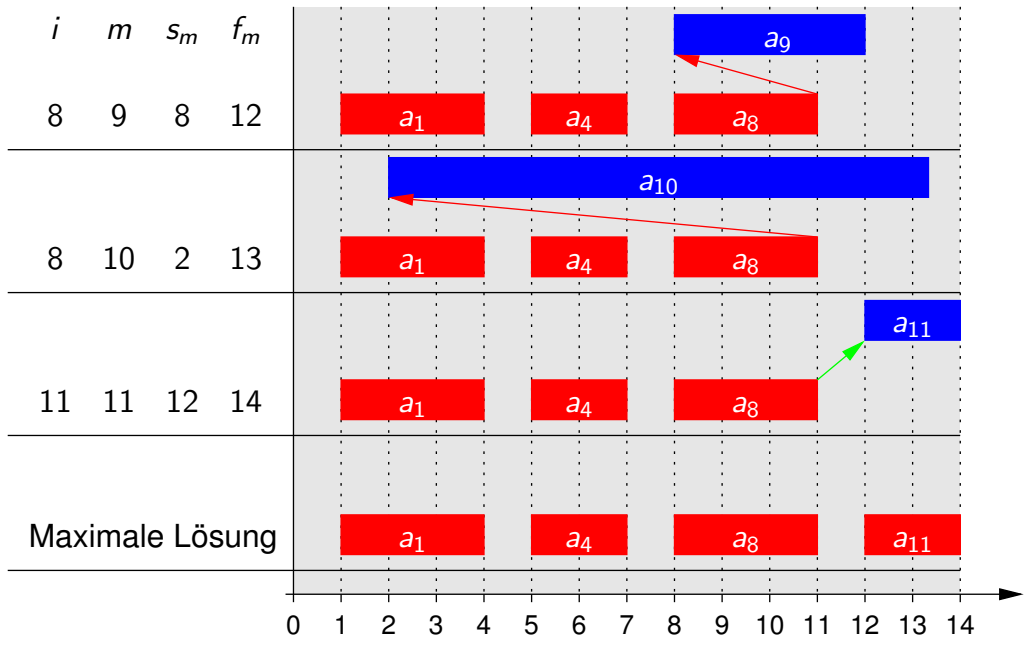
Beispiel



Beispiel (Forts.)



Beispiel (Forts.)



Korrektheit der Greedy Strategie

Schleifeninvariante: Nach jedem Durchlauf der for-Schleife gilt:

- A ist Teilmenge einer optimalen Lösung O
- Die in O enthaltenen Aktivitäten aus $\{a_1, \ldots, a_m\}$ stimmen mit A überein, d.h., $O \cap \{a_1, \ldots, a_m\} = A$
- Die Aktivität a_i wurde zuletzt zu A hinzugefügt

Beweistechnik: Cut & Paste Technik

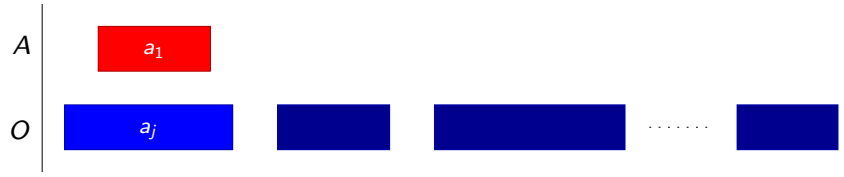
Korrektheit der Greedy Strategie (Forts.)

Initialisierung: Vor dem ersten Durchlauf der Schleife ist $A = \{a_1\}$. Sei O eine optimale Lösung für S.

Fall 1: $a_1 \in O$: ✓

Fall 2: $a_1 \notin O$. Sei $a_j \in O$ die Aktivität mit der frühesten Endzeit.

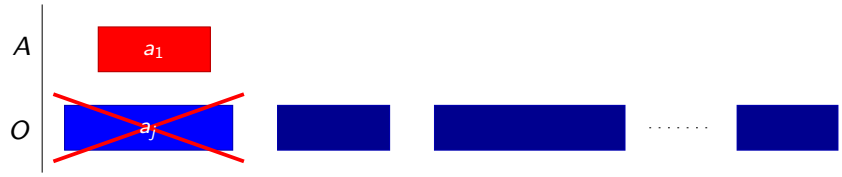
Wegen der Sortierung der Aktivitäten folgt: $f_1 \leq f_j$



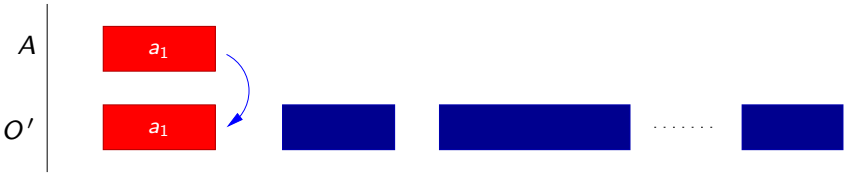
Korrektheit der Greedy Strategie (Forts.)

Beobachtung: Man kann in O a_j durch a_1 ersetzen, ohne Inkompatibilitäten zu erzeugen

Cut:



Paste:



Fazit: $O' = (O \setminus \{a_j\}) \cup \{a_1\}$ ist eine optimale Lösung

Korrektheit der Greedy Strategie (Forts.)

Aufrechterhaltung: Betrachte den Durchlauf der for-Schleife, bei dem a_m verarbeitet wird.

Laut Invariante gilt:

- a_i wurde zuletzt zu A hinzu gefügt
- es gibt es ein Optimum O mit $A = O \cap \{a_1, \ldots, a_{m-1}\}$

Falls $s_m < f_i$, dann wird A nicht verändert ✓

Falls $s_m \geq f_i$. Dann ist $A = A \cup \{a_m\}$

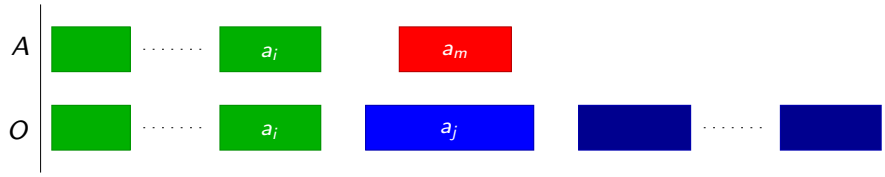
Fall 1: $a_m \in O$ ✓

Korrektheit der Greedy Strategie (Forts.)

Fall 2: $a_m \notin O$. Sei $a_j \in O$ die Aktivität mit $f_i \leq s_j$ und der frühesten Endzeit.

Wegen $O \cap \{a_1, \ldots, a_m\} = A$ ist a_j die Aktivität, die in O direkt nach a_i ausgeführt wird

Wegen der Sortierung der Aktivitäten gilt: $f_m \leq f_j$

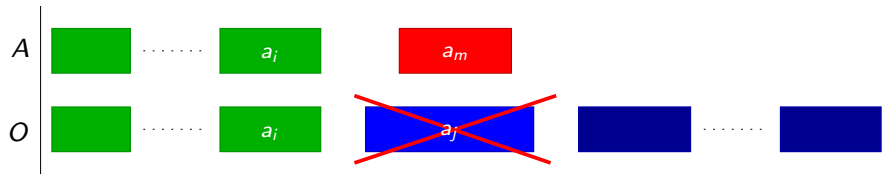


Korrektheit der Greedy Strategie (Forts.)

Beobachtung: a_i und a_j sowie a_i und a_m sind paarweise kompatibel

Folgerung: man kann in O a_j durch a_m ersetzen

Cut:



Paste:

Fazit: $O' = (O \setminus \{a_j\}) \cup \{a_m\}$ ist eine optimale Lösung

Korrektheit der Greedy Strategie (Forts.)

Beendigung: Am Ende der Schleife gilt:

- A ist Teilmenge einer optimalen Lösung O
- $O \cap \{a_1, \ldots, a_n\} = A$

Folglich ist A mit O identisch und somit eine optimale Lösung.

Ergebnis: Der Algorithmus liefert eine optimale Lösung und ist deshalb korrekt.

Huffman Kodierung

- Die Huffman Kodierung ist eine Technik zur Datenkompression
- Die Kompressionsrate liegt bei 20% bis 90% je nach Struktur der Daten
- Im folgenden wird die Huffman Kodierung für die Komprimierung von Zeichenketten eingesetzt
- Die Buchstabenmenge wird mit in binärem Präfixkode kodiert
- Zur Konstruktion des Kodes kommt ein Greedy Algorithmus zum Einsatz

Huffman Kodierung Beispiel

Betrachte einen Text mit 100000 Buchstaben über dem Alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}\}$

	a	b	c	d	e	f
Häufigkeit (1000)	45	13	12	16	9	5
Kode fester Länge	000	001	010	011	100	101
Präfixkode	0	101	100	111	1101	1100

Anzahl Bits bei Kode fester Länge:

$$100000 \cdot 3 = 300000$$

Anzahl Bits bei Einsatz des Präfixkodes:

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224000$$

Präfixkode

- Ein Präfixkode hat die Eigenschaft, dass kein Kodewort ein Präfix eines anderen Kodeworts ist
- Zur Kodierung eines Texts wird jedes Zeichen durch das entsprechende Kodewort ersetzt

  Beispiel: $\mathtt{abce} = 0\ 101\ 100\ 1101$
- Die Dekodierung kann dank der Präfixeigenschaft in einem Durchlauf erfolgen

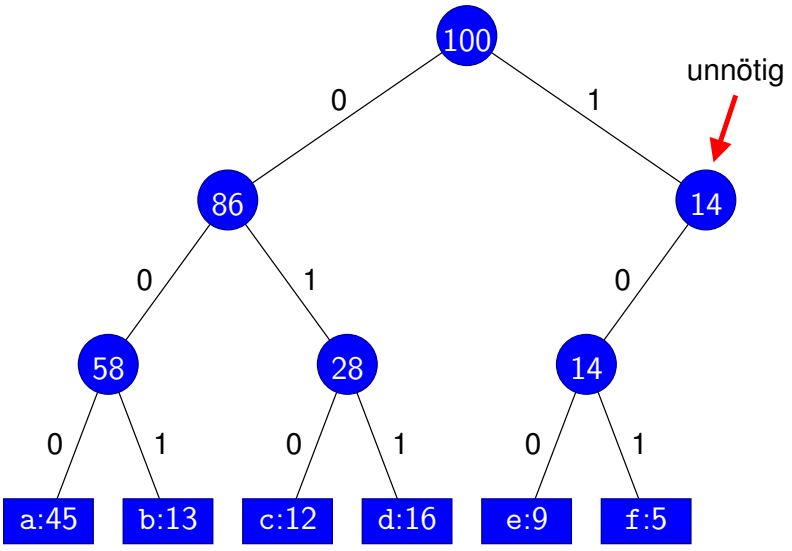
  Beispiel: $01101100 = 0\ 1101\ 100 = \mathtt{aec}$
- Zur Dekodierung wird der Präfixkode in einem Binärbaum gespeichert

Darstellung eines Präfixkodes als Binärbaum

- Jeder Präfixkode ist als Binärbaum darstellbar
- Der Baum ist wie folgt aufgebaut:
  - Die Blätter enthalten die Buchstaben des Alphabets
  - Die Länge des Pfads von der Wurzel zu einem Blatt ist gleich der Länge des Kodeworts des entsprechenden Buchstabens
  - Der Pfad von der Wurzel zu einem Blatt ist mit dem Kodewort des im Blatt gespeicherten Buchstabens beschriftet
- Zur Dekodierung eines Kodeworts wird der Baum von der Wurzel aus durchlaufen
  - Aktuelles Bit $= 0 \rightsquigarrow$ Verzweige nach links
  - Aktuelles Bit $= 1 \rightsquigarrow$ Verzweige nach rechts

  Wird ein Blatt erreicht, dann wird der Buchstabe des Blatts ausgegeben

Präfixkode



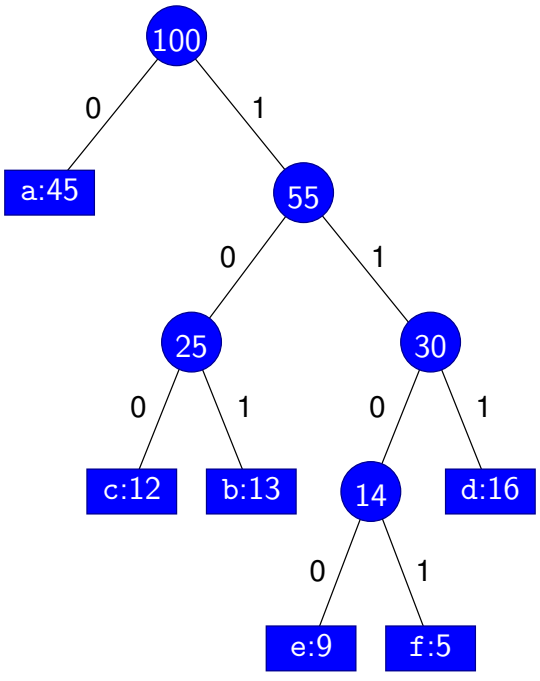
Optimale Präfixkodes

- Ein Binärbaum ist vollständig, wenn jeder interne Knoten zwei Kinder hat
- Der Baum eines optimalen Präfixkodes über dem Alphabet C ist ein vollständiger Binärbaum mit $\|C\|$ Blättern und $\|C\| - 1$ internen Knoten
- $d_T(c)$ bezeichnet die Tiefe des Blatts des Buchstabens c im Binärbaum
- $f(c)$ bezeichnet die Häufigkeit des Buchstabens c im Text
- Die Anzahl Bits zur Kodierung des Textes ist

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

- Ziel: Berechne einen Präfixkode mit minimalem $B(T)$

Optimaler Präfixkode



Zwei wichtige Eigenschaften

Lemma 1. Sei C ein Alphabet und sei $f(c)$ die Häufigkeit des Buchstabens $c \in C$. Seien x und y die Buchstaben mit der kleinsten Häufigkeit.

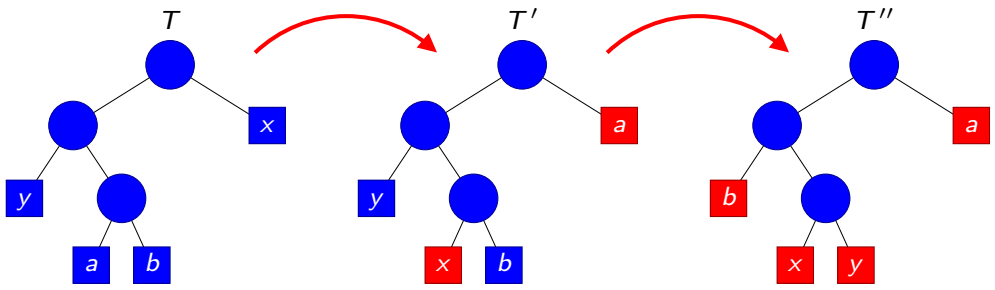
Dann gibt es einen optimalen Präfixkode für C mit folgenden Eigenschaften:

- Die Kodewörter für x und y haben dieselbe Länge.
- Die Kodewörter für x und y unterscheiden sich im letzten Bit.
- Die Kodewörter für x und y haben maximale Länge, vergleichen mit den anderen Kodewörtern.

Folgerung: Im Kodebaum von C sind die Blätter von x und y Geschwisterknoten mit maximaler Tiefe im Baum.

Zwei wichtige Eigenschaften (Forts.)

Beweis. Cut & Paste: Modifiziere einen optimalen Kodebaum T so, dass er die Eigenschaften des Lemmas erfüllt



Zwei wichtige Eigenschaften (Forts.)

Betrachte einen Kodebaum T mit minimalem $B(T)$. Seien a und b zwei Geschwisterblätter mit maximaler Tiefe in T.

Annahme: $f(a) \leq f(b)$ und $f(x) \leq f(y)$

Hieraus folgt: $f(x) \leq f(a)$ und $f(y) \leq f(b)$

Sei T' der Kodebaum, der aus T durch Vertauschen der Knoten a und x entsteht

Da T optimal ist, gilt $B(T) \leq B(T')$

Zwei wichtige Eigenschaften (Forts.)

Abschätzung:

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

Somit: $B(T) \geq B(T')$

Zwei wichtige Eigenschaften (Forts.)

Insgesamt: $B(T) = B(T')$

Sei T'' der Kodebaum, der aus T' durch Vertauschen der Knoten b und y entsteht.

Auf analoge Weise zeigt man: $B(T'') = B(T')$

Also ist T'' ein optimaler Präfixkode mit den in Lemma 1 geforderten Eigenschaften.

Zwei wichtige Eigenschaften (Forts.)

Lemma 2. Sei C ein Alphabet und sei $f(c)$ die Häufigkeit des Buchstabens $c \in C$. Seien x und y Buchstaben mit minimaler Häufigkeit.

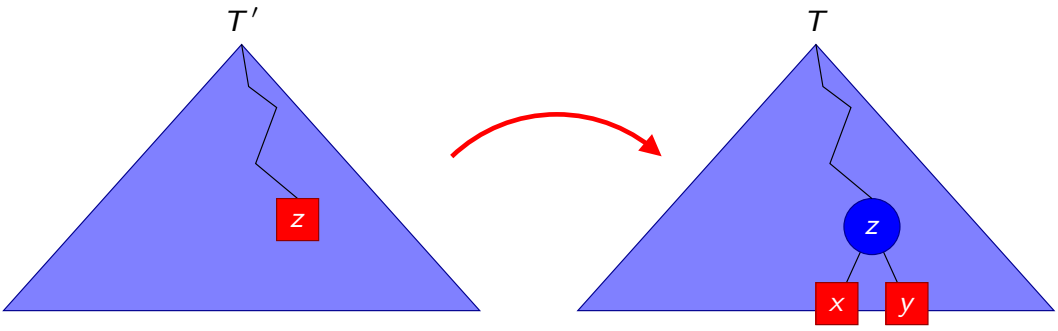
Betrachte das Alphabet $C' = C \setminus \{x, y\} \cup \{z\}$, das aus C durch Streichen der Buchstaben x und y und Hinzunahme des neuen Buchstabens z entsteht. Die Häufigkeit von z ist definiert als $f(z) = f(x) + f(y)$

Sei T' der Baum eines optimalen Präfixkode für C'. Betrachte den Baum T, der aus T' entsteht, indem man das Blatt z durch einen internen Knoten mit Blättern x und y ersetzt.

Es gilt: T ist der Baum eines optimalen Präfixkode für C.

Zwei wichtige Eigenschaften (Forts.)

Interpretation.



optimal für $C' = C \setminus \{x, y\} \cup \{z\}$
mit $f(z) = f(x) + f(y)$

optimal für C

Zwei wichtige Eigenschaften (Forts.)

Beweis. Aufgrund der Konstruktion gilt:

- Für alle $c \in C \setminus \{x, y\}$: $d_T(c) = d_{T'}(c)$. Somit:

$$f(c)d_T(c) = f(c)d_{T'}(c)$$

- Für x, y: $d_T(x) = d_{T(y)} = d_{T'}(z) + 1$. Somit:

$$\begin{aligned} & f(x)d_T(x) + f(y)d_T(y) \\ = \; & (f(x) + f(y))(d_{T'}(z) + 1) \\ = \; & f(z)d_{T'}(z) + (f(x) + f(y)) \end{aligned}$$

Hieraus folgt: $B(T') = B(T) - f(x) - f(y)$

Zwei wichtige Eigenschaften (Forts.)

Annahme: T repräsentiert keinen optimalen Präfixkode

Dann gibt es einen optimalen Präfixkode für dessen Baum T_o $B(T_o) < B(T)$ gilt.

Wegen Lemma 1 sind x und y Geschwisterblätter in T_o. Betrachte den Baum T_o' in dem der Vater von x und y durch das Blatt z ersetzt wird, wobei $f(z) = f(x) + f(y)$.

Es gilt:

$$\begin{aligned} B(T_o') &= B(T_o) - f(x) - f(y) \\ &< B(T) - f(x) - f(y) \\ &= B(T') \end{aligned}$$

Widerspruch zur Annahme dass T' der Baum eines optimalen Präfixkodes ist

Algorithmus RecursiveHuffman(C)

RecursiveHuffman(C, f)
Input: Alphabet C mit zugehörigen Buchstabenhäufigkeiten f
Output: Optimaler Präfixkode B

```
1 if C = {x, y} then
2     B[x] := 0
3     B[y] := 1
4     return B
5 else
6     Finde zwei Buchstaben x und y mit minimaler
      Häufigkeit
7     Erzeuge neuen Buchstaben z ∉ C
```

Algorithmus RecursiveHuffman(C) (Forts.)

```
8   C' := C \ {x, y} ∪ {z}
9   f'(z) = f(x) + f(y)
    for a ∈ C \ {x, y} do
10    f'(a) := f(a)
11  B' := RecursiveHuffman(C', f')
12  B[x] := B'[z]0
13  B[y] := B'[z]1
14  for a ∈ C \ {x, y} do
15    B[a] := B'[a]
16  return B
```


Bemerkungen zu RecursiveHuffman(C)

- Der Algorithmus RecursiveHuffman(C) berechnet einen optimalen Präfixkode für das Alphabet C mit den Buchstabenhäufigkeiten f
- Die Korrektheit folgt direkt aus Lemma 2
- Bei einem Alphabet C mit n Buchstaben ist die Anzahl der rekursiven Aufrufe gleich $n-2$
- Die Laufzeit hängt davon ab von
  - der Datenstruktur für C und f, und
  - der Suche der minimalen Elemente
- Die Implementierung ist aufwendig
- Praxis: Einsatz eines iterativen Algorithmus

Idee hinter dem iterativen Algorithmus

Ansatz: Speichere Teilbäume des Präfixkodes in einer Priority Queue (Schlüssel $\rightsquigarrow$ Gewicht des Wurzelknotens)

Beachte: Das Gewicht eines Knotens ist gleich der Summe seiner beiden Kinder

Greedy Strategie: Wiederhole folgende Schritte, bis die Queue nur noch einen Baum enthält:

1. Entferne zwei Knoten mit minimalem Gewicht aus der Priority Queue
2. Fasse die beiden Knoten zu einem Baum zusammen
3. Füge den neuen Baum in die Priority Queue ein

Algorithmus HUFFMAN(C)

HUFFMAN(C)
Input: Alphabet C mit zugehörigen Buchstabenhäufigkeiten
Output: Baum T eines Präfixkodes mit minimalem $B(T)$

```
1  n := ||C||
2  Initialisiere eine Min-Priority Queue Q mit den
   Buchstaben in C
3  for i := 1 to n − 1 do
4     Allokiere den neuen Knoten z
5     x := ExtractMin(Q)
6     left(z) := x
7     y := ExtractMin(Q)
8     right(z) := y
9     f(z) := f(x) + f(y)
10    Insert(Q, z)
11 return ExtractMin(Q)
```

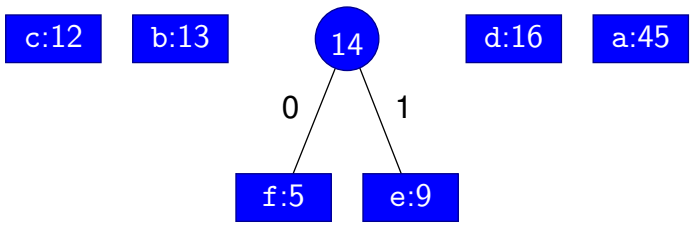
Bemerkungen

- Die Häufigkeit eines Buchstabens beeinflusst die Länge der Kodierung
- Zur Berechnung kommt eine Min-Priority Queue zum Einsatz
- Jeder interne Knoten speichert die Summe der Häufigkeiten der Buchstaben im Unterbaum
- Die Laufzeit ist $O(n \log_2 n)$

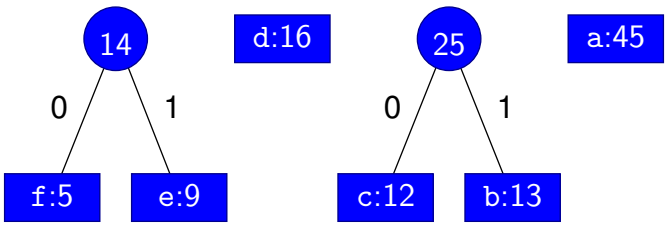
Huffman Kodierung Beispiel

f:5 e:9 c:12 b:13 d:16 a:45

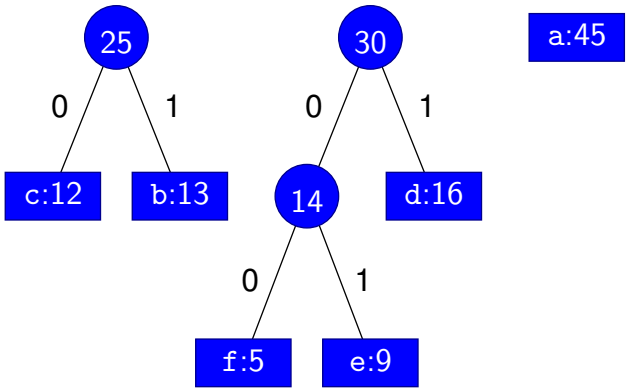
Huffman Kodierung Beispiel (Forts.)



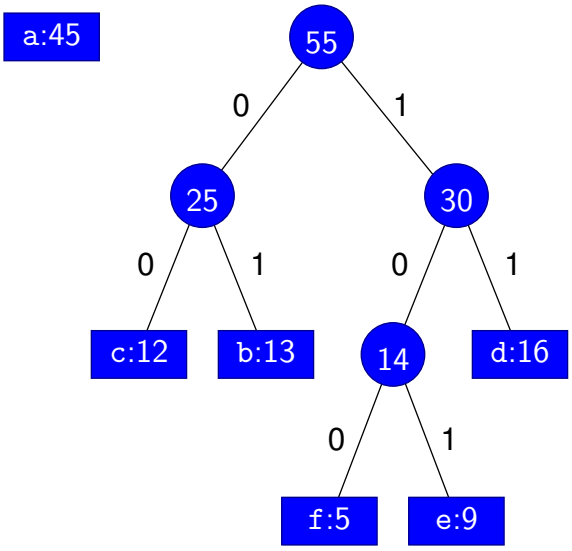
Huffman Kodierung Beispiel (Forts.)



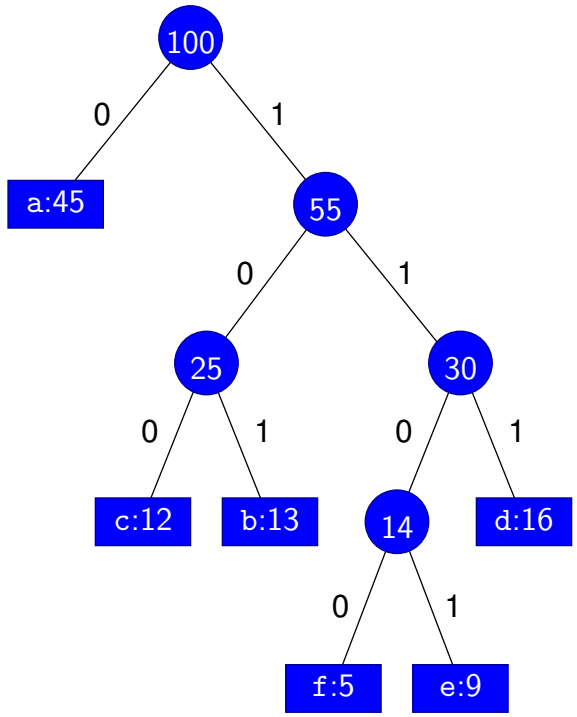
Huffman Kodierung Beispiel (Forts.)



Huffman Kodierung Beispiel (Forts.)



Huffman Kodierung Beispiel (Forts.)



Zusammenfassung

- Greedy Algorithmen werden zur Lösung von Optimierungsproblemen eingesetzt
- Eine Teillösung wird Schritt für Schritt zu einer Gesamtlösung erweitert
- Strategie: Wähle in jedem Schritt die (lokal) beste Erweiterung
- Greedy Algorithmen liefern optimale Lösungen für eine Vielzahl von Problemen, z.B.:
  - Minimale Spannbäume
  - Kürzeste Pfade in Graphen
- Greedy Algorithmen werden als Heuristiken für schwer zu lösende Probleme eingesetzt