Informatik II - SS 2018

(Algorithmen & Datenstrukturen)

Vorlesung 21 (11.7.2018)

String Matching (Textsuche) II
Greedy Algorithmen I

Fabian Kuhn

Algorithmen und Komplexität

UNI FREIBURG

Textsuche / String Matching

Gegeben:

Annahme: $m \ll n$

- Zwei Zeichenketten (Strings)
- Text T (typischerweise lang) Länge von T: n
- Muster P (engl. pattern, typischerweise kurz) Länge von P: m

Ziel:

- Finde alle Vorkommen von P in T

Annahmen:

- Länge Text T : $\boldsymbol{n}$, Länge Muster P : $\boldsymbol{m}$

naiver Alg: $O(n \cdot m)$

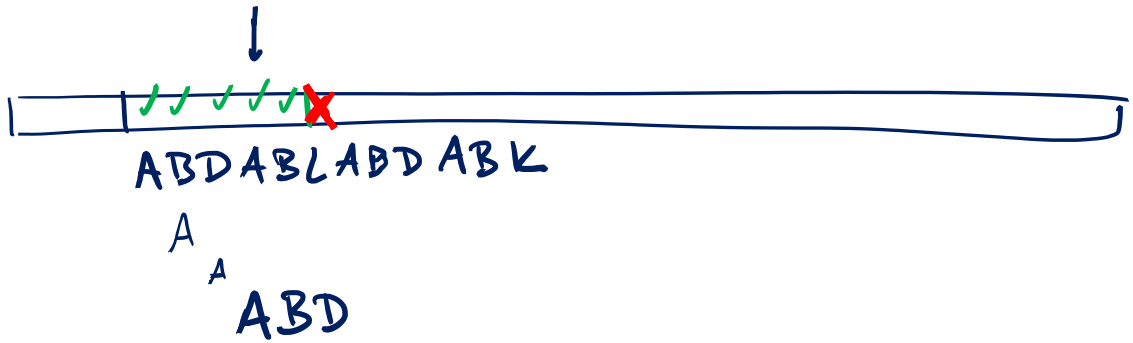
besserer Alg. $O(n + k \cdot m)$ k: Anz. Vorkommen von P
(auf Hashing basiert)

Knuth-Morris-Pratt Algorithmus

Idee:

- Falls wir beim Testen des Musters P an Stelle t feststellen, dass $P[t]$ nicht mit dem Text an der entsprechenden Stelle übereinstimmt, dann wissen wir, dass die Stellen $P[0 \dots t-1]$ übereingestimmt haben.
- Das können wir bei der weiteren Suche ausnutzen

Beispiel: $P = \text{ABDABLABDABK}$



Knuth-Morris-Pratt Algorithmus

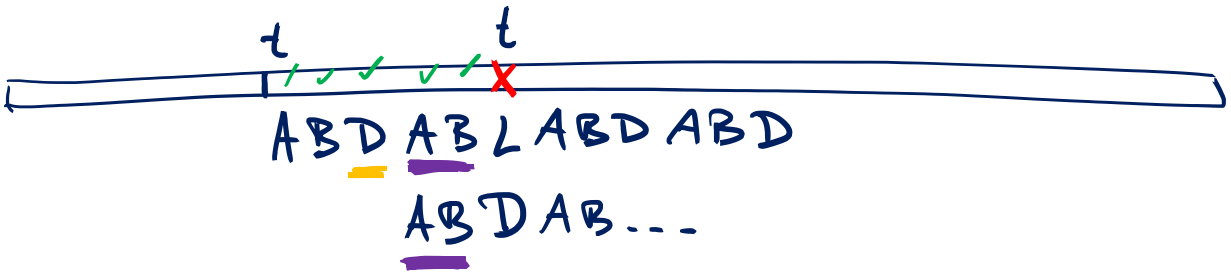
Vorberechnung: Array S der Länge $m + 1$

- $S[i]$: Stelle in P, an welcher man die neue Suche beginnt, falls beim Testen der Stelle i im Pattern ein Mismatch auftritt
- $S[0] = -1, \quad S[1] = 0$
- $S[m]$: Stelle in P, an welcher man weitersucht, nachdem P erfolgreich gefunden wurde

Beispiel:

$P =$ [A, B, D, A, B, L, A, B, D, A, B, D]

$S =$ [-1, 0, 0, 0, 1, 2, 0, 1, 2, 3, 4, 5, 3]



Initialisierung

Vorberechnung von Array S:

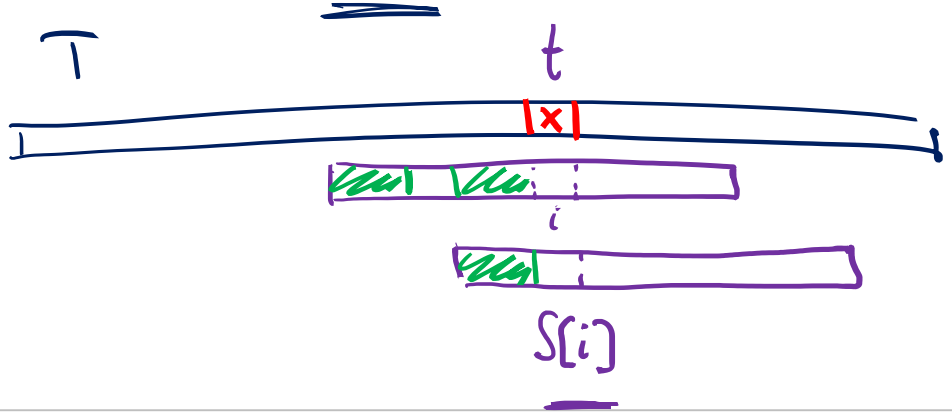
- $P = [\ \underbrace{\texttt{A, B, D}}_{k},\ \texttt{A, B, L},\ \underbrace{\texttt{A, B, D}}_{k},\ \overset{i}{\texttt{A}},\ \texttt{B},\ \texttt{D}\ ]$

  $S = [\texttt{-1, 0, 0, 0, 1, 2, 0, 1, 2, 3, 4, 5, 3}]$

- An Position in $S[i]$ (für $i \in \{2, \dots, m\}$) steht

$$\boldsymbol{S[i] := \overset{\max}{\cancel{\min}}_{k<i}\{\underbrace{P[i-k \dots i-1]}_{\text{suffix}} = \underbrace{P[0 \dots k-1]}_{\text{prefix}}\}}$$

- $S[i]$: Länge des längsten echten Teilstückes von $P[0 \dots i-1]$, welches an Stelle $i-1$ endet, und welches auch Anfangsstück von P ist



Knuth-Morris-Pratt Algorithmus

```
t := 0;  p := 0          // t: Position in Text,   p: Position im Pattern
while t < n do
    if T[t] = P[p] then          // characters match
        if p = m − 1 then        // pattern found
            pattern found at position t − m + 1
            p := S[m];  t := t + 1
        else
            p := p + 1;  t := t + 1
    else                         // characters don't match
        if p = 0 then            // mismatch at first character
            t := t + 1
        else
            p := S[p]
```

Knuth-Morris-Pratt Alg.: Beispiel

$S[i] < i$

Pattern: ABCABC $\quad S = [-1,0,0,0,1,2,3]$

ABC ABC
-1 0 0 0 1 2 3

t

Text: A D A B C D A B C A G A B V A B C A B C A B C

A	B	C	A	B	C																		
	A	B	C	A	B	C																	
		A	B	C	A	B	C																
					A	B	C	A	B	C													
						A	B	C	A	B	C												
							A	B	C	A	B	C											
										A	B	C	A	B	C								
											A	B	C	A	B	C							
												A	B	C	A	B	C						

Knuth-Morris-Pratt Alg.: Laufzeit

Laufzeit ohne Initialisierung des Arrays S:

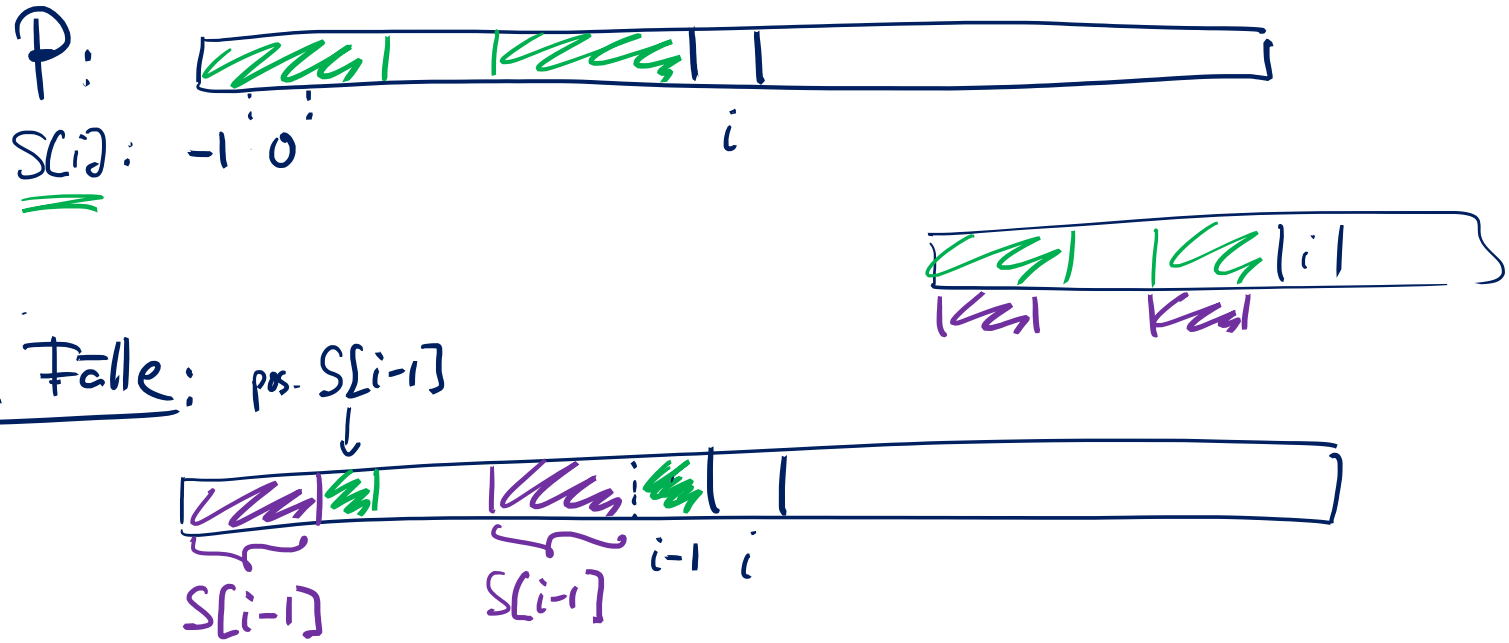
Laufzeit: $\Theta(n)$

```
t := 0;  p := 0
while t < n do
    if T[t] = P[p] then
        if p = m − 1 then
            pattern found
            p := S[m];  t := t + 1
        else
            p := p + 1;  t := t + 1
    else
        if p = 0 then
            t := t + 1
        else
            p := S[p]
```

kommen zus. $\leq n$ mal vor

kommt $\leq n$ mal vor, da $S[p] < p$

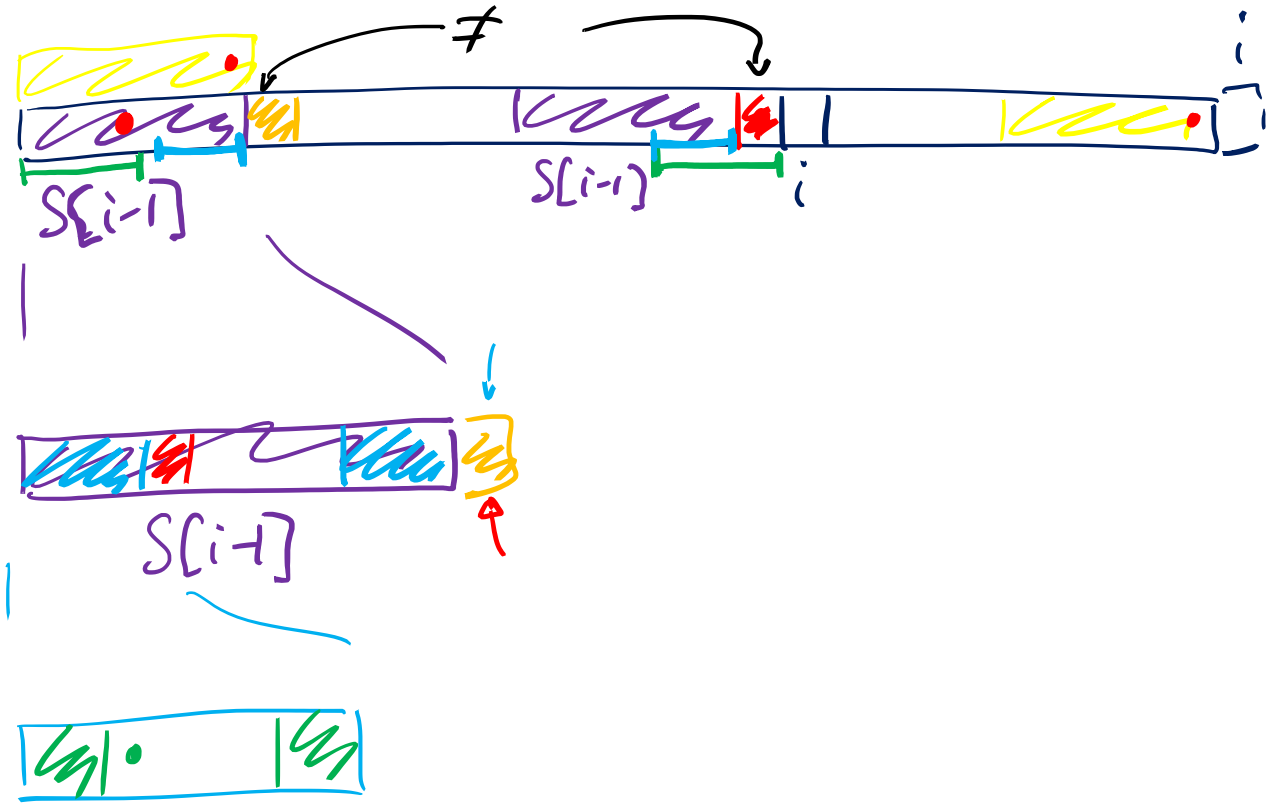
Berechnung von $S[i]$



falls $P[i-1] = P[S[i-1]]$, dann ist $S[i] = S[i-1]+1$

Berechnung von $S[i]$

Annahme: $P[S[i-1]] \neq P[i-1]$



Berechnung von $S[i]$: Beispiel

$h := S[i-1]$

`while` $h \geq 0$ `do`

  `if` $P[i-1] = P[h]$ `then`

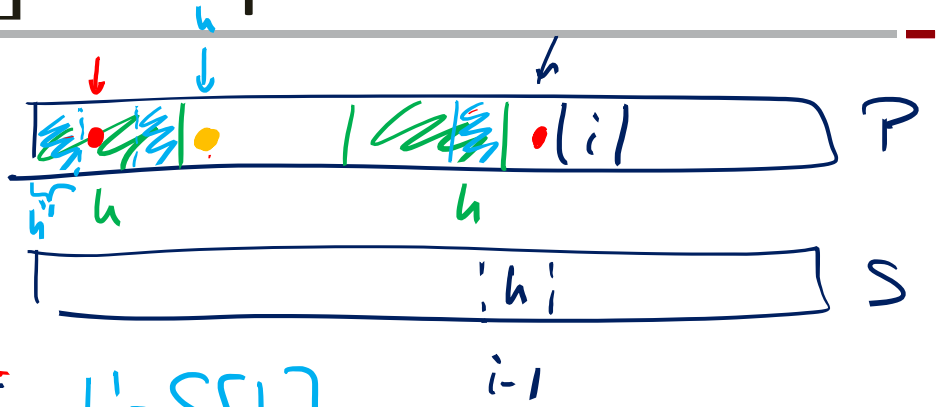
    $S[i] := h + 1;$ $h := -2$

  `else`

    $h := S[h]$

`if` $h = -1$ `then` $S[i] = 0$

Fall 1



$h' = S[h]$

Beispiel: $P = [\mathtt{A},\mathtt{B},\mathtt{D},\mathtt{A},\mathtt{B},\mathtt{L},\mathtt{A},\mathtt{B},\mathtt{D},\mathtt{A},\mathtt{B},\mathtt{D},\mathtt{X}]$

Berechnung von $S[i]$: Laufzeit

$h \coloneqq S[i-1]$
while $h \geq 0$ do
    if $P[i-1] = P[h]$ then
        $S[i] \coloneqq h+1$; $h \coloneqq -2$
    else
        $h \coloneqq S[h]$ ← h wird kleiner
if $h = -1$ then $S[i] = 0$

$h_1 > h_2 > \ldots > h_k$

Beobachtung:

$S[i] \leq S[i-1]+1$

$h_1 = S[i-1]$ $h_k \geq S[i]-1$

oder $h_k = 0 = S[i]$

$S[h] < h$

$S[i-1] = 7$ $S[i] = 3$

7, 6, 5, 4, 3, 2

Falls $S[i] = S[i-1]+1$: 1 Schleifendurchlauf

Falls $S[i] < S[i-1]$:

- Wert von h nimmt in jedem Schleifendurchlauf ab
- Am Schluss ist $S[i] = h+1$
- Anzahl Schleifendurchläufe $\leq \underbrace{\Delta h + 1}_{k} = S[i-1] - S[i] + 2$

Berechnung von $S[i]$: Laufzeit

Falls $\boldsymbol{S[i] = S[i-1] + 1}$:

- Anzahl Schleifendurchläufe $= 1 = S[i-1] - S[i] + 2$

Falls $\boldsymbol{S[i] < S[i-1]}$:

- Anzahl Schleifendurchläufe $\leq \Delta h + 1 = S[i-1] - S[i] + 2$

Gesamtlaufzeit:

$$\leq \sum_{i=2}^{m} \left(S[i-1] - S[i] + 2 \right)$$

$$= 2(m-1) + \left(S[1] - \cancel{S[2]} + \cancel{S[2]} - \cancel{S[3]} + \cancel{S[3]} - \cancel{S[4]} + \cancel{\dots} + \cancel{S[m-2]} - \cancel{S[m-1]} + \cancel{S[m-1]} - S[m] \right)$$

$$= 2(m-1) + \underbrace{S[1]}_{\hookrightarrow = 0} - \underbrace{S[m]}_{\hookrightarrow \geq 0} \leq 2(m-1) = O(m)$$

Knuth-Morris-Pratt Alg.: Zusammenfassung

Knuth-Morris-Pratt Algorithmus:

- Berechnet zuerst in Zeit $O(m)$ das Array S der Länge m
  - hängt nur vom Pattern P ab
  - beschreibt an jeder Position im Pattern, wo (im Pattern) man bei einem Mismatch weitersuchen muss
- Mit Hilfe von S werden dann alle Vorkommen von P in T in Zeit $O(n)$ gefunden
  - In jedem Schritt kann man entweder die aktuelle Suchposition in T oder die Position des Suchfensters in T um mindestens 1 nach rechts verschieben

Gesamtlaufzeit: $\boldsymbol{O(m+n) = O(n)}$

Greedy Algorithmen

- Greedy Algorithmen sind eine Algorithmenmethode, um Optimierungsprobleme zu lösen.
  - wie auch z.B. dynamische Programmierung

- Greedy Algorithmen funktionieren oft nicht gut, aber wenn Sie funktionieren, dann sind sie besonders einfach und oft effizient

- Wir haben schon Beispiele gesehen und werden noch ein weiteres wichtiges Beispiel anschauen

- Vielen Dank an Christian Scheideler (U. Paderborn) für die Folien

Greedy Algorithmen

UNI FREIBURG

“Gierige” Strategie für Optimierungsprobleme:

- Aufbau einer Lösung in „kleinen“ Schritten
- In jedem dieser Schritte wird entsprechend eines definierten Optimierungskriteriums eine irreversible Entscheidung getroffen

Frage 1

Wann kann eine solche Strategie zu einer optimalen Lösung führen?

Frage 2

Wie beweist man, dass ein Greedy Algorithmus eine optimale Lösung liefert?

- Wir werden das an einem Beispiel sehen:
  zusätzliche Beispiele und verschiedene Beweisstrategien werden in der Algorithmentheorie besprochen

Greedy Algorithmen

1. Greedy Algorithmen bestimmen Lösung durch sukzessives Er-
   weitern von bereits gefundenen Teillösungen.

2. Erweitern geschieht durch lokal optimale Wahlen.

3. Analyse muss dann zeigen, dass lokal optimale Wahlen zu global optimalen Lösungen führt.

Greedy Algorithmen – Prim's MST Alg.

1. Greedy Algorithmen bestimmen Lösung durch sukzessives Er-weitern von bereits gefundenen Teillösungen.

2. Erweitern geschieht durch lokal optimale Wahlen.

3. Analyse muss dann zeigen, dass lokal optimale Wahlen zu global optimalen Lösungen führt.

1. Algorithmus von Prim bestimmt minimalen Spannbaum durch sukzessives Hinzufügen von Kanten.

2. Prims Algorithmus wählt möglichst leichte Kante, die isolierten Knoten mit Teilbaum verbindet.

3. Analyse über Schnitte und die Definition von leichten Kanten über einen Schnitt

Greedy Algorithmen – Kruskal's MST Alg.

1. Greedy Algorithmen bestimmen Lösung durch sukzessives Er-weitern von bereits gefundenen Teillösungen.

2. Erweitern geschieht durch lokal optimale Wahlen.

3. Analyse muss dann zeigen, dass lokal optimale Wahlen zu global optimalen Lösungen führt.

1. Algorithmus von Kruskal bestimmt minimalen Spannbaum durch sukzessives Hinzufügen von Kanten.

2. Kruskals Algorithmus wählt möglichst leichte Kante, die Zusammenhangskomp. verbindet.

3. Analyse über Schnitte und die Definition von leichten Kanten über einen Schnitt

Greedy Algorithmen – Datenkompression

UNI FREIBURG

Datenkompression

- Reduziert Größen von Files
- Viele Verfahren für unterschiedliche Anwendungen: MP3, MPEG, JPEG, …
- Wie funktioniert Datenkompression?

Zwei Typen von Kompression:

- Verlustbehaftete Kompression (Bilder, Musik, Filme,…)
- Verlustfreie Kompression (Programme, Texte, …)

Greedy Algorithmen – Datenkompression

UNI FREIBURG

Kodierung:

- Computer arbeiten auf Bits (Symbole 0 und 1), nutzen also das Alphabet {0,1}
- Menschen nutzen umfangreichere Alphabete (z.B. Alphabete von Sprachen)
- Darstellung auf Rechner erfordert Umwandlung in Bitfolgen

Greedy Algorithmen – Datenkompression

Beispiel:

- Alphabet Σ={a,b,c,d,…,x,y,z, ,.,:,!,?,&} (32 Zeichen)
- 5 Bits pro Symbol: $2^5 = 32$ Möglichkeiten

a	b	…	z		.	:	!	?	&
00000	00001		11001	11010	11011	11100	11101	11110	11111

Fragen?

- Sind 4 Bits pro Symbol nicht genug ?
- Müssen wir im Durchschnitt 5 Bits für jedes Vorkommen eines Symbols in langen Texten verwenden?

Greedy Algorithmen – Datenkompression

UNI FREIBURG

Beobachtung:

- Nicht jeder Buchstabe kommt gleich häufig vor
- z. B. kommen x, y und z in der deutschen Sprache viel seltener vor als e, n oder r

Idee:

- Benutze kurze Bitstrings für Symbole die häufig vorkommen

Effekt:

- Gesamtlänge der Kodierung einer Symbolfolge (eines Textes) wird reduziert

Greedy Algorithmen – Datenkompression

Grundlegendes Problem:

- Eingabe: Text über dem Alphabet Σ
- Gesucht: Eine binäre Kodierung von Σ, so dass die Länge des Textes in dieser Kodierung minimiert wird

Beispiel:

- $\Sigma=\{0,1,2,\ldots,9\}$
- Text = 00125590043 56789 (17 Zeichen)

Länge der Kodierung:
$4 \cdot 17 = 68$ Bits

0	1	2	3	4	5	6	7	8	9
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Greedy Algorithmen – Datenkompression

UNI FREIBURG

Grundlegendes Problem:

- Eingabe: Text über dem Alphabet Σ
- Gesucht: Eine binäre Kodierung von Σ, so dass die Länge des Textes in dieser Kodierung minimiert wird

Beispiel:

- $\Sigma=\{0,1,2,\ldots,9\}$
- Text = 00125590004356789 (17 Zeichen)

$5 \cdot 1 + 3 \cdot 2 + 9 \cdot 5 = 56$ Bits

0	1	2	3	4	5	6	7	8	9
0	11000	11001	11010	11011	10	11100	11101	11110	11111

Greedy Algorithmen – Datenkompression

MorseCode:

- Elektrische Pulse über Kabel
- Punkte (kurze Pulse)
- Striche(Lange Pulse)

Beispiele aus dem MorseCode:

- e ist 0 (ein einzelner Punkt) •
- t ist 1 (ein einzelner Strich) —
- a ist 01 (Punkt – Strich) •—

etet : •:—:•:—

aa : •—:•—

Problem:

- Ist 0101 eta, aa, etet, oder aet ?

Greedy Algorithmen – Datenkompression

UNI FREIBURG

Problem Mehrdeutigkeit:

- Ist die Kodierung eines Buchstabens ein Präfix der Kodierung eines anderen Buchstabens, dann kann es passieren, dass die Kodierung nicht eindeutig ist.

Beispiel:

- e = 0, a = 01
- 0 ist Präfix von 01

Präfix-freie Kodierung

Präfix-Kodierung:

Eine Präfix-Kodierung für ein Alphabet Σ ist eine Funktion γ, die jeden Buchstaben $x \in \Sigma$ auf eine endliche Sequenz von 0 und 1 abbildet, so dass für $x,y \in \Sigma$, $x \neq y$, die Sequenz $\gamma(x)$ *kein* Präfix der Sequenz $\gamma(y)$ ist.

Beispiel (Präfix-Kodierung):

$x \in \Sigma$	0	1	2	3	4	5	6	7	8	9
$\gamma(x)$	00	0100	0110	0111	1001	1010	1011	1101	1110	1111

Präfix-Code: Dekodierung

- Wie dekodiert man eine gegebenen mit einem Präfix-Code kodierte Zeichenkette?

Beispiel

$x \in \Sigma$	0	1	2	3	4	5	6	7	8	9
$\gamma(x)$	00	1000	1001	1010	010	011	1011	1101	1110	1111

Kodierte Zeichenkette

010001010011001111011000101101100010000100010010110 0

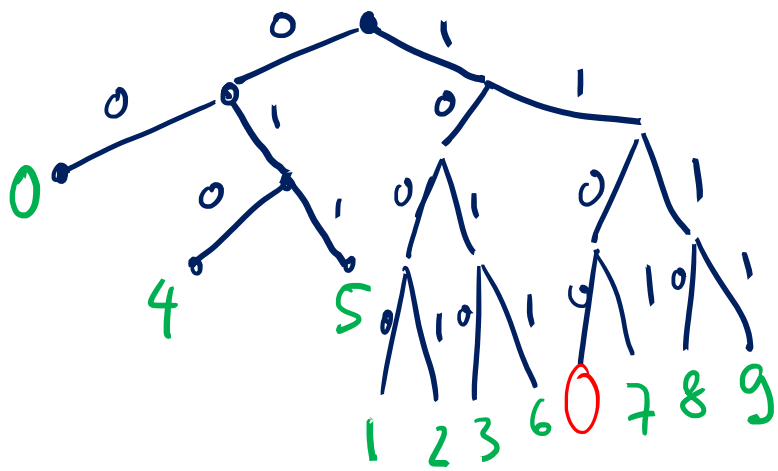
4 0 3 5 0 9

Präfix-Code als Binärbaum

- Wie dekodiert man eine gegebenen mit einem Präfix-Code kodierte Zeichenkette?

Beispiel

$x \in \Sigma$	0	1	2	3	4	5	6	7	8	9
$\gamma(x)$	00	1000	1001	1010	010	011	1011	1101	1110	1111



UNI FREIBURG

Definition (Frequenz)

relative Häufigkeit

- Die Frequenz f[x] eines Buchstaben $x \in \Sigma$ bezeichnet den Bruchteil der Buchstaben im Text, die x sind.

Beispiel:

- $\Sigma = \{0,1,2\}$
- Text =„0010022001“ (10 Zeichen)
- f[0] = 3/5
- f[1] = 1/5
- f[2] = 1/5

Definition (Kodierungslänge)

Die Kodierungslänge eines Textes mit n Zeichen bzgl. einer Kodierung γ ist definert als

Anz. Vorkommen von x

$$\text{Kodierungslänge} = \sum_{x \in \Sigma} n \cdot f[x] \cdot |\gamma(x)|$$

Codewort-Länge von x

Beispiel:

- Σ = {a,b,c,d}
- γ(a) = 0; γ(b) =101; γ(c)= 110; γ(d)=111
- Text = „aacdaabb“
- Kodierungslänge = 16

UNI FREIBURG

Definition (durchschn. Kodierungslänge)

Die durchschnittliche Kodierungslänge eines Buchstabens in einem Text mit n Zeichen und bzgl. einer Kodierung γ ist definiert als

$$\mathbf{ABL}(\gamma) = \sum_{x \in \Sigma} f[x] \cdot |\gamma(x)|$$

Beispiel:

- $\Sigma = \{a,b,c,d\}$
- $\gamma(a) = 0$; $\gamma(b) = 101$; $\gamma(c) = 110$; $\gamma(d) = 111$
- Text = „aacdaabb“
- Durchschnittliche Kodierungslänge = 16/8 = 2

UNI FREIBURG

Problem einer optimalen Präfix-Kodierung:

- **Eingabe:**

  Alphabet Σ und für jedes $x \in \Sigma$ seine Frequenz $f[x]$

- **Ausgabe:** Präfix-Code

  Eine Kodierung γ, die ABL(γ) minimiert