



Strings

Stringsuche, Boyer-Moore,
Textkompression, Huffman Codes.



Suche Substring

- Häufiges Problem
- Relevante Beispiele:
 - Suche ein Schlagwort in einem Buch
 - Alphabet: A-Za-z0-9
 - Suche Virussignatur auf der Festplatte
 - Strings über ASCII-Alphabet
 - Suche einen bestimmten Abschnitt auf einem Genom
 - Strings im Alphabet {A,D,G,T}



Text:

F I S C H E R S F R I T Z F I S C H T F I S C H E

S C A M P I

k



Gesuchtes Muster

```
static int findSubstring(String text, String muster){
    for(int i=0; i <= text.length()-muster.length(); i++){
        int k=0;
        while(k < muster.length() &&
            text.charAt(i+k) == muster.charAt(k))
            k++;
        if (k==muster.length()) return i;
    }
    return -1;
}
```



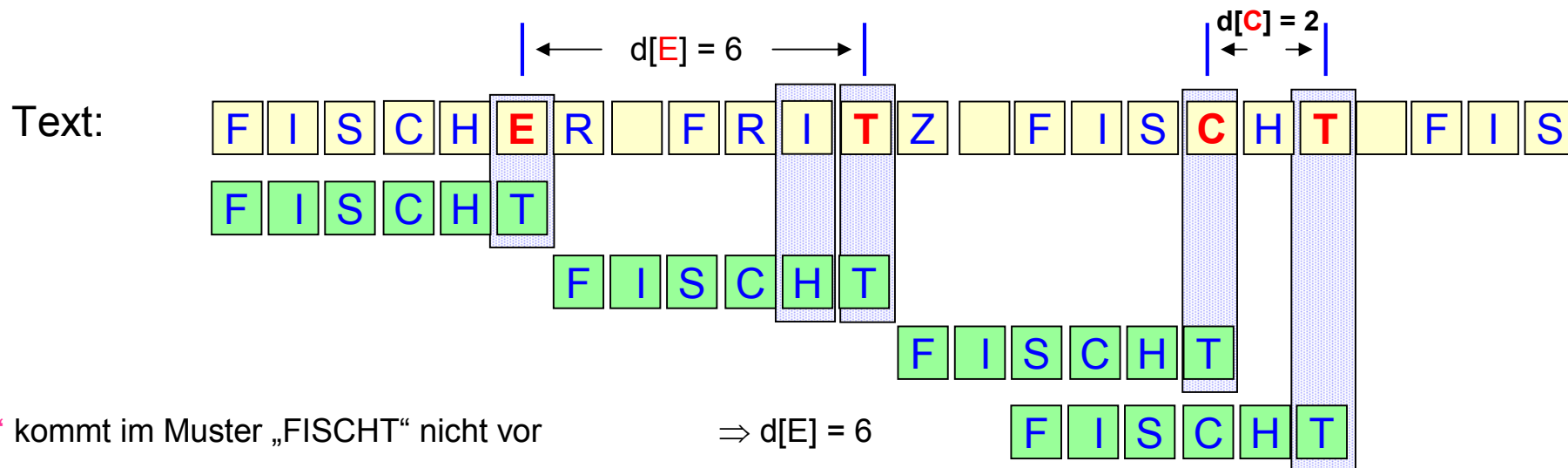
Boyer-Moore – die Idee

F I S C H T

- Durchlaufe Muster von hinten nach vorn
 - $k = \text{musterlänge} - 1$
- Vergleiche mit dem Text
 - $\text{text}[i+k] == \text{muster}[k]$
- Bei Fehlschlag schiebe Muster um **Distanz d** nach rechts

Distanzen:

| | |
|-----|-----|
| A | 6 |
| B | 6 |
| C | 2 |
| D | 6 |
| E | 6 |
| F | 5 |
| ... | ... |



- 'E' kommt im Muster „FISCHT“ nicht vor $\Rightarrow d[E] = 6$
- 'T' kommt im Muster „FISCHT“ nicht **nochmal** vor $\Rightarrow d[T] = 6$
- 'C' kommt **zwei Positionen** vor Ende vor $\Rightarrow d[C] = 2$



Boyer-Moore

E R S F R I T Z F

F I S C H T

| | |
|-----|-----|
| A | 6 |
| B | 6 |
| C | 2 |
| D | 6 |
| E | 6 |
| F | 5 |
| ... | ... |

```
static int boyerMoore(String text, String muster){
    int musterLänge = muster.length();

    // Initialisiere die Sprungtabelle */
    int [] sprungung = new int [65336];
    for(int i=0; i<65336; i++) sprungung[i]=musterLänge;
    for(int k=0; k < (musterLänge-1); k++)
        sprungung[muster.charAt(k)] = (musterLänge-1)-k;

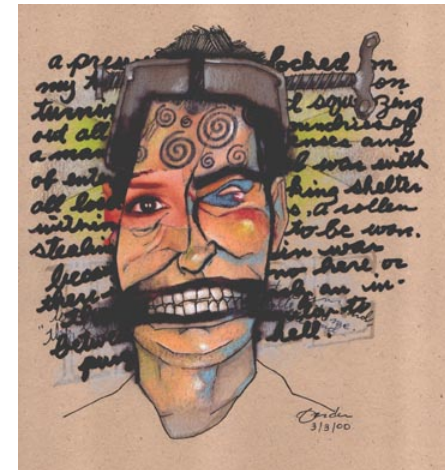
    // Suche das Muster im Text
    int i=0; // Index im Textstring
    while(i <= text.length()-musterLänge ){
        int k=musterLänge-1; // Index im MusterString
        while( k >= 0 && text.charAt(i+k)==muster.charAt(k))
            k--;
        if (k == -1) return i; // muster komplett gefunden
        else i += sprungung [text.charAt(i+musterLänge-1)];
    }
    return -1;
}
```

- Berechne Sprungtabelle:
 - $d[z]$ = Abstand von z vom rechten Ende des Musters
 - bzw. Musterlänge
- Laufe durch den Textstring
 - $i = 0 \dots$
- Laufe rückwärts durch Muster
 - $k = \text{musterlänge}-1 \dots$
- Vergleiche $\text{text}[i+k]$ mit $\text{muster}[k]$
- Falls verschieden, verschiebe Muster um Distanz $d[\text{text}[i+\text{musterlänge}-1]]$

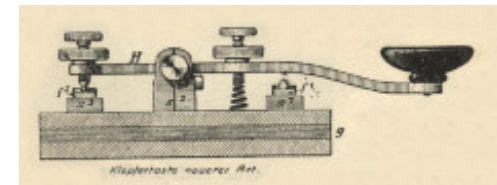


Textkompression

- In ASCII und UNICODE:
 - Jedes Zeichen verbraucht gleich viel Speicherplatz
 - Ein 'e' verbraucht genauso viel Speicherplatz wie ein 'q'
- Im Morsealphabet :
 - Häufig benutzte Zeichen haben kurzen Morsecode
 - e, t, a, i, m, n,
 - Selten benutzte Zeichen längeren
 - j, q, y,



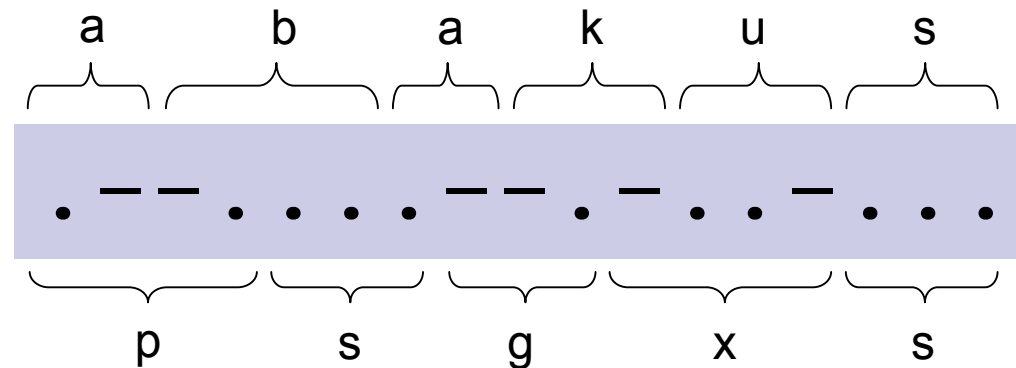
| | | | | | |
|---|------|---|------|---|------|
| a | .- | j | .--- | s | ... |
| b | -... | k | -.- | t | - |
| c | -.. | l | .-.. | u | ..- |
| d | -.. | m | -- | v | ...- |
| e | . | n | -. | w | .- |
| f | ..-. | o | --- | x | -..- |
| g | --. | p | .-. | y | -.- |
| h | | q | --.- | z | --.. |
| i | .. | r | .-. | | |





Dekodierbarkeit - Präfixcodes

- Wörter aus Morsecode sind nicht eindeutig dekodierbar



- Notwendig und hinreichend für Dekodierbarkeit (ohne Vorausschau):
 - Kein Codewort darf *Präfix* (Anfangsstück) eines anderen Codeworts sein
 - Solche Codes heißen *Präfixcodes*
- Morsecode ist *kein* Präfixcode:
 - *. -* ist Präfix von *. - - .*
- Morsecode benutzt zusätzliches Trennzeichen: **Pause**.





Huffman Codes

- Präfix Codes, bei denen häufig benutzte Zeichen kurze Codewörter haben

- Beispiel:

| | | | | | |
|---|-------|---|---------|---|--------|
| a | 1100 | j | 000011 | s | 01110 |
| b | 00101 | k | 000010 | t | 1101 |
| c | 01011 | l | 01100 | u | 01010 |
| d | 01101 | m | 01001 | v | 00010 |
| e | 111 | n | 1010 | w | 00100 |
| f | 01000 | o | 1011 | x | 000001 |
| g | 00011 | p | 00111 | y | 00110 |
| h | 01111 | q | 0000001 | z | 000000 |
| i | 1000 | r | 1001 | | |

Jede Folge von Codewörtern ist auch ohne Trennzeichen eindeutig dekodierbar

11000010111000000100101001110

a b a k u s



Konstruktion eines Huffman Codes

- Betrachte kleines Alphabet [a .. f] mit Häufigkeitstabelle

| a | b | c | d | e | f |
|-----|------|------|------|-----|------|
| 0,2 | 0,05 | 0,04 | 0,16 | 0,4 | 0,15 |

- Die seltensten Zeichen: b und c
 - müssen sich in einem Bit (z.B. dem letzten) unterscheiden.
 - Codiere : b = ?0 und c = ?1
 - ? steht hier für das *abstrakte Zeichen* „b oder c“

| | |
|---|----|
| a | |
| b | ?0 |
| c | ?1 |
| d | |
| e | |
| f | |

- Neue Häufigkeitsverteilung

| a | ? | d | e | f |
|-----|------|------|-----|------|
| 0,2 | 0,09 | 0,16 | 0,4 | 0,15 |

- Die beiden seltensten Zeichen jetzt : ? und f ...
 - Codiere: ? = #0 und f = #1
 - # steht für das neue *abstrakte Zeichen* „? oder f“
 - Häufigkeit von #: $0,09 + 0,15 = 0,24$

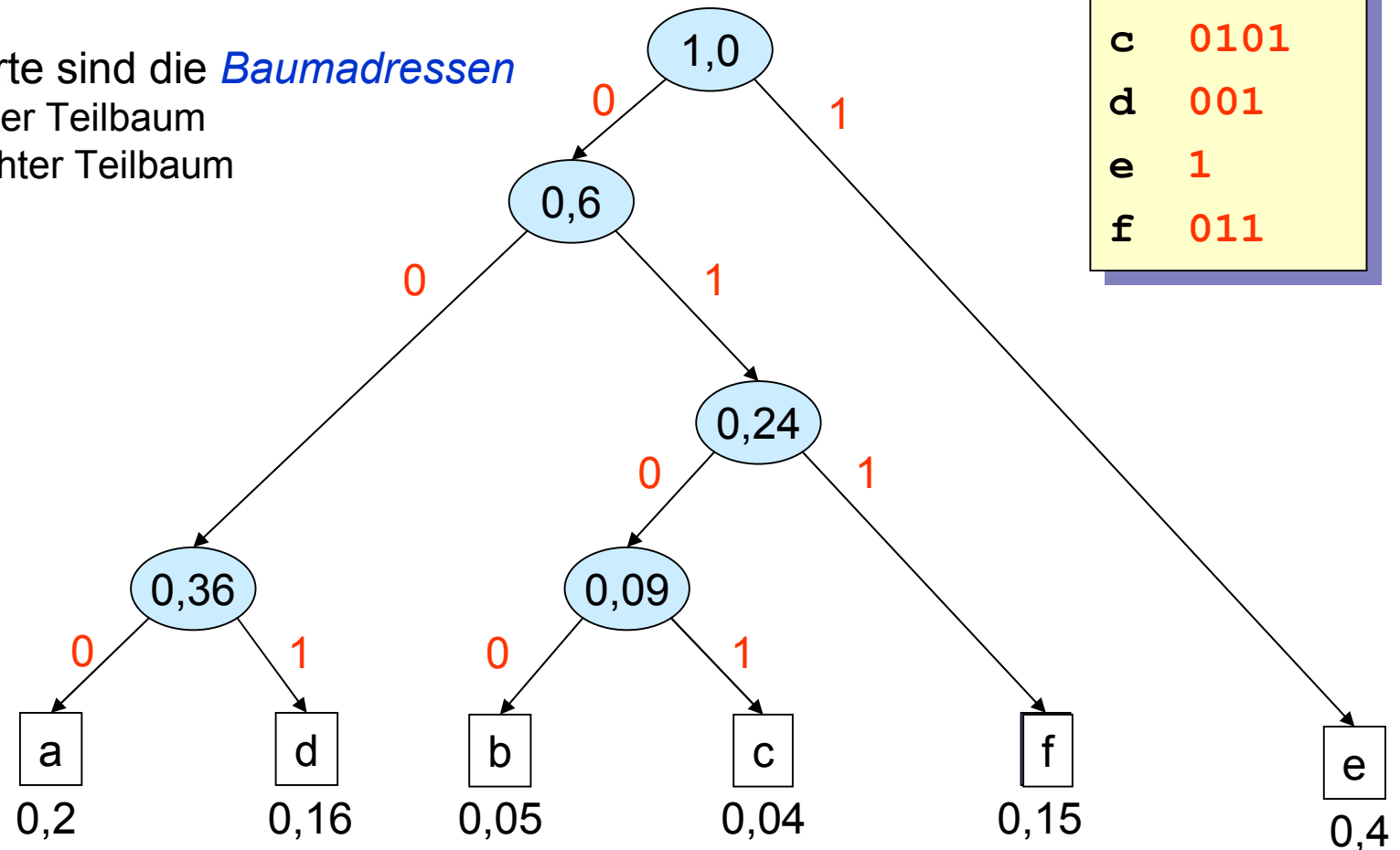
| | |
|---|-----|
| a | |
| b | #00 |
| c | #01 |
| d | |
| e | |
| f | #1 |



Huffman-Baum

- Es entsteht ein Baum
 - Knoten sind die Zeichen (bzw. abstrakten Zeichen) mit ihren Häufigkeiten
- Codeworte sind die *Baumadressen*
 - 0 linker Teilbaum
 - 1 rechter Teilbaum

| | |
|---|------|
| a | 000 |
| b | 0100 |
| c | 0101 |
| d | 001 |
| e | 1 |
| f | 011 |





Decodierung und Kompressionsrate

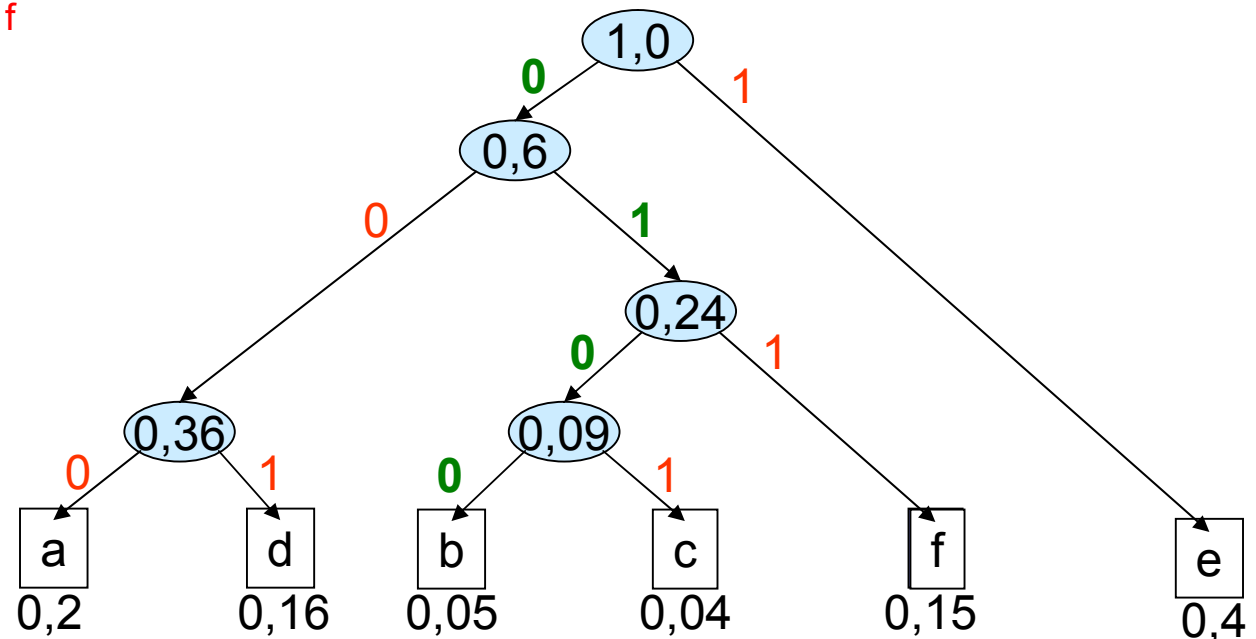
- Benutze man den Code als Wegweiser in den Baum

- 0 : in den linken Teilbaum
- 1 : in den rechten Teilbaum
- Am Blatt wurde das erste Zeichen erkannt.
- Schneide es ab und steige wieder in den Baum ein

- Beispiel

- 01001000001
- 0100 b 1 e 000 a 011 f

| | |
|---|------|
| a | 000 |
| b | 0100 |
| c | 0101 |
| d | 001 |
| e | 1 |
| f | 011 |





Kompressionsrate

■ Angenommene Häufigkeitstabelle

| a | b | c | d | e | f |
|-----|------|------|------|-----|------|
| 0,2 | 0,05 | 0,04 | 0,16 | 0,4 | 0,15 |

■ Länge der Codeworte:

| a | b | c | d | e | f |
|---|---|---|---|---|---|
| 3 | 4 | 4 | 3 | 1 | 3 |

| | |
|---|------|
| a | 000 |
| b | 0100 |
| c | 0101 |
| d | 001 |
| e | 1 |
| f | 011 |

■ Durchschnittlicher Platzverbrauch pro Zeichen :

$$3 \times 0.2 + 4 \times 0.05 + 4 \times 0.04 + 3 \times 0.16 + 1 \times 0.4 + 3 \times 0.15 = 2.29 \text{ Bit}$$



Adaptive Codes



■ Kompressionsrate

- $\text{länge(QuellString)} / \text{länge(CodeString)}$
 - hängt ab von Güte der Häufigkeitsschätzung

■ Häufigkeiten hängen ab von

- Art des Textes
 - Programmcode
 - Dokumentencodierung
 - Text
- Sprache
 - Englisch (e ist häufigster Buchstabe)
 - Finnisch (viele Vokale)
 - Tschechisch, Polnisch (viele Konsonanten)

■ Mehrpass Codes

- analysieren das Dokument
- bestimmen Häufigkeiten
- codieren
- fügen Häufigkeitstabelle dem codierten Dokument bei

■ Adaptive Codes

- ein-Pass Codes
- starten mit initialer Schätzung
- adaptieren Schätzung während der Codierung
- Huffman-Baum ändert sich
 - beim Codieren
 - analog beim Dekodieren

■ Adaptive Codes sind lernende Codes

- Ergebnis
 - kann besser sein
 - kann schlechter seinals bei statischer Codierung

Der englische Schriftsteller *Ernest Vincent Wright* schrieb einen Roman „*Gadsby*“ in dem kein einziges „e“ vorkommt.