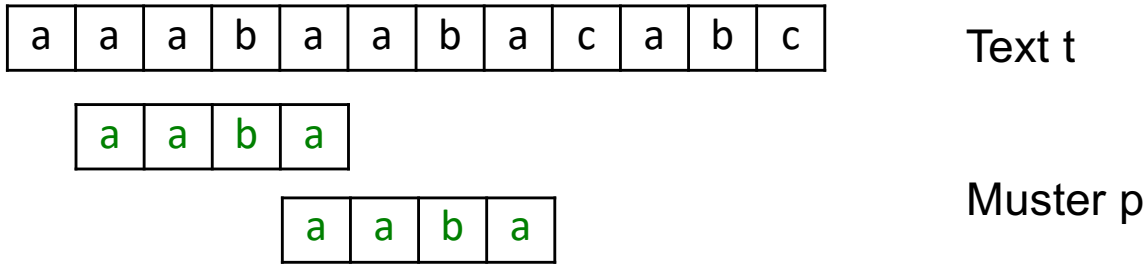
15. Suchen in Texten

- Problemstellung
- Naive Suche
- Heuristische Suche
- Knuth-Morris-Pratt-Algorithmus
- Boyer-Moore-Algorithmus
- Karp-Rabin-Algorithmus
- Approximative Suche
- Literatur

Textsuchproblem

- Finde alle Vorkommen eines Musters (pattern) p in einem (typischerweiser längerem) Text t:



- Text $t = t_o t_1 ... t_{n-1}$ und Muster $p = p_o p_1 ... p_{m-1}$ sind Zeichenketten (Strings, Wörter) über ein Alphabet A (endliche Menge von Zeichen).

- Beispiele:
  - A = {0,1} ergibt Binärfolgen
  - A = Menge der ASCII-Zeichen ergibt die gewohnten Texte
  - A = {A, G, C, T} ergibt DNA-Sequenzen

- Textsuchproblem (formal):

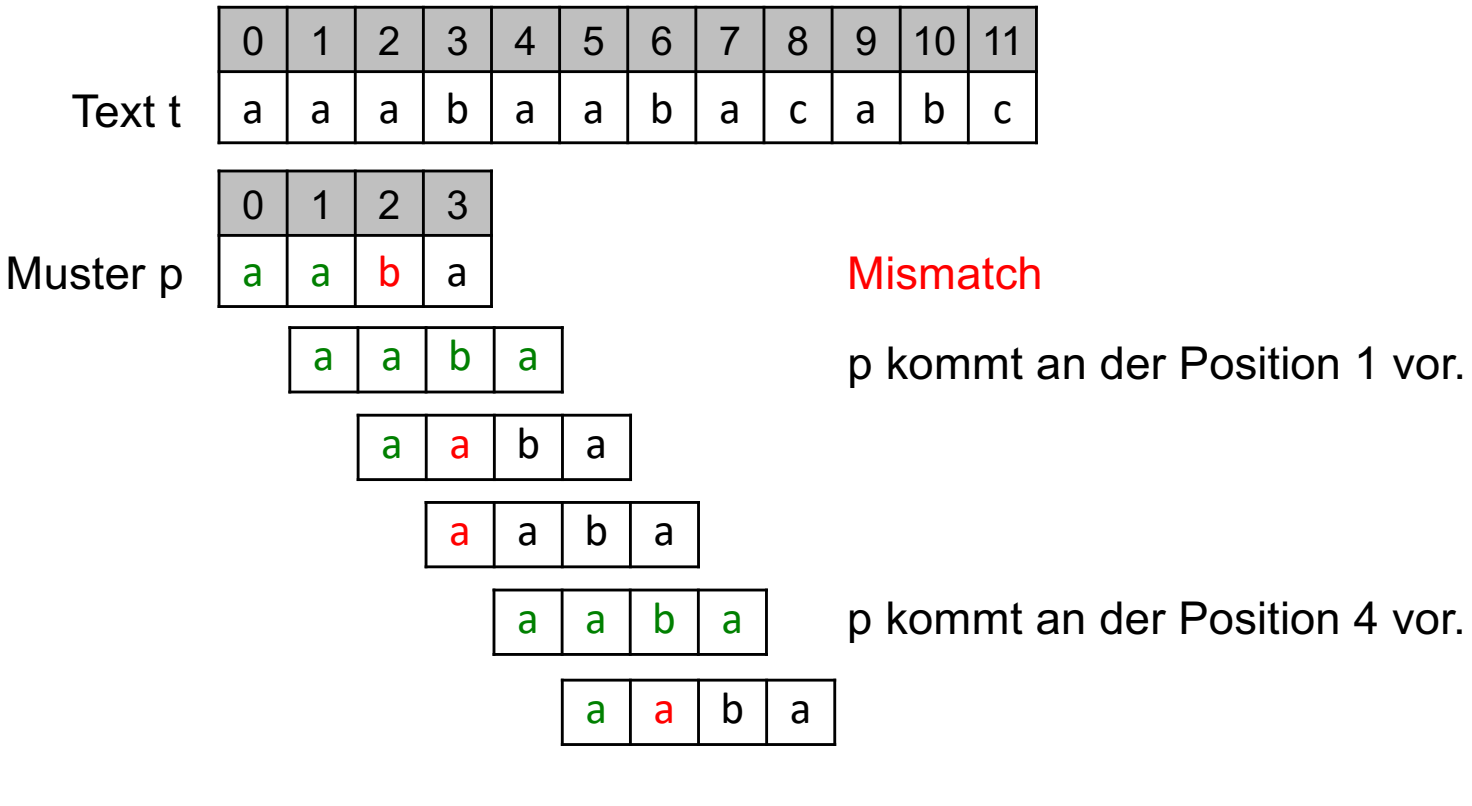
  Gegeben: Muster $p = p_o p_1 ... p_{m-1}$ und Text $t = t_o t_1 ... t_{n-1}$

  Gesucht: $\{i \leq n-m \,/\, p_o p_1 ... p_{m-1} = t_i t_{i+1} ... t_{i+m-1}\}$

  (Menge aller Textpositionen i, wo das Muster vorkommt)

Naive Suche

- An jeder Position i des Textes t wird geprüft, ob das Muster p vorkommt.



Naive Suche als Java-Methode

```
public static void naiveStringSearch(String t, String p) {
        int n = t.length();
        int m = p.length();

        for (int i = 0; i <= n-m; i++) {
            int j = 0;
            while (j < m && t.charAt(i+j) == p.charAt(j))
                j++;
            if (j == m)
                report(p,i);
        }
}
```

Muster p kommt an Textposition i vor.

Analyse der naiven Suche

- Laufzeit im schlechtesten Fall:
  - Für die Anzahl der Zeichenvergleiche gilt:

    $C(m,n) = (n-m+1)m.$
  - Damit ist

    $T(m,n) = O(m*n)$
  - Beispiel: t = aaaaaaaaaaaaa und p = aaaab

- Laufzeit im durchschnittlichen Fall:
  - Sei h_j die Häufigkeit des Zeichens p_j im Text t.
  - Dann ist die durchschnittliche Anzahl der Zeichenvergleiche
    für jede Textposition i (durchschnittliche Durchlauflänge der inneren while-Schleife):

    $1 + h_0 + h_0h_1 + ... + h_0h_1...h_{m-2}$

    $\leq 1 + h + h^2 + ... + h^{m-1} \leq 1/(1-h)$ (geometrische Reihe)

    wobei h die Häufigkeit des häufigsten Zeichens ist.
    Z.B. ist in deutschen Texten e der häufigste Buchstabe mit etwa h = 0.17.
  - Für die durchschnittliche Anzahl der Zeichenvergleiche folgt:

    $C(m,n) \leq (n-m+1)/(1-h).$
  - Damit ist

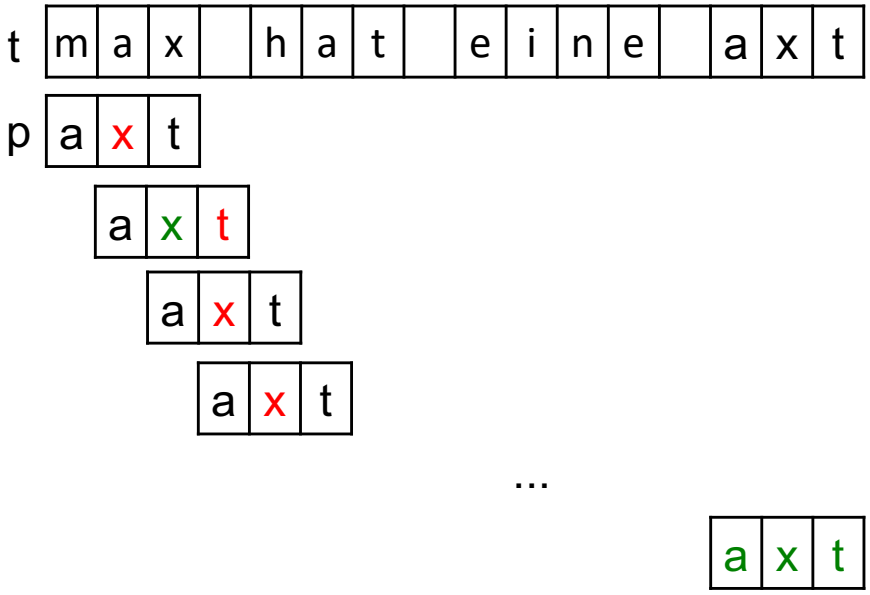
    $T(m,n) = O(n)$

14. Suchen in Texten

- Problemstellung
- Naive Suche
- Heuristische Suche
- Knuth-Morris-Pratt-Algorithmus
- Boyer-Moore-Algorithmus
- Karp-Rabin-Algorithmus
- Approximative Suche
- Literatur

Idee der heuristischen Suche

- wähle Musterzeichen mit der geringsten Auftrittshäufigkeit im Text zuerst und prüfe auf Gleichheit mit Textzeichen.
- dazu ist Wissen über die Häufigkeit der Zeichen im Text notwendig.
- statt die Häufigkeit der Zeichen im Text zu ermitteln wird zurückgegriffen auf empirisch ermittelte Häufigkeiten in einer repräsentativen Menge von Texten. Für Texte in einer gegeben Sprache wie z.B. deutschsprachige Texte sind Häufigkeiten bekannt.
- Beispiel



Buchstabe	Häufigkeit in dt. Texten
x	0.03 %
t	6.1 %
a	6.5 %

Bei Muster p = "axt" prüfe zuerst x, dann t und dann a.

Algorithmus

```
public static void heuristicStringSearch(String t, String p) {
        int n = t.length();
        int m = p.length();

        int[ ] r = new int[m];
        for (int j = 0; j < m; j++) r[j] = j;
        sortiere r nach den Häufigkeiten der Musterzeichen aufsteigend;

        for (int i = 0; i <= n-m; i++) {
                int j = 0;
                while (j < m && t.charAt( i + p[ r[j] ] ) == p.charAt( p[ r[j] ] ) )
                        j++;
                if (j == m)
                        report(p,i);
        }
}
```

- Feld r enthält die Indizes des Musters p in der Reihenfolge, in der die Zeichen verglichen werden sollen.
- Beispiel:

	0	1	2
p	a	x	t

r	1	2	0

vergleiche zuerst p[1] = x, dann p[2] = t und dann p[0] = a.

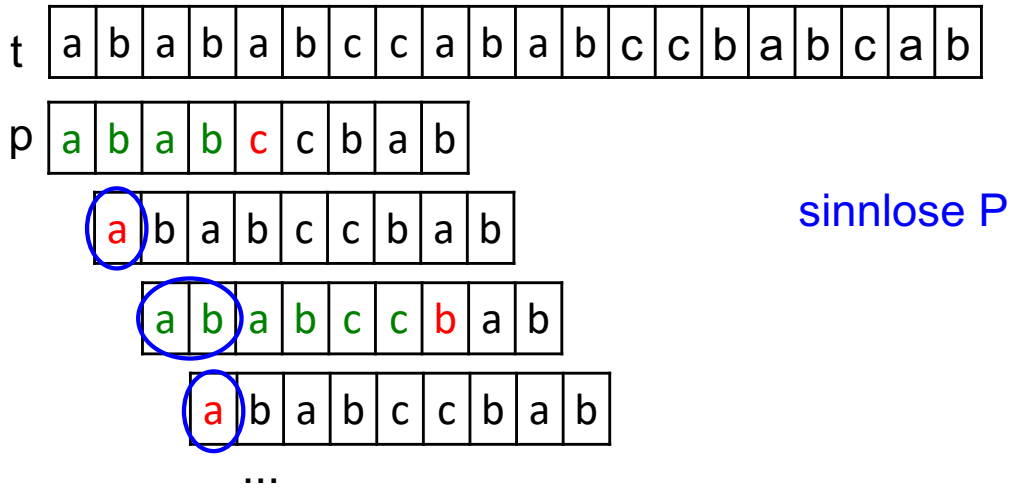
- Sind alle Zeichen gleichwahrscheinlich, dann ergibt sich keine Verbesserung gegenüber dem naiven Verfahren.
- Kommt jedoch im Muster ein seltenes Zeichen vor (z.B. x), dann ergibt sich eine signifikante Verbesserung.

14. Suchen in Texten

- Problemstellung
- Naive Suche
- Heuristische Suche
- Knuth-Morris-Pratt-Algorithmus
- Boyer-Moore-Algorithmus
- Karp-Rabin-Algorithmus
- Approximative Suche
- Literatur

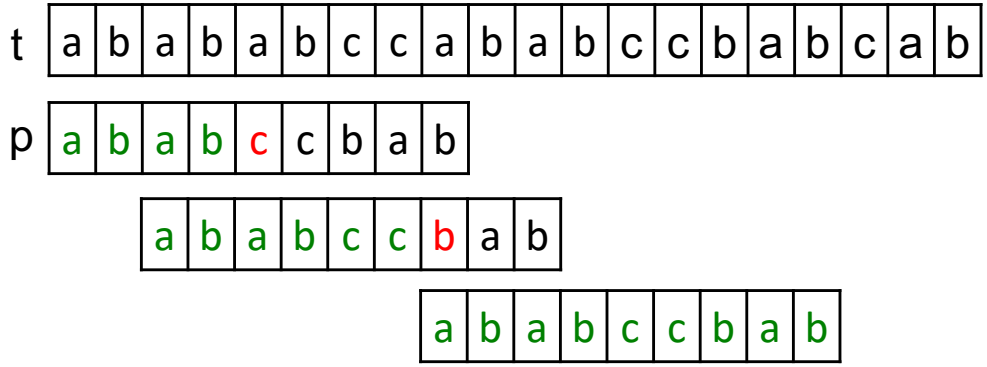
Idee

- Die naive Textsuche schiebt bei einem Mismatch das Muster um genau ein Zeichen weiter. Zuvor erfolgreich durchgeführte Vergleiche werden vergessen.



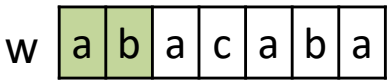
sinnlose Prüfungen.

- Der Algorithmus von Knuth-Morris-Pratt analysiert das Muster in einer Vorlaufphase und speichert Informationen über seine Struktur ab. Damit wird versucht, das Muster um mehr als ein Zeichen nach rechts zu verschieben. Sinnlose Prüfungen werden vermieden.

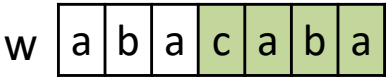


Präfix, Suffix und Rand

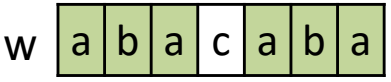
- Die leere Zeichenkette (Zeichenkette der Länge 0) wird mit ε bezeichnet.
- Eine Zeichenkette u ist ein Präfix (Anfangsstück) einer Zeichenkette w, falls w = uv für eine Zeichenkette v.
- Eine Zeichenkette u ist ein Suffix (Endstück) einer Zeichenkette w, falls w = vu für eine Zeichenkette v.
- Für eine beliebige Zeichenkette w ist sowohl ε als auch w Präfix und Suffix von w.
- Ein Präfix u bzw. Suffix u einer Zeichenkette w ist echt, falls u ≠ w ist.
- Ein Rand einer Zeichenkette w ist eine Zeichenkette, die sowohl echter Präfix als auch echter Suffix von w ist. Die Anzahl der Zeichen im Rand heisst auch Breite des Rands.



ab ist Präfix von w.
Alle Präfixe von w: ε, a, ab, aba, abac, ..., abacaba.



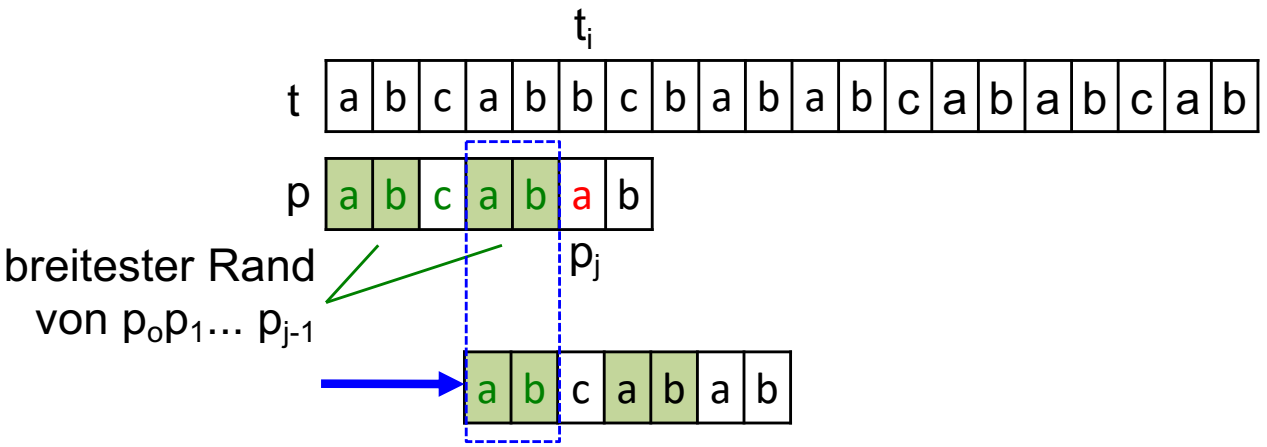
caba ist Suffix von w.
Alle Suffixe von w: ε, a, ba, aba, caba, ..., abacaba.



aba ist Rand von w mit Breite 3.
Alle Ränder von w: ε, a, aba.

Knuth-Morris-Pratt-Algorithmus (1)

- Musterzeichen und Textzeichen werden zeichenweise wie beim naiven Verfahren verglichen.
- Falls das Musterzeichen p_j nicht mit dem Textzeichen t_i übereinstimmt (Mismatch), dann muss der Präfix $p_o p_1 \ldots p_{j-1}$ und das Textstück $t_{i-j} t_{i-j+1} \ldots t_{i-1}$ übereinstimmen.
- Bestimme den breitesten Rand von $p_o p_1 \ldots p_{j-1}$ und verschiebe Muster nach rechts, so dass beide Randteile übereinanderliegen.



- In einer Vorlaufphase werden für alle Präfixe von p die breitesten Ränder bestimmt. lbl[j] = Größe des breitesten Rands von $p_o p_1 \ldots p_{j-1}$ (lbl = longest border length). Aus technischen Gründen wird lbl[0] = -1 gesetzt.
- Beispiel

p	a	b	c	a	b	a	b	
lbl	-1	0	0	0	1	2	1	2

Knuth-Morris-Pratt-Algorithmus (2)

```
private static int[ ] lbl;

public static void kmpStringSearch(String p, String t) {
        kmpPreprocess(p);

        int n = t.length();
        int m = p.length();

        int i = 0;
        int j = 0;

        while (i < n) {
                while (j >= 0 && t.charAt(i) != p.charAt(j))
                        j = lbl[j];
                i++;
                j++;
                if (j == m) {
                        report(p,i-m);
                        j = lbl[j];
                }
        }
}
```

kmpPreprocess berechnet lbl (Größe der breitesten Ränder für alle Präfixe von p

Solange Mismatch verschiebe Muster auf nächsten breitesten Rand.

Text- und Musterzeichen stimmen überein. Gehe zu nächstem Zeichen in Text und Muster.

Muster gefunden. Verschiebe Muster auf nächsten breitesten Rand.

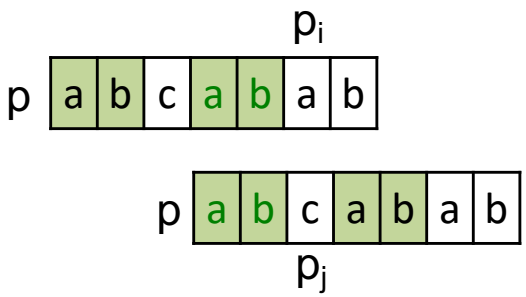
Preprocessing

- Im Vorlauf muss für jeden Präfix von p die Größe des breitesten Rands bestimmt werden.
- Dazu wird p mit der Knuth-Morris-Pratt-Textsuche über sich selbst (!) verschoben.
- Immer wenn Musterzeichen p_i und p_j übereinstimmen, ist ein breitester Rand bestimmt und seine Größe kann in das Feld lbl eingetragen werden.

```
public static void kmpPreprocessing(String p) {
        int m = p.length();
        lbl = new int[m+1];
        lbl[0] = -1;

        int i = 0;
        int j = -1;

        while (i < m) {
                while (j >= 0 && p.charAt(i) != p.charAt(j))
                        j = lbl[j];
                i++; j++;
                lbl[i] = j;
        }
}
```

Hier gilt: $p_o p_1 \ldots p_{j-1} = p_{i-j} p_{i-j+1} \ldots p_{i-1}$.

Damit hat der breiteste Rand von $p_o p_1 \ldots p_{i-1}$ die Größe j.



Analyse

- In der äußeren while-Schleife werden i und j genau n-mal inkrementiert.
  In der inneren while-Schleife wird j durch die Zuweisung j = lbl[j] verkleinert.
  Das kann jedoch nur so lange geschehen, wie zuvor j inkrementiert wurde.
- Damit ergibt sich für die äußere while-Schleife einen Aufwand von O(n).
- Mit der gleichen Argumentation benötigt kmpPreprocess O(m).
- Insgesamt folgt für die KMP-String-Suche: T(n,m) = O(n+m).

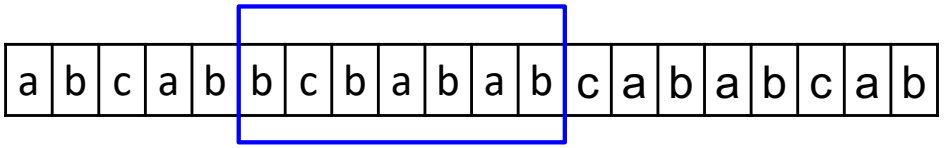
```
public static void kmpStringSearch(String p, String t) {
        kmpPreprocess(p);
        int n = t.length();
        int m = p.length();
        int i = 0;
        int j = 0;

        while (i < n) {
                while (j >= 0 && t.charAt(i) != p.charAt(j))
                        j = lbl[j];
                i++; j++;
                if (j == m) {
                        report(p,i-m);
                        j = lbl[j];
                }
        }
}
```

Besondere Eignung für Textdateien

- Das KMP-Verfahren ist besonders geschickt bei Textdateien:

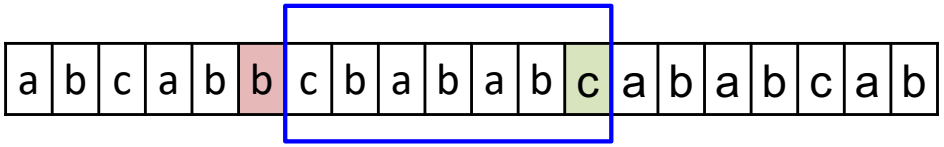
  Das KMP-Verfahren erfordert kein Zurückspringen im Text.
  Daher ist keine Pufferung des Textes notwendig.

- Vergleiche dazu das naive Verfahren:
  - Nach einem nicht erfolgreichen Vergleich eines Textzeichens mit einem Musterzeichen muss wieder auf ein früheres Textzeichen zurückgesprungen werden.
  - Da das Zurückspringen in einer Datei nicht unterstützt wird, muss jeweils ein Textausschnitt der Länge m in einem Ringpuffer (zirkuläres Feld) gespeichert werden.



Ringpuffer für Textauschnitt
der Größe m

Datei mit Text t.



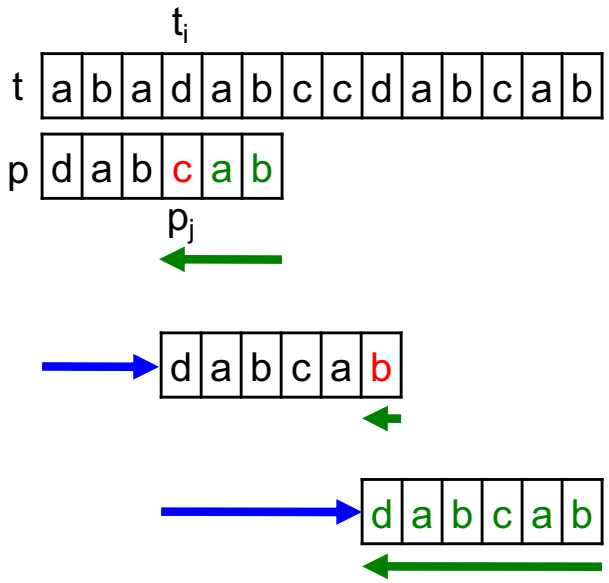
Gehe im Text ein Zeichen weiter.
Im Ringpuffer kommt neues Zeichen dazu
und ältestes Zeichen wird gelöscht

14. Suchen in Texten

- Problemstellung
- Naive Suche
- Heuristische Suche
- Knuth-Morris-Pratt-Algorithmus
- Boyer-Moore-Algorithmus
- Karp-Rabin-Algorithmus
- Approximative Suche
- Literatur

Idee

- Das Muster wird von rechts nach links (!) mit den Textzeichen verglichen.
- Bei einem Mismatch von t_i und p_j wird das Muster möglichst weit nach rechts verschoben.



Vergleichen

Verschieben

- Dabei wird bei einem Verschiebe-Schritt zwischen zwei Strategien entschieden:

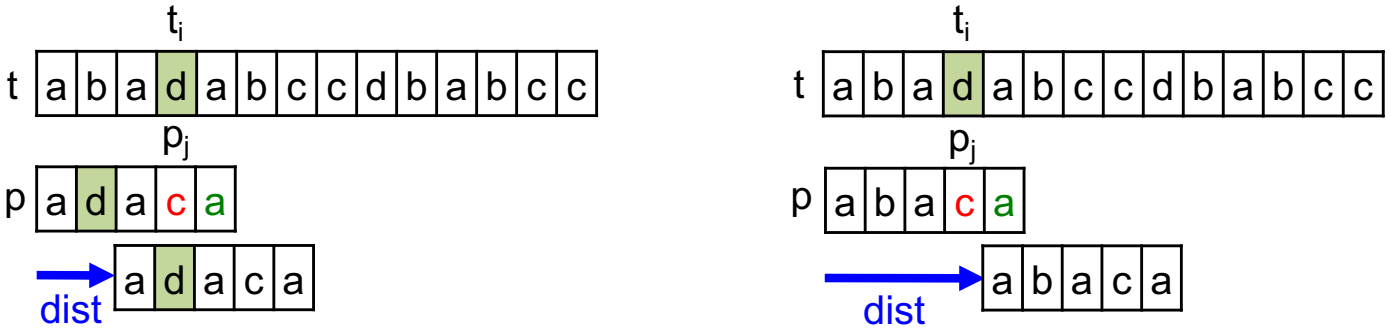
  (1) Bad-Character-Strategie

  (2) Good-Suffix-Strategie

- Die jeweiligen Verschiebedistanzen werden ähnlich wie bei Knuth-Morris-Pratt durch eine Analyse des Musters in einer Vorlaufphase ermittelt.

Bad-Character-Strategie

- Bei einem Mismatch bei t_i (bad character) wird das Muster p bis zum letzten Vorkommen von t_i in p verschoben. Falls t_i in p nicht vorkommt, wird p_0 auf t_{i+1} geschoben.



- Dazu wird in einer Vorlaufphase für jedes Zeichen c des Alphabets der Index des letzten Vorkommens (last occurrence) von c in p bestimmt (-1, falls c nicht vorkommt)

```
private static int[ ] lastOcc;

private static void bmPreprocessCharacterOccurrence(String p) {
        int m = p.length();
        lastOcc = new int[256];
        for (int c = 0; c < 256; c++)
                lastOcc[c] = -1;
        for (int j = 0; j < m; j++)
                lastOcc[p.charAt(j)] = j;
}
```

Beispiel:

0	1	2	3	4
a	b	a	c	a

...	'a'	'b'	'c'	...
-1	4	1	3	-1

lastOcc

- Die Verschiebdistanz ergibt sich als dist = j – lastOcc[t[i]].
  Beachte: dist kann negativ werden, was jedoch die Good-Suffix-Strategie kompensiert.

Good-Suffix-Strategie

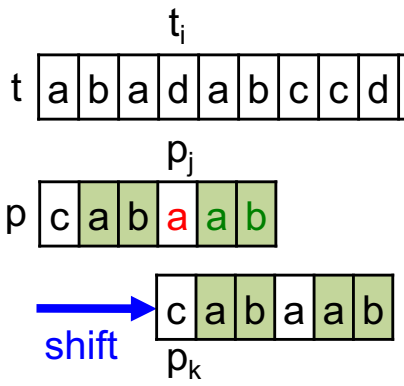
- Bei einem Mismatch von t_i und p_j wird das Muster soweit nach rechts verschoben bis

  (Fall 1) das bisher als richtig erkannte Suffix (good suffix) $p_{j+1}p_{j+2}\ldots p_{m-1}$ mit einem Präfix von $p_{k+1}p_{k+2}\ldots p_{m-1}$, mit $0 \le k < j$ und $p_k \neq p_j$ übereinstimmt, oder

  (Fall 2) ein Suffix des als richtig erkannten Suffix mit einem Präfix des Musters übereinstimmt.

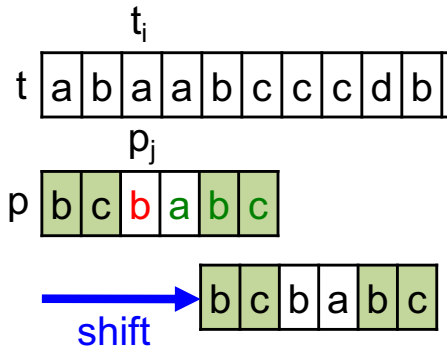
- Dazu wird in einer Vorlaufphase für jede Musterposition j die Verschiebedistanz in shift[j] gespeichert.

(Fall 1)



0	1	2	3	4	5	j
c	a	b	a	a	b	p
6	6	6	3	6	1	shift

(Fall 2)



0	1	2	3	4	5	j
b	c	b	a	b	c	p
4	4	4	4	6	1	shift

Boyer-Moore-Algorithmus

```
private static int[ ] lastOcc;
private static int[ ] shift;

public static void kmpStringSearch(String p, String t) {
        bmPreprocessCharacterOccurrence(p);
        bmPreprocessShift1(p);
        bmPreprocessShift2(p);

        int n = t.length();
        int m = p.length();

        int i = 0;
        while (i <= n-m) {
                int j = m-1;
                while (j >= 0 && t.charAt(i+j) == p.charAt(j)) j--;
                if (j == -1) {
                        report(p,i);
                        i += shift[0];
                }
                else
                        i += Math.max(shift[j], j - lastOcc[t.charAt(i)]);
        }
}
```

Vorlauf für die Bad-Character-Strategie. lastOcc wird berechnet.

Der Vorlauf für die Good-Suffix-Strategie geschieht in zwei Schritten (Fall (1) und Fall (2)). Dabei wird shift berechnet.

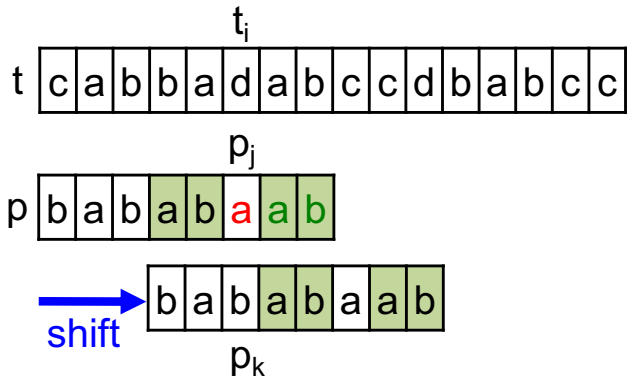
Vergleiche Muster und Text von rechts nach links.

Muster gefunden. Verschiebe Muster mit Good-Suffix-Strategie.

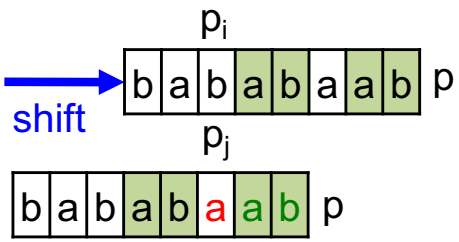
Mismatch. Verschiebe Muster nach günstigerer Strategie.

Preprocessing für Good-Suffix-Strategie – Fall 1 (1)

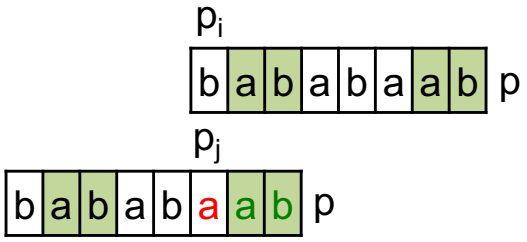
- Bei einem Mismatch von t_i und p_j wird das Muster soweit nach rechts verschoben bis das bisher als richtig erkannte Suffix (good suffix) $p_{j+1}p_{j+2}\ldots p_{m-1}$ mit einem Präfix von $p_{k+1}p_{k+2}\ldots p_{m-1}$, mit $k \geq 0$ und $p_k \neq p_j$ übereinstimmt:



- Dazu wird im Vorlauf das Muster p über sich selbst verschoben und von rechts nach links verglichen.
- Bei einem Mismatch von p_j und p_i wird shift[j] = j – i gesetzt (aber nur wenn zuvor noch kein shift-Wert eingetragen wurde, d.h. shift[j] != 0)



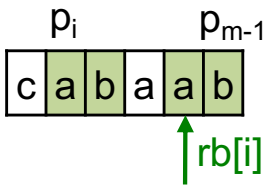
Da zuvor shift[j] == 0 (für j = 5),
wird shift[j] = j - i = 5 - 2 = 3 gesetzt.



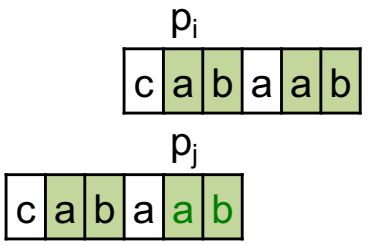
Da zuvor shift[j] == 3 (für j = 5),
bleibt shift[j] unverändert.

Preprocessing für Good-Suffix-Strategie – Fall 1 (2)

- Zusätzlich wird eine Tabelle rb berechnet:
  rb[i] = Beginn des breitesten rechten Rands von $p_i p_{j+1} \ldots p_{m-1}$.



- Falls der breiteste Rand leer ist, wird rb[i] = m gesetzt.
- Solange beim Verschieben des Musters über sich selbst p_j und p_i übereinstimmen, gilt rb[i] = j.



Es wird rb[i] = j gesetzt,
wobei i = 1 und j = 4.

- rb wird benutzt um bei einem Mismatch das Muster auf den nächsten rechten Rand zu verschieben.

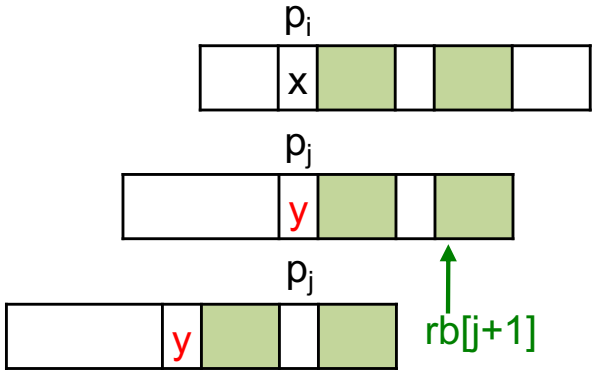
Preprocessing für Good-Suffix-Strategie – Fall 1 (3)

```
private static int[ ] shift;
private static int[ ] rb;

private static void bmPreprocessGoodSuffixShift1(String p) {
        int m = p.length();
        shift = new int[m];
        rb = new int[m];

        int i = m-2;
        int j = m-1;
        rb[m-1] = m;

        while (i >= 0) {
                while (j < m && p.charAt(i) != p.charAt(j)) {
                        if (shift[j] == 0)
                                shift[j] = j-i;
                        j = j < m-1 ? rb[j+1]-1 : m;
                }
                rb[i--] = j--;
        }
}
```

Siehe Seite 15-22.

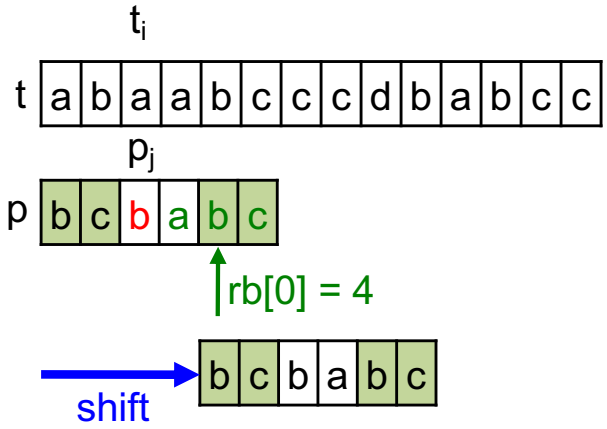
Siehe Seite 15-23.

- Bei einem Mismatch wird das Muster auf den nächsten rechten Rand verschoben.



Preprocessing für Good-Suffix-Strategie – Fall 2 (1)

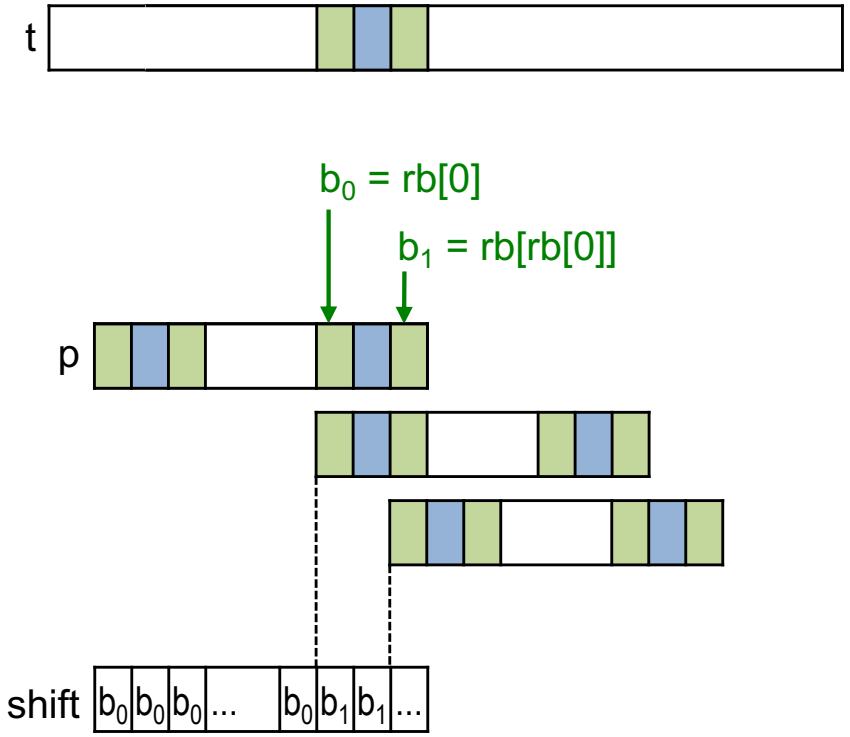
- Wenn in Schritt 1 für eine Musterposition j keine Verschiebedistanz gefunden wurde (d.h. shift[j] == 0), dann wird p auf den Beginn des breitesten rechten Rands (d.h. shift[j] = rb[0]) gesetzt.




```
private static int[ ] shift;
private static int[ ] rb;

private static
void bmPreprocessGoodSuffixShift2(String p) {
        int m = p.length();

        int b = rb[0];
        for (int j = 0; j < m; j++) {
                if (j == b)
                        b = rb[b];
                if (shift[j]==0)
                        shift[j] = b;
        }
}
```

Preprocessing für Good-Suffix-Strategie – Fall 2 (2)

- Sobald j den breitesten rechten Rand b erreicht hat, dann muss auf den nächsten breitesten Rand umgeschalten werden (d.h. b = rb[b]).




```
private static int[ ] shift;
private static int[ ] rb;

private static
void bmPreprocessGoodSuffixShift2(String p) {
        int m = p.length();

        int b = rb[0];
        for (int j = 0; j < m; j++) {
                if (j == b)
                        b = rb[b];
                if (shift[j]==0)
                        shift[j] = b;
        }
}
```

Beispiele

0	1	2	3	4	5	j
c	a	b	a	a	b	p
0	0	0	3	0	1	shift nach Prepocessing Fall 1
6	4	5	6	6	6	rb
6	6	6	3	6	1	shift nach Prepocessing Fall 2

0	1	2	j
a	b	c	p
0	0	1	shift nach Prepocessing Fall 1
3	3	3	rb
3	3	1	shift nach Prepocessing Fall 2

0	1	2	3	4	j
a	a	a	a	a	p
0	0	0	0	0	shift nach Prepocessing Fall 1
1	2	3	4	5	rb
1	2	3	4	5	shift nach Prepocessing Fall 2

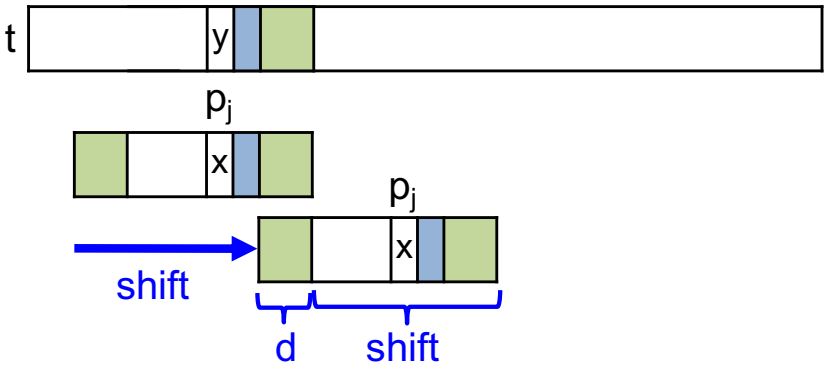
0	1	2	3	4	5	6	7	8	9	10	11	12	13	j
a	a	b	a	a	c	b	a	a	a	a	b	a	a	p
0	0	0	0	0	0	0	0	0	0	5	3	1	2	shift nach Prepocessing Fall 1
9	10	11	12	13	14	11	12	12	12	13	14	13	14	rb
9	9	9	9	9	9	9	9	9	12	5	3	1	2	shift nach Prepocessing Fall 2

Optimierung nach Galil

- Immer wenn eine Verschiebung aufgrund der Good-Suffix-Strategie Fall (2) zustandekommt, genügt es im nächsten Durchlauf das Muster nur bis

  $j \geq d = m\text{-}shift$

  zu prüfen.
- Fall (2) der Good-Suffix-Strategie liegt genau dann vor, wenn $j < shift$.




```
public static void kmpStringSearch(String p, String t) {
        bmPreprocessCharacterOccurrence(p);
        bmPreprocessShift1(p);
        bmPreprocessShift2(p);
        int n = t.length();
        int m = p.length();

        int d = 0;
        int i = 0;
        while (i <= n-m) {
                int j = m-1;
                while (j >= d && t.charAt(i+j) == p.charAt(j))
                        j--;
                if (j < d) {
                        report(p,i);
                        d = m - shift[0];
                        i += shift[0];
                }
                else {
                        int s = shift[j];
                        d = j < s ? m - s : 0;
                        i += Math.max(s, j-lastOcc[t.charAt(i)]);
                }
        }
}
```

Analyse

- bmPreprocessCharacterOccurrence benötigt O(m),
  falls die Alphabetgröße als konstant vorausgesetzt wird.
- bmPreprocessGoodSuffixShift2 benötigt O(m).
- In bmPreprocessGoodSuffixShift1 wird in der inneren while-Schleife j maximal so oft vergrößert werden, wie j in der äußeren Schleife dekrementiert werden kann. Da i und damit auch j maximal m-mal dekrementiert werden, benötigt bmPreprocessGoodSuffixShift1 ebenfalls O(m).
- Die BM-String-Suche mit der Galil-Optimierung benötigt T(n,m) = O(n+m).
  Der nicht ganz einfache Beweis ist z.B. in [Gusfield 97] zu finden.
- Das Boyer-Moore-Verfahren ist in der Praxis außerordentlich schnell:
  Kommen die Zeichen im Muster p im Text t kaum vor, dann wird das Muster p bei einem Mismatch fast immer um m Zeichen nach rechts verschoben.
  Man erhält dann eine sublineare Laufzeit O(n/m).

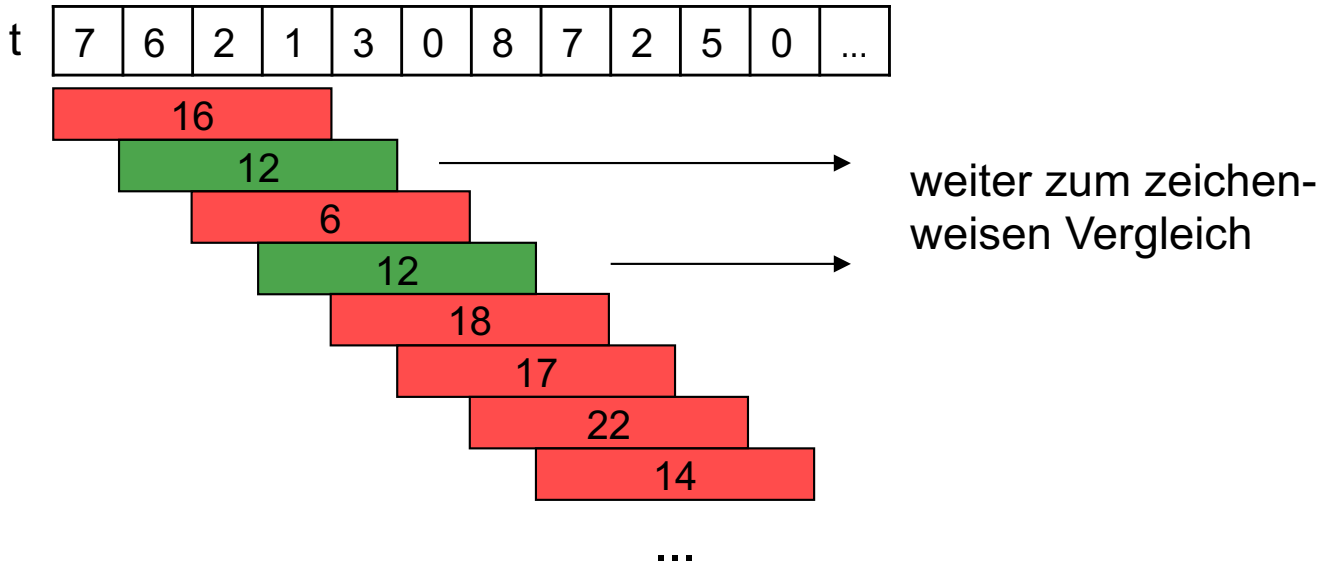
14. Suchen in Texten

- Problemstellung
- Naive Suche
- Heuristische Suche
- Knuth-Morris-Pratt-Algorithmus
- Boyer-Moore-Algorithmus
- Karp-Rabin-Algorithmus
- Approximative Suche
- Literatur

Idee

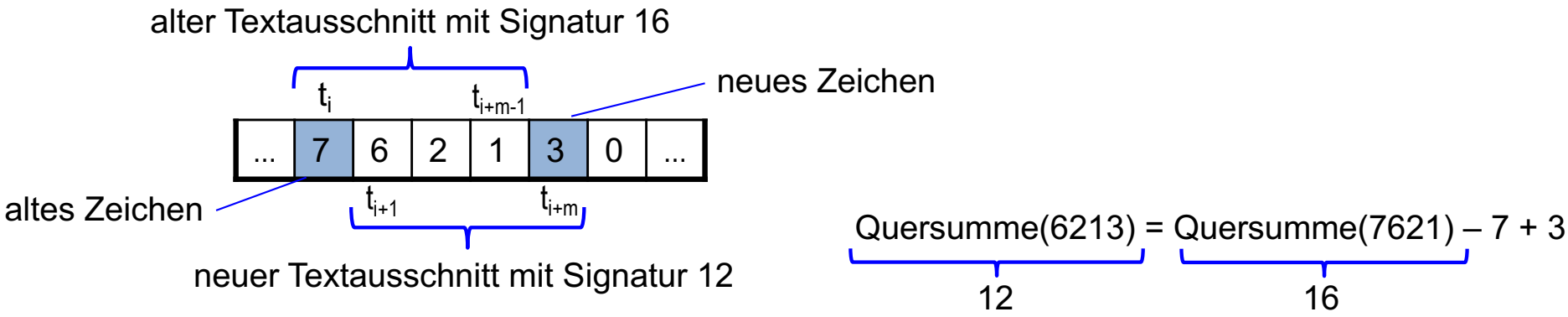
- Wie beim naiven Algorithmus wird das Muster mit jedem Textausschnitt verglichen.
- Statt jedoch Muster und Textausschnitt zeichenweise zu vergleichen, werden zuvor aus dem Textausschnitt und aus dem Muster Signaturen (numerische Werte; ähnlich wie Hashwerte) berechnet und diese verglichen.
  Bei Übereinstimmung wird ein zeichenweiser Vergleich durchfgeführt.

- Beispiel:
  Alphabet = {0, 1, ..., 9} und Muster p = 1308
  Signaturfunktion ist Quersumme. Für p ergibt sich damit die Signatur 12.



Signaturfunktion

- Eine Signaturfunktion s bildet eine Zeichenkette u auf einen ganzzahligen Wert ab. Der Wert s(u) ist die Signatur von u.
- Zwei Zeichenketten u und v bilden eine Kollision, falls $u \neq v$, aber $s(u) = s(v)$.
- Anforderungen an eine Signaturfunktion:
  - Kollisionen sollten möglichst ausgeschlossen sein.
  - Signatur muß in konstanter Zeit berechenbar sein.
    Berechne dazu Signaturwert des neuen Textausschnitt $t_{i+1} \ldots t_{i+m}$
    aus Signaturwert des alten Textauschnitts $t_i \ldots t_{i+m-1}$.
- Beispiel Quersumme:
  Signatur ist zwar in konstanter Zeit berechenbar;
  leider treten jedoch viele Kollisionen auf.



Signaturfunktion des Karp-Rabin-Algorithmus

- Für eine Zeichenkette $u = u_0u_1 \ldots u_{m-1}$ wird eine Signatur definiert:

$$s(u_0u_1 \ldots u_{m-1}) = u_0*2^{m-1} + u_1*2^{m-2} + \ldots + u_{m-1}*2^0 \mod 2^{32}$$

- Die Signatur s‘ des neuen Textauschnitts $t_{i+1} \ldots t_{i+m}$ lässt sich aus der Signatur s des vorhergehenden Textauschnitts $t_i \ldots t_{i+m-1}$ wie folgt berechnen:

$$s' = (2 * (s - 2^{m-1} t_i) + t_{i+m}) \mod 2^{32}$$

- Die Signatur für das Muster p lässt sich mittels dem Horner-Schema besonders geschickt berechnen:

```
public static int patternSignature(String p) {
        int m = p.length();
        int sp = p.charAt(0);

        for (int j = 1; j < m; j++)
                sp = (sp << 1) + p.charAt(j);

        return sp;
}
```

Multiplikation mit 2 durch Binär-Links-Shift <<.

Mod 2^{32} wird implizit durch die 32-Bit-Darstellung für int bewirkt.

- Im schlimmsten Fall (z.B. p = aaa...a und t = aaa...a) ist T(n,m) = O(nm).
  Im Durchschnitt ist aber T(n,m) = O(n).

14. Suchen in Texten

- Problemstellung
- Naive Suche
- Heuristische Suche
- Knuth-Morris-Pratt-Algorithmus
- Boyer-Moore-Algorithmus
- Karp-Rabin-Algorithmus
- Approximative Suche
- Literatur

Editierabstand

- Der Editierabstand (Levenshtein-Distanz) zwischen zwei Zeichenketten

  $u = u_0 u_1 ... u_{n-1}$

  $v = v_0 v_1 ... b_{m-1}$

  ist die minimale Anzahl an Editieroperationen
  - Zeichen löschen,
  - Zeichen einfügen und
  - Zeichen ändern,

  um u in v überzuführen.

- Beispiel

ANANAS	
BANANAS	B einfügen
BANANA	S löschen ,
BANANE	letztes A in E ändern

  ANANAS kann mit 3 Editieroperationen in BANANE geändert werden.
  Es gibt keine andere Sequenz mit weniger als 3 Editieroperationen.
  Damit ist der Editierabstand = 3.

- Berechnung des Editierabstands mit dynamischer Programmierung
  - Editierabstand rekursiv formulieren
  - Editierabstand damit tabellenartig berechnen.

Rekursive Formulierung des Editierabstands

- Für die beiden Zeichenketten $u = u_0u_1...u_{n-1}$ und $v = v_0v_1...v_{m-1}$ definieren wir

  $D(i,j)$ = Editierabstand von $u_0u_1...u_{i-1}$ und $v_0v_1...v_{j-1}$.

- D lässt sich rekursiv formulieren:
  - Basisfälle (i = 0 oder j = 0):
    (i = 0 und j = 0 bedeuten leere Zeichenketten)

    $D(0, j) = j$ ($v_0v_1...v_{j-1}$ in u einfügen)

    $D(i, 0) = i$ ($u_0u_1...u_{i-1}$ löschen)
  - Rekursionfall (i, j > 0):
    D(i, j) = Minimum aus:

    (1) $D(i-1, j) + 1$ (u_{i-1} löschen)

    (2) $D(i, j-1) + 1$ (v_{j-1} nach u_{i-1} einfügen)

    (3a) $D(i-1, j-1) + 1$, falls $u_{i-1} \neq v_{j-1}$ (u_{i-1} zu v_{j-1} ändern)

    (3b) $D(i-1, j-1)$, falls $u_{i-1} = v_{j-1}$

Tabellenartige Berechnung des Editierabstands

D(i,j)

		v_j	B	A	N	A	N	E
	i\j	0	1	2	3	4	5	6
u_i	0	0	1	2	3	4	5	6
A	1	1	1	1	2	3	4	5
N	2	2	2	2	1	2	3	4
A	3	3	3	2	2	1	2	3
N	4	4	4	3	2	2	1	2
A	5	5	5	4	3	2	2	2
S	6	6	6	5	4	3	3	3

Basisfall

Rekursionsfall

D(i,j) ist Minimum aus:

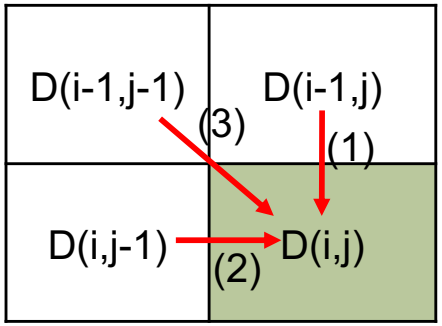
(1) D(i-1, j) + 1

(2) D(i, j-1) + 1

(3a) D(i-1, j-1) + 1, falls $u_{i-1} \neq v_{j-1}$

(3b) D(i-1, j-1), falls $u_{i-1} = v_{j-1}$

Minimaler Editierabstand zwischen BANANE und ANANAS ist 3.



Bestimmung der Editieroperationen (1)

- Wie lassen sich aus der berechneten Tabelle die notwendigen Editieroperationen ablesen, um $u = u_0u_1...u_{n-1}$ in $v = v_0v_1...v_{m-1}$ überzuführen?
- Starte bei D(n,m) und speichere in eine Editierliste, wie der jeweilige Tabelleneintrag zustande kam (d.h. welcher der Fälle (1), (2), (3a) oder (3b) vorlag).
- Beende bei Tabelleneintrag D(0,0).

D(i,j)

	v_j	B	A	N	A	N	E
u_i	i\j	0	1	2	3	4	5	6
	0	0	1	2	3	4	5	6
A	1	1	1	1	2	3	4	5
N	2	2	2	2	1	2	3	4
A	3	3	3	2	2	1	2	3
N	4	4	4	3	2	2	1	2
A	5	5	5	4	3	2	2	2
S	6	6	6	5	4	3	3	3

D(n,m)

↓ : Fall (1); Minimum ist D(i-1, j) + 1.

→ : Fall (2); Minimum ist D(i, j-1) + 1.

↘ : Fall (3a); es ist $u_{i-1} \neq v_{j-1}$ und Minimum ist D(i-1, j-1) + 1

⇢ : Fall (3b); es ist $u_{i-1} = v_{j-1}$ und Minimum ist D(i-1, j-1).

Tabelleneinträge für Basisfälle werden wie Fall (1) bzw. (2) behandelt.

Bestimmung der Editieroperationen (2)

- Beachte, dass das Minimum nicht immer eindeutig ist und somit verschiedene Editierlisten möglich sind.

D(i,j)

		v_j	B	A	N	A	N	E
	i\j	0	1	2	3	4	5	6
u_i	0	0	1	2	3	4	5	6
A	1	1	1	1	2	3	4	5
N	2	2	2	2	1	2	3	4
A	3	3	3	2	2	1	2	3
N	4	4	4	3	2	2	1	2
A	5	5	5	4	3	2	2	2
S	6	6	6	5	4	3	3	3

D(i,j)

		v_j	B	A	N	A	N	E
	i\j	0	1	2	3	4	5	6
u_i	0	0	1	2	3	4	5	6
A	1	1	1	1	2	3	4	5
N	2	2	2	2	1	2	3	4
A	3	3	3	2	2	1	2	3
N	4	4	4	3	2	2	1	2
A	5	5	5	4	3	2	2	2
S	6	6	6	5	4	3	3	3

Bestimmung der Editieroperationen (3)

- Die Editerliste kann direkt als Folge von Editieroperationen interpretiert werden.
- Setze Cursor i auf u_0 (d.h. i = 0) und setze Cursor j auf v_0 (d.h. j = 0) .
- Durchlaufe Editierliste und führe dabei folgende Aktionen durch:
  - ↓ Zeichen u_i löschen;
  - → Zeichen v_j vor u_i einfügen; i++; j++;
  - ↘ Dann ist $u_i \neq v_j$: u_i durch v_j ersetzen; i++; j++;
  - ⇢ Dann ist $u_i = v_j$: i++; j++;

D(i,j)

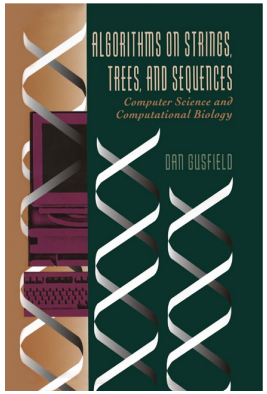
	v_j	B	A	N	A	N	E
u_i	i\j	0	1	2	3	4	5	6
	0	0	1	2	3	4	5	6
A	1	1	1	1	2	3	4	5
N	2	2	2	2	1	2	3	4
A	3	3	3	2	2	1	2	3
N	4	4	4	3	2	2	1	2
A	5	5	5	4	3	2	2	2
S	6	6	6	5	4	3	3	3

u	v	Editieroperationen
ANANAS	BANANE	
BANANAS	BANANE	→ B einfügen
BANANAS	BANANE	⇢
BANANAS	BANANE	⇢
BANANAS	BANANE	⇢
BANANAS	BANANE	⇢
BANANES	BANANE_	↘ A durch E ersetzen
BANANE_	BANANE_	↓ S löschen

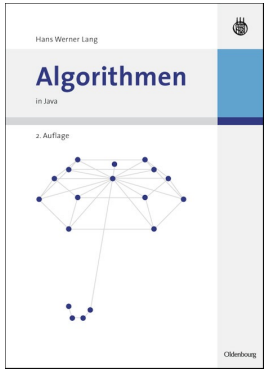
14. Suchen in Texten

- Problemstellung
- Naive Suche
- Heuristische Suche
- Knuth-Morris-Pratt-Algorithmus
- Boyer-Moore-Algorithmus
- Karp-Rabin-Algorithmus
- Approximative Suche
- Literatur

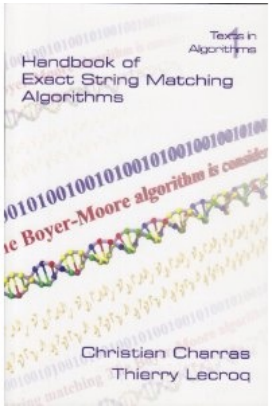
Spezialliteratur



- Dan Gusfield,
  Algorithms on Strings, Trees and Sequences: Computational Science and Biology,
  Cambridge University Press, 1997.
- Sehr umfangreiche Sammlung von Algorithmen,
  die sehr weit über den Vorlesungsstoff hinausgeht.



- Hans Werner Lang, *Algorithmen in Java,*
  Oldenbourg Wissenschaftsverlag, 2. Auflage; 2006.
- Sehr gute Darstellung verschiedener Algorithmen
  zur Textsuche, die es auch Online gibt:
  http://www.iti.fh-flensburg.de/lang/algorithmen/pattern/index.htm



- C. Charras and T. Lecroq,
  Handbook of Exact String Matching Algorithm,
  Kings College Pub, 2004.
- Sehr umfangreiche Sammlung von Algorithmen zur Textsuche.
- Buch und weitere Materialien gibt es in elektronischer Form
  auch unter http://www-igm.univ-mlv.fr/~lecroq/string/