6. Algorithmen auf Zeichenketten

6.1 Einführung

6.2 Der "naive" Suchalgorithmus

6.3 Das Suchverfahren von Knuth-Morris-Pratt

6.4 Das Suchverfahren von Boyer-Moore

6.5 Das Suchverfahren von Rabin-Karp

6.6 Pattern Matching und Editierdistanz

6.7 Komprimierung von Zeichenketten

6.1 Einführung

Bisher behandelte Datenstrukturen zeichnen sich häufig durch ihre

Gliederung in logisch unabhängige Datensätze

(jeder mit gleichem, komplexem Aufbau) aus, auf welche entsprechende Algorithmen bei der Verarbeitung zugreifen.

Neue Fragestellungen ergeben sich für andersartige Datenstrukturen, die nicht in den vorherigen Rahmen passen. Das sind zum Beispiel extrem flache Datenstrukturen wie

Texte, die als
↓
Folge von Zeichen über einem Alphabet
↓ bzw.
lineare Reihung von einzelnen Zeichen ohne etwa die Struktur eines Arrays aufgefaßt werden können.

Texte sind die zentrale Datenstruktur

- beim Editieren
- Volltext - Datenbanken im Information Retrieval
- bei Systemen zur Symbolmanipulation (z.B. Computer-Algebra)
- als Ankerdokument in Hypermedia-Systemen (z.B. im WWW)

und viele mehr.

Eigenschaften von Zeichenketten (Text)

Merkmale von Texten im Sinne von Zeichenfolgen (Character-Strings) sind einmal ihre **Strukturlosigkeit** („Flachheit“) und zum anderen folgende Verarbeitungs-Algorithmen:

- Suchen
- Suchen und Anpassen/ Pattern Matching
- Komprimieren und Dekomprimieren
- Analysieren (Parsing) und Übersetzen
- Verschlüsseln und Entziffern (Kryptographie)

Historisch hat sich die Thematik der Verarbeitung von Zeichenketten beim Entwerfen von Editiersystemen entwickelt.

Wichtigste Fragestellung:

Mit welchem Algorithmus kann man am schnellsten M vorgegebene Zeichen - das Muster- in einem Text der Länge N auffinden?

Einige Meilensteine auf dem Weg zu einer praktikablen Beantwortung:

- der naive, intuitiv naheliegende und zeitlose Algorithmus mit der Laufzeit O(N*M), er ist schnell implementiert und leicht an spezielle Hardware anzupassen (Ausnutzung von Befehlen auf Maschinenebene)
- um 1970 bewies S.A.Cook: es gibt einen Algorithmus mit der Laufzeit O(N+M)
- Boyer/Moore-Algorithmus mit einem optimalen Verhalten O(N+M)
- Knuth-Morris-Pratt: Umsetzung der Cook'schen Beweisführung in einen praktikablen Algorithmus
- 1980 Algorithmus von Rabin/Karp mit der Laufzeit O(N+M) und einer simplen Implementierungsstruktur

6.2 Der naive Such-Algorithmus

Problemstellung

Ausgangspunkt ist eine Textzeichenfolge

a[1..N]; a[i] char,

in welcher nach dem ersten Auftreten des Textmusters

p[1..M]; p[i] char, M≤N

gesucht wird.

Alle Zeichen a[i] des Textes und alle Zeichen p[i] des Musters stammen aus einem Alphabet Σ.

Intuitiv kann man schnell einen naiven Algorithmus angeben - der zunächst am Beispiel demonstriert werden soll:

p[1..4] = "aber"

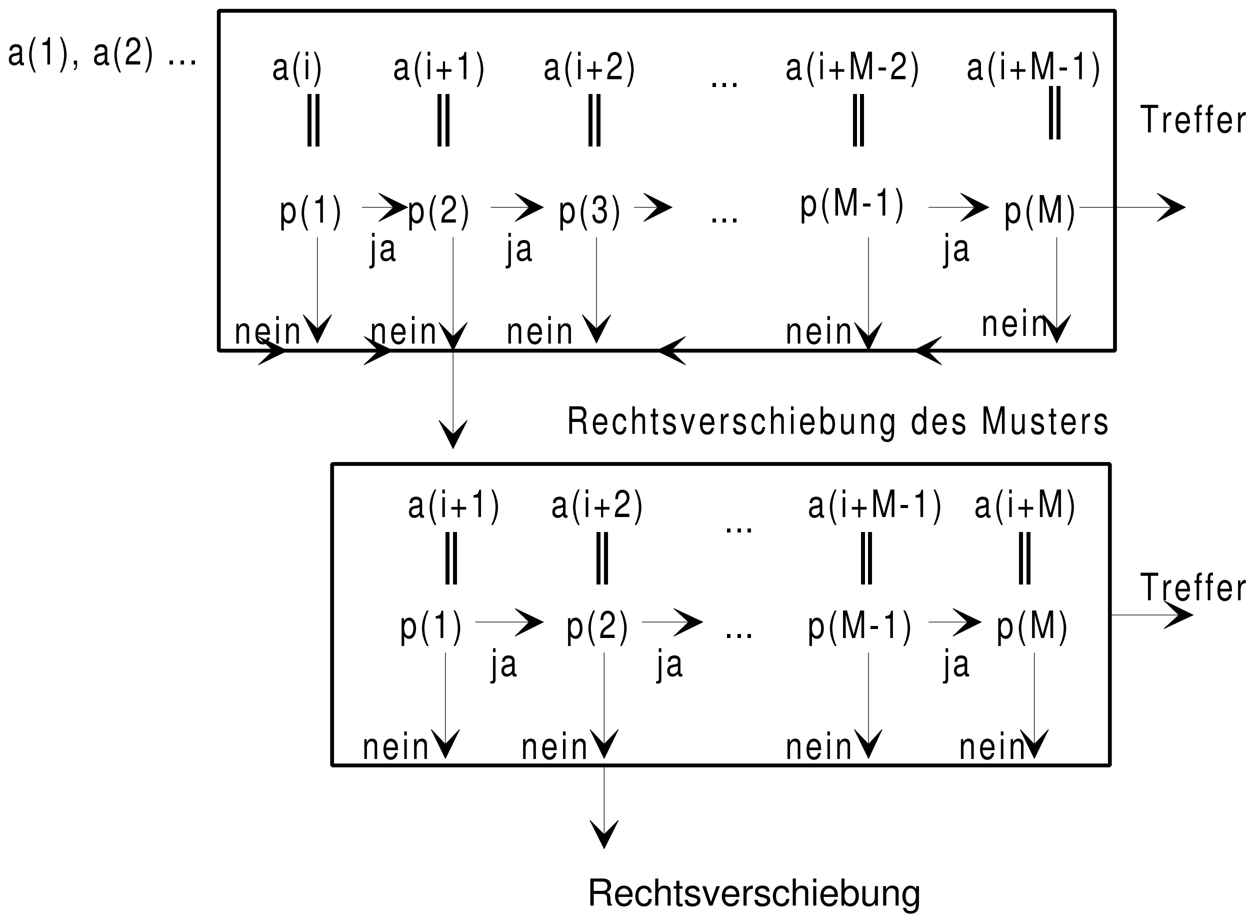
a[1..50] = "er sprach abrakadabra es bewegte sich aber nichts"

N = 50

M = 4

```
er sprach abrakadabra, es bewegte sich aber nichts
aber
 aber
  aber ...
          aber
           aber
                                      aber
                                       aber
```

Wenn i ($1 \leq i \leq N$) die aktuelle Position im String a[1..N] ist, und mit $i+M-1 \leq N$ das Muster noch innerhalb des zu untersuchenden Textes liegen muß, dann entspricht die im Beispiel skizzierte Vorgehensweise dem folgenden, allgemeineren Algorithmus:



Naive Suche, Programm

```
int naivesearch(char *p, char *a)
       {
              int i, j, M=strlen(p), N=strlen(a);
              for (i=0, j=0; j<M&&i<N; i++,j++)
                     while (a[i]!=p[j]) {i-=j-1;
j=0; }
              if (j==M) return i-M; else return i;
       }
```

Naive Suche, Komplexität

Bei einer Musterlänge M ist die letzte Position für das Anlegen im Text durch den Index

$$I = N-M+1$$

gegeben, so daß insgesamt I*M Zeichenvergleiche durchgeführt werden müssen.

Die **Laufzeit** des naiven Verfahrens ist durch

$$I * M = M * (N-M+1)$$

bestimmt, die naive Suche ist also

$$O(N*M).$$

Eine Verbesserung dieses Programmstücks ist nach unserem gerade skizzierten Verfahren leicht möglich - man läßt die innere Schleife nur bis zum ersten fehlgeschlagenen Vergleich (MISMATCH) laufen.

In der Praxis hat man in normalen Texten im überwiegenden Fall den MISMATCH schon in der ersten Position, so daß eine Laufzeit der Größenordnung O(M+N) vorliegt.

Bei Binärdateien finden sich viele Mismatches erst später innerhalb der inneren Schleife (viele „Fehlstarts“, also haben wir die schlechtere Laufzeit O(N*M) zu erwarten.

Naive Suche, Verbesserung

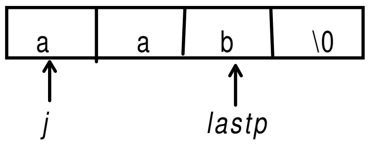
Eine relativ einfache und wirksame Verbesserung von **naivesearch** erhalten wir durch die Vorab-Prüfung beim ersten und **letzten** Musterelement.

```
int naivesearch2 (char *a, char *p)
{
/*Passe das letzte Zeichen des Musters zuerst an,
                                        und passe dann
vom Anfang an an */
        int i, j, start=0;
        int lasts=strlen(a)-1;
        int lastp=strlen(p)-1;
        int endmatch=lastp;

        for(i=0; endmatch <=lasts; endmatch++,start++){
                if(a[endmatch] == p[lastp])
                        for (j=0, i=start; j<lastp &&

                       a[i]==p[j];i++,j++)
                                        ;
                if (j==lastp)
                               return start; /*
successful */
                }
                return -1;
}
```

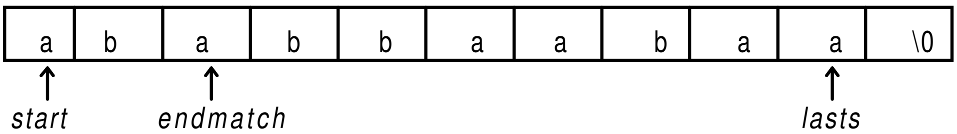
Zunächst vergleicht **naivesearch2** die beiden Enden a[endmatch] mit p[lastp], um sich erst anschließend wie **naivesearch** zu verhalten.

Beispiel zu naivesearch2

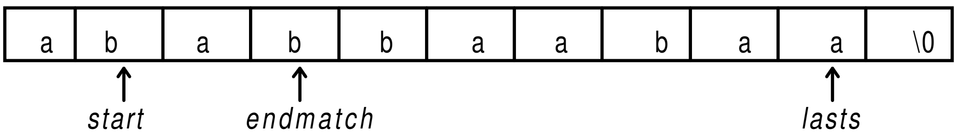
mit p = "aab", a = "ababbaabaa"



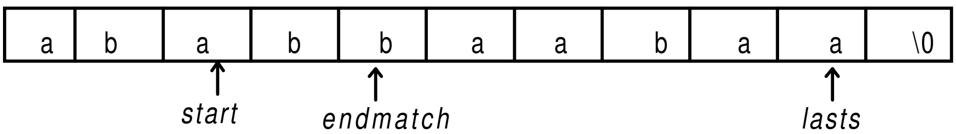
(a) Muster



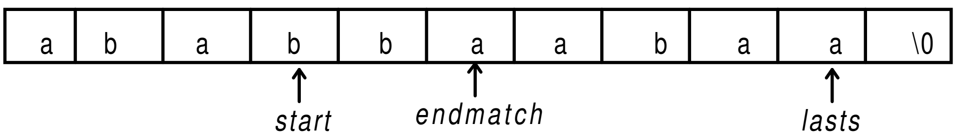
(b) Keine Übereinstimmung



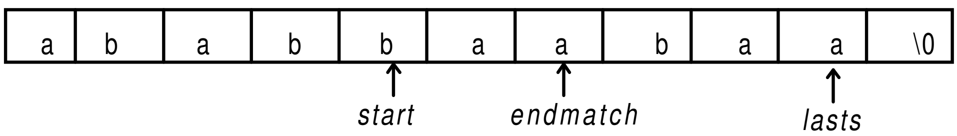
(c) Keine Übereinstimmung



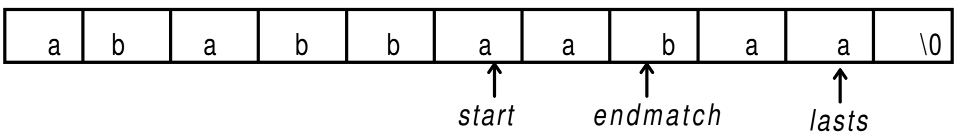
(d) Keine Übereinstimmung



(e) Keine Übereinstimmung



(f) Keine Übereinstimmung



(g) Keine Übereinstimmung

Naive Suche, Kommentare

- leicht verständlich (naiv)
- keine komplizierte Musteraufbereitung erforderlich
- einer Reihe von implementierungstechnischen Verbesserungen leicht möglich
- bei normalen Texten (ASCII-Alphabet) im Mittel eine Laufzeit $< M*N$, wenn $M << N$; die innere Mustervergleichsschleife wird selten ganz durchlaufen
- im ungünstigsten Fall, zum Beispiel bei Binärdateien, ist eine sehr ungünstige Laufzeit $O(N*M)$ zu erwarten.

6.3 Das Suchverfahren von Knuth-Morris-Pratt

Ideal wäre ein Algorithmus, der schon von der Theorie her in der Zeit

$$O(N+M)$$

arbeitet und bei $M<<N$ das Optimum $O(N)$ liefert.

Um dies zu erreichen, gilt es zu vermeiden, daß **eine Textstelle mehrfach inspiziert wird**, denn beim naiven Ansatz wird vergessen, welche Übereinstimmung schon erzielt wurde:

er ist ohne Gedächtnis

Idee von Knuth-Morris-Pratt

Die Idee, die dem von Knuth, Morris und Pratt entdeckten Algorithmus zugrunde liegt, ist folgende: Wenn eine Nichtübereinstimmung festgestellt wird, besteht unser „Fehlstart“ aus Zeichen, die wir bereits kennen (da sie sich im Muster befinden). Irgendwie sollte es uns gelingen, aus dieser Information einen Vorteil zu ziehen, anstatt den Zeiger i über alle diese bekannten Zeichen hinweg zurückzusetzen.

KMP-Suche, Beispiel 1

Text	a	b	a	b	b	a	b	a
Muster	a	b	a	b	a	b		
Position	0	1	2	3	4	5		
					↑ Mismatch entdeckt			

naives Verfahren

Text	a	b	a	b	b	a	b	a
Muster		a	b	a	b	a	b	

eine Position gewonnen

verbessertes Verfahren KMP, Rechtsverschiebung um 2

Text	a	b	a	b	b	a	b	a
Muster			a	b	a	b	a	b

zwei Positionen gewonnen

Da wir schon fünf Positionen verglichen haben, von denen vier übereingestimmt haben, wissen wir sehr viel über die aktuelle Stelle des Vergleichens. Dieses Wissen nutzen wir aus, um eine schnellere Rechtsverschiebung zu erreichen.

Ziel:
Unter Ausnutzung der Informationen eines teilweise erfolgreichen Vergleichs allein aus der Musterstruktur eine maximale Rechtsverschiebung zu bestimmen.

KMP-Suche, Beispiel 2

↓Vergleichsposition

Text I N F O R M A T I K

Pattern I N F O S T A N D

verschobenes Pattern I N F O S T A N D

Treten am Anfang des Pattern Wiederholungen von Zeichenfolgen auf, ist die zulässige Verschiebung geringer:

↓Vergleichsposition

Text G E G E N S A T Z

Pattern G E G E B E N

verschobenes Pattern G E G E B E N

Die zulässigen Verschiebungen werden *vor* dem Suchvorgang aus dem Pattern hergeleitet und in einer *Tabelle* gespeichert. Die Elemente der Tabelle geben jeweils an, an welche Stelle links der Ungleichheit das Pattern verschoben werden kann. Die Stelle im Text, an der verglichen werden muß, bleibt gleich oder wird um eine Position nach rechts verschoben, wenn die neue Position des Patternanfangs *rechts* von der Vergleichsposition steht.

Ideen zur Fehlschlagsfunktion

- **Muster**: p[0] p[1] p[2] ... p[j] ...p[M-1]

  mögliche

  Mismatchstell j = 0, 1, 2, ..., M-1

- **Auffinden** möglicher identischer Teilmusterstücke links in der Mismatchposition j durch das Schema

	p[0]	p[1]	p[2]	...			p[j-1]	p[j]	
	p[0]	p[1]	p[2]				p[j-1]	p[j]	
	.							.	
	.							.	
	.							.	
k ➔	p[0]	p[1]	...	p[k-1]	p[k]	...		p[j]	← k’
	.							.	
	.							.	
	.							.	
	p[0]	p[1]	...	p[k-1]	...		p[j-1]	p[j]	

(k, k’ = j-k) Zeilennummern gleicher Musterlänge

Man erkennt:

Die untere Dreiecksmatrix enthält alle Teilstücke, welche von Anfang p[0], und die obere Dreiecksmatrix alle diejenigen, welche vom Ende p[j] gebildet wurden.

- Sei k die Zeile in der unteren Dreiecksmatrix mit der **maximalen Musterlänge**, die auch ein entsprechendes (**gleiches**) Element in der oberen Dreiecksmatrix (Zeile k = j-k) besitzt.
  Dann nimmt die Fehlschlagsfunktion **fail(j)** den Indexwert k-1 des am weitesten rechts stehenden Elementes an.
  In allen anderen Fällen gilt fail(j) = -1; dies ist zunächst willkürlich, erleichtert aber den Neustart von p[0] aus

- Die Wiederanlaufsposition j' bei festgehaltener Textstelle i (aktueller, letzter Mismatch) ergibt sich durch j' = fail(j-1)+1. Dies liegt im wesentlichen daran, daß man die Übereinstimmung von p[0] ... p[j-1] nicht vergessen will.

Beispiel

Textposition					i				
Text	a	b	a	b	b	a	b	a	b
Muster	a	b	a	b	a	b			
Musterposition j	0	1	2	3	4	5			

↑

Mismatch j = 4

Matrixbildung für j-1=3

	a	b	a	b		
k = 2->	a	b		a	b	k’ = 2
		a	b	a		b

Wiederanlaufposition

fail(j-1) = k-1 = 2-1 = 1

j' = fail(j-1)+1 = 1+1 = 2

also folgt

Textposition					i				
Text	a	b	a	b	b	a	b	a	b
Muster			a	b	a	b	a	b	
Musterposition j			0	1	2	3	4	5	

Matrixbildung für j-1 = 1

a b haben keine

gemeinsamen Teilketten

fail(1) = -1

Wiederaufsetzen mit p[0]} = a an der Stextstelle i

Text a b a b

b a b a b

Muster a

b a b a b

Musterposition j 0 1

2 3 4 5

Situation wie beim naiven Algorithmus

fail(0) = -1

d.h. Weiterschieben um eine Position

Text a b a b

b a b a b

Muster

a b a b a

Musterposition j 0

1 2 3 4

Ableitung der Fehlschlagsfunktion fail(j)

- p[0] p[1] p[2] ... p[j] ...p[M-1]
  ist das M-komponentige Muster
- Für die Position j= 0 wird immer gesetzt fail(0)=-1
- Für die Positionen j=1,2,3,...,M-1 folgt

$$\text{fail}(j) = \begin{cases} \max\{k_l,\ l=1,2,...,j-1\} \text{ mit } k_l<j \\ \text{und } p[0]p[1]...p[k_l] = p[j-k_l]p[j-k_l+1]...p[j-1]p[j] \\ \end{cases}$$

- Beispiele

a	b	a	a	a	a
-1	-1	0	-1	0	1
	p[0]=p[2]	0	p[0] p[1]=p[1] p[2]		
			p[0]=p[2]		

Zur Implementierung verändern wir **fail** ein wenig, übergeben wird nur das Muster, die **Wiederaufsetzindizes** werden in einer globalen *Tabelle* failure[0...M-1] abgespeichert.

Der entscheidende Trick:

Die maximal mögliche Teilkette bei j wird zunächst durch die bekannte im Fall j-1 bestimmt (i := failure [j-1]).

Anschließend werden nur noch Teilketten größerer Länge analysiert.

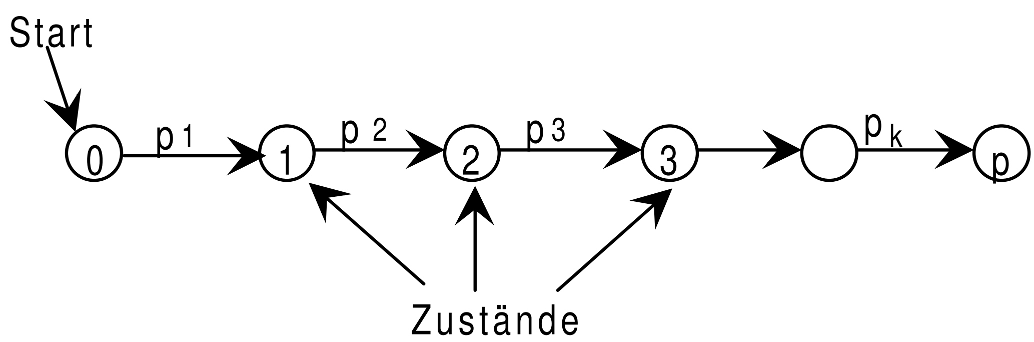
Erklärung von KMP mit Hilfe eines Automaten

A: Text p: Muster

1. Baue aus dem Muster einen Automaten, bestehend aus:
   - k+1 Zuständen, wobei k = |p|
   - einer Übergangsfunktion

2. Bestimme eine Hilfsfunktion h, die ermittelt, wie weit man im Text vorrücken kann, falls eine Nichtübereinstimmung eintritt.

Ad 1) Ist $p = p_1 p_2 ... p_k$ mit $p_k \in \Sigma$ das Muster, so hat der Automat zunächst folgende Gestalt:



Lies Text A in den Automaten ein, d.h. folge den entsprechenden Kanten, solange Übereinstimmung vorliegt. Tritt eine Nichtübereinstimmung auf, ermittle mittels h, in welchen Zustand man zurückgeben muß.

Ad 2)

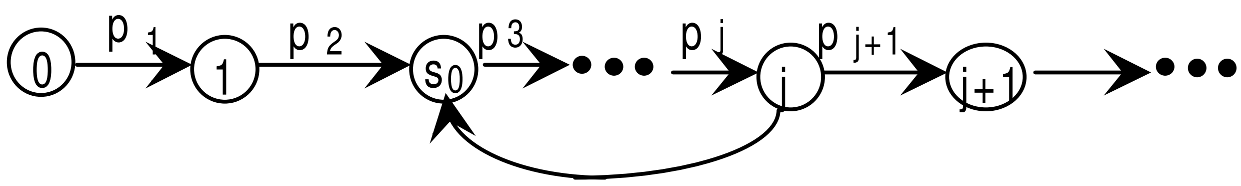
Die Funktion h für ein Muster $p = p_1p_2...p_k$

h: {1 ...k } -> {0, ..., k-1}

$h(j) = \max\{s: s<j$ und $p_1....p_s$ ist Suffix von $p_1...p_j\} = s_0$

Suffix:	1 4 5 8 7 9 1 4 5....	Text
	1 4 5 8 7 9 1 4 6....	Muster
	1 4 5	
		Suffix: 1 4, $s_0 = 2$

Trage die Funktion h in den Automaten ein (als „Rückwärtspfeile“):



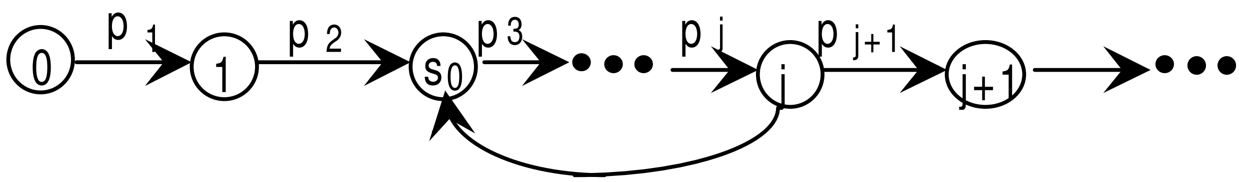
von j nach s_0

Verwendung des Automaten

Sei A: $a_1 \ldots\ldots\ldots a_h \mid a_{h+1}$ Text

$p_1 \ldots\ldots p_j \mid p_{j+1}$ Muster, $a_{h+1} \neq p_{j+1}$

d.h. der Anfang $p_1 \ldots p_j$ des Musters ist gefunden, aber $a_{h+1} \neq p_{j+1}$.
Dann ist der Automat jetzt im Zustand j.



von j nach s_0

Jetzt wird h wiederholt angewandt, d.h. es wird

$h(j)$, $h(h(j))$ usw. berechnet,

bis der erste Wert g gefunden wird mit

$h^g(j) = u$ und $a_{h+1} = p_{u+1}$

oder

$h^g(j) = 0$ und $a_{h+1} \neq p_1$

Danach:

a_{h+2} wird gelesen usw.

Wird auf diese Weise irgendwann der Zustand k erreicht, so ist ein Vorkommen des Musters in A gefunden.

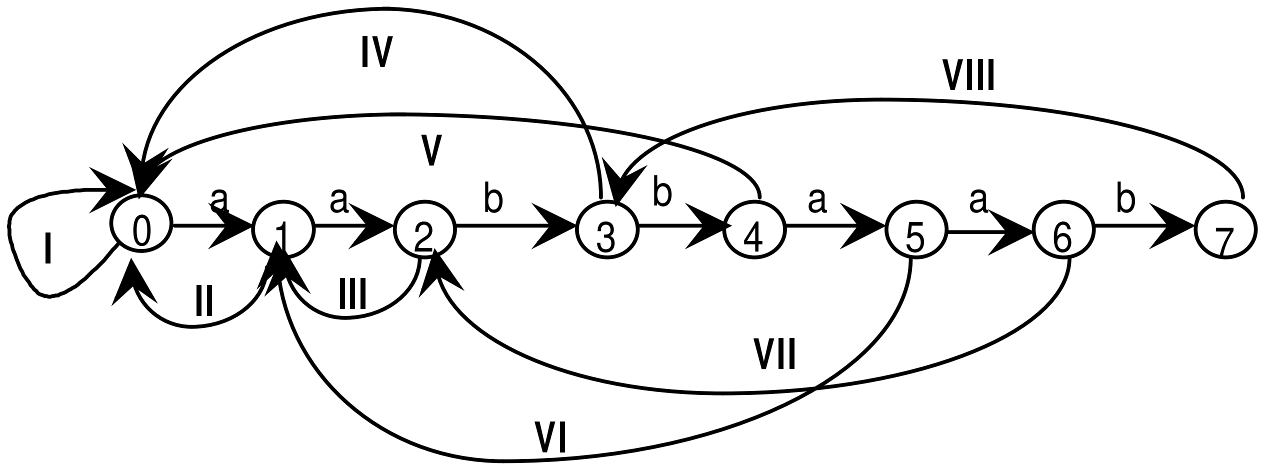
Beispiel

$\Sigma = \{a,b\}$

A = abaabaabbaab

p = aabbaab

Aus dem Muster erstellt man als erstes einen Automaten:



I. Falls der erste Buchstabe kein a ist, muß ein neuer erste Buchstabe eingelesen werden.

II. Nun ist der erste Buchstabe ein a. Ist der zweite Buchstabe ein b, so entspricht das Muster nicht dem Text, so daß zum Zustand 0 gesprungen wird. Hier wird festgestellt, daß nun b der erste Buchstabe ist, so daß ein neuer Buchstabe (erster) eingelesen werden muß.

III. Die ersten beiden Buchstaben sind a's. Folgt nun wieder ein a, so springt man zu Zustand 1, und das a wird eingelesen.

IV. Ist nun der vierte Buchstabe wieder ein a, so muß zu Zustand 0 gesprungen werden, da ba als Folge vorher nicht vorkam im Muster

V. klar

VI. Folgt nun kein a, so geht man zu Zustand 1über, da ein a bereits gelesen ist und dies dem ersten Buchstaben des Musters entspricht. Von dort wird sicherlich auf Zustand 0 gesprungen, da nach Zustand 1 ebenfalls ein a folgen muß.

VII. Wenn nach Zustand 6 ein a folgt, so wird zu Zustand 2 gesprungen, da die zwei a's vor dem Zustand 6 den ersten beiden a's des Musters entsprechen. Wir gehen deshalb in den Zustand „zwei a's stimmen überein", also den Zustand 2.

VIII. Hier springt man zu Zustand 3, da die letzten Buchstaben aab den ersten Buchstaben bis Zustand 3 entsprechen und so weiter...

⇨ h(1) = 0 h(2) = 1 h(3) = 0 h(4) = 0
h(5) = 1 h(6) =2 h(7) = 3

Algorithmus Hilfsfunktion für $p = p_1 \ldots p_k$

```
   begin
1)   h(1) := 0;
2)   for j:= 2 TO k DO BEGIN
3)      i := h(j-1);
4)      WHILE p_j ≠ p_{i+1} AND i>0 DO
          i := h(i);
        IF p_j ≠ p_{i+1} AND i=0 THEN h(j) :=0
5)      ELSE h(j) := i+1;
     END
   END;
```

1) Falls der erste Buchstabe aus dem Text nicht dem ersten Buchstaben aus dem Muster entspricht, so springt der Automat zum Zustand 0.
2) Nun wird jedem Zustand ein Rücksprungzustand zugewiesen, zu dem gesprungen wird, falls der eingelesene Buchstabe nicht dem Buchstaben im Muster entspricht.
3) i ist der Rücksprungzustsand vom letzten Zustand
4) Ist nun unser aktueller Musterbuchstabe ungleich dem Buchstaben nach dem letzten Rücksprung, so darf dort nicht hingesprungen werden, und der Rücksprungzustand vom Rücksprungzustand wird in i eingelesen.
5) Ist der aktuelle Buchstabe gleich dem Buchstaben nach dem letzten Rücksprung, so ist der Rücksprung vom aktuellen Zustand gleich dem Rücksprung +1 vom letzten Zustand.

Es bleibt festzuhalten, daß immer von dem zuletzt zurückgesprungenen Zustand ausgegangen wird, um den neuen Rücksprungzustand zu bestimmen.

Kosten für die Berechnung der Hilfsfunktion

Zeile 3, 5, 6 : konstante Kosten

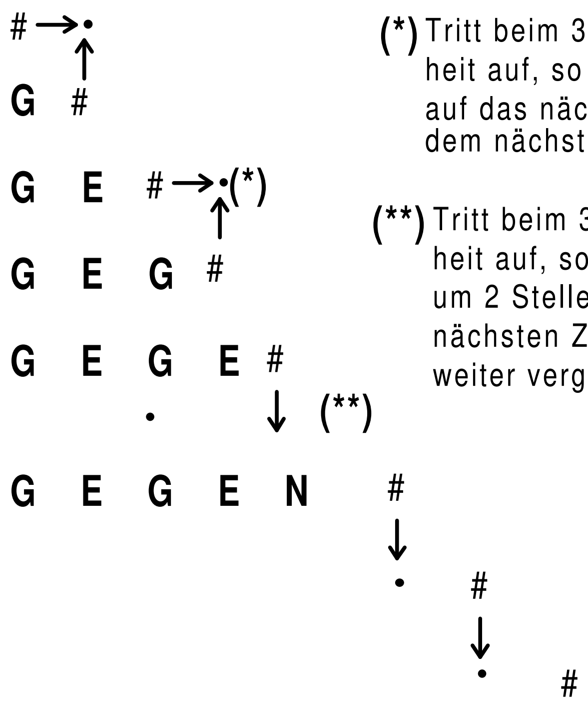
while-statement wird über alle j=2...k höchstens k-mal ausgeführt
⇨ O(2k) = O(k)

Man beachte, daß die Berechnung der Hilfsfunktion und somit auch des gesamten Automaten ausschließlich auf der Basis des Musters p erfolgt, unabhängig vom Text.

1) KMP, Versuch einer Veranschaulichung

G	E	G	E	N	S	A	T	Z
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

Pattern



(*) Tritt beim 3. Zeichen des Patterns eine Ungleichheit auf, so wird die Startposition des Patterns auf das nächste Zeichen gesetzt (*) und mit dem nächsten Zeichen verglichen (→).

(**) Tritt beim 3. Zeichen des Patterns eine Ungleichheit auf, so wird die Startposition des Patterns um 2 Stellen Nach rechts verschoben (• in der nächsten Zeile) und mit dem aktuellen Zeichen weiter verglichen(↓).

-1	0	-1	0	2	0	0	0	0
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

Verschiebungstabelle
(Gleich [])

\# Ungleichheit zwischen Text und Patter

• Neue Startposition des Patternanfangs bei Ungleichheit

→ Fortsetzung mit nächstem Textzeichen
Fortsetzung mit aktuellem Textzeichen
↓↑ Die Pfeile zeigen auf die für den jeweiligen Fall geltende neue Startposition des Vergleichens

KMP-Suche, Programm

```
#define max_string_size 100
#define max_pattern_size 100;
int kmpsearch( );
void fail( );
int failure [max_string_size];
char a [max_pattern_size];
char p [max_pattern_size];
```

```
int kmpsearch(char *a, char *p)
{
/*Knuth,Morris,Pratt Mustererkennungsalgorithmus*/
        int i = 0, j = 0;
        int lens = strlen(a);
        lenp = strlen(p);
        while (i<lens&&j<lenp) {
                if (a[i] == p[j]) {
                        i++; j++; }
                else if (j == 0) i++;
                        else j == failure[j-1]+1;
        }
        return ( )j == lenp) ? (i-lenp) : -1;
}
```

Bemerkungen:

- i wird nie verkleinert, d.h. es kann maximal N-mal ausgeführt werden
- j kann kleiner werden (j=failure(j-1)+1), aber maximal so oft j++ angestoßen wird; j++ ist aber mit i++ gekoppelt, so daß j auch nicht häufiger als N-mal inkrementiert wird
- aus den beiden obigen Punkten folgt das Zeitverhalten 0(N)

Programmanweisungen zur Realisierung von fail

```
void fail(char *p)
{
/* Berechnung der Musterfehlschlag-Funktion */
       int m = strlen(p);
       failure[0] = -1;
       for (j=1; j<m; j++) {
              i = failure[j-1];
              while ((P[j] != p[i+1]) && (i>= 0))
                     i = failure[i];
              if (p[j] == p[i+1])
                     failure[j] = i+1;
              else failure[j] = -1;
```

Bemerkungen:

- die for-Schleife wird maximal M-1 durchlaufen, dies gilt auch für die while-Schleife
- die Laufzeit für dieses Programmstück ist also un der Ordnung 0(M)

Zusammenfassung für das KMP-Verfahren

- nicht sehr leicht zugänglich
- erfordert eine aufwendige Vorausarbeitung des Musters vor der eigentlichen Suche
- Laufzeitverhalten 0(N+M), wobei N die Text- und M die Musterlänge;- und zwar unabhängig von zugrundeliegendem Alphabet

KMP-Suche, Komplexität

+ Aufwand KMP in extremen Fällen deutlich *geringer* als naives Verfahren. Anzahl Vergleiche höchstens 2 N-M.
  Begründung: Zwischen zwei Vergleichen wird Pattern geschoben (höchstens N-M mal) oder Vergleichsposition wandert nach rechts (höchstens N-1 mal) => nicht mehr als N-M+N-1+1 = 2 N-M Vergleiche.
+ Auch im average case KMP besser als naiv!
+ Textzeiger wird während des Suchens nicht zurückgesetzt => Suche in sequentiellen *Dateien*.
- Aufbau (Komplexität: O(s)) und die Speicherung der Tabelle.

6.4 Das Suchverfahren von Boyer-Moore

Eine andere Verbesserung des naiven Algorithmus wurde unabhängig von Knuth-Morris-Pratt von Boyer und Moore vorgeschlagen.

Grundidee

Das Muster p[0:M-1] wird **konventionell von links nach rechts** über den Text i[0:N-1] bewegt, aber das Auffinden der Übereinstimmung läuft von **rechts nach links** tabellengesteuert durch das Muster.

Vorteil

In vielen Fällen kann man schnell erkennen, daß nur eine ganze Musterlänge verschoben werden kann.

Beispiel

Text a[0..49] = „er sagte abrakadabra..............“

Muster p[0..3] = „aber“

```
er sagte abrakadabra, es bewegte sich aber nichts
aberaber
     aberaber
            aberaber
                   aber
                    aberaberaber
                         aberaberaber
                                  aber
```

Kommentar

Im ersten und vierten Vergleich sieht man das Herüberziehen des ganzen Musters sehr deutlich, da der jeweilige Mismatch-Buchstabe im Text a[3]=’s’ bzw. a[15]=’d’ im Muster p gar nicht vorkommt.

Anzahl der Vergleiche:

- naives Verfahren ~ 40*4 = 160
- Knuth-Morris-Pratt (KMP) ~ 40+4 = 44
- Boyer-Moore (BM) ~ 17

Boyer-Moore, Beispiel 2

Die Verschiebung des Musters ist maximal, wenn das Mismatch-Zeichen im Muster nirgends vorkommt.

```
A STRING SEARCHING EXAMPLE CONSISTING OF ...
STING
        STING
               STING
                         STING
                                 STING
                                         STING
                                                 STING
A STRING SEARCHING EXAMPLE CONSISTING OF ...
```

Das Beispiel ist charakteristisch für die Situation

- umfangreiches Alphabet
- kurzes Muster mit wenig Zeichen aus dem Text,

für die der Algorithmus von Boyer-Moore gut arbeitet.

Im Prinzip

Musterverschiebung wie bei KMP, nur die Berechnung der Teilmuster und damit der Wiederaufsetz-Positionen ist etwas anders.

Die **Wiederaufsetz-Position** in Muster und Text hängt auch vom Mismatch-Zeichen μ = a[i] ab. In diesem Zusammenhang wird der Begriff der Verkommensheuristik entwickelt; das ist nichts anderes als eine Tabelle skip (μ,j), welche in Abhängigkeit vom Alphabet die jeweilige Sprungweite im Text a[0..N-1] festlegt.

```
Er sagte ab...
aberaber
     aber
   ↓   ↓
 μ=s   ↓
skip=4 ↓
       μ=e
   skip=1
```

Für μ=s und Mismatch in Position 4: skip = 4
Für μ=e und Mismatch in Positon 4: skip = 1

Die Tabelle skip(μ,j) in Abhängigkeit vom Zeichen μ im Text und der Mismatch-Position j läßt sich wie folgt definieren:

$$\text{skip}(\mu,j) = \begin{cases} \text{M, wenn } \mu \notin p[0:m-1] \\ \text{M-j (kürzeste Entfernung zum Musterende), sonst} \end{cases}$$

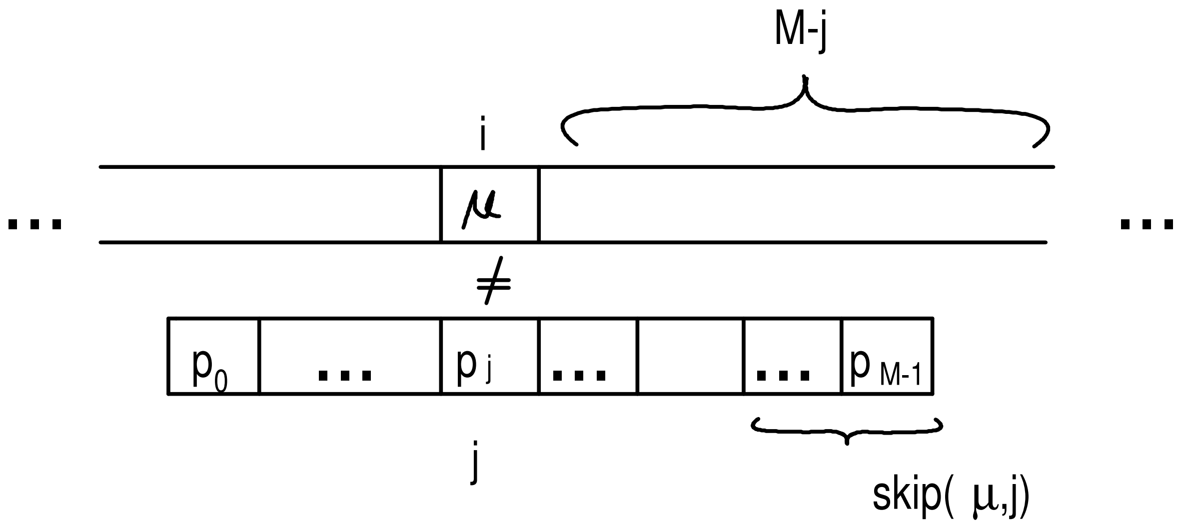
Beispiel	p[0]	p[1]	p[2]	p[3]
	a	b	e	r
	3	2	1	0

Tabelle für das Alphabet

		a	b	c	d
e	f....	r	s....	z	
		3	2	4	4
1	4...	0	4....	4	

Aus der Tabelle skip(μ,j) berechnen wir die relevante Verschiebung des Textzeigers, dabei müssen zwei Fälle unterschieden werden:

Fall 1)

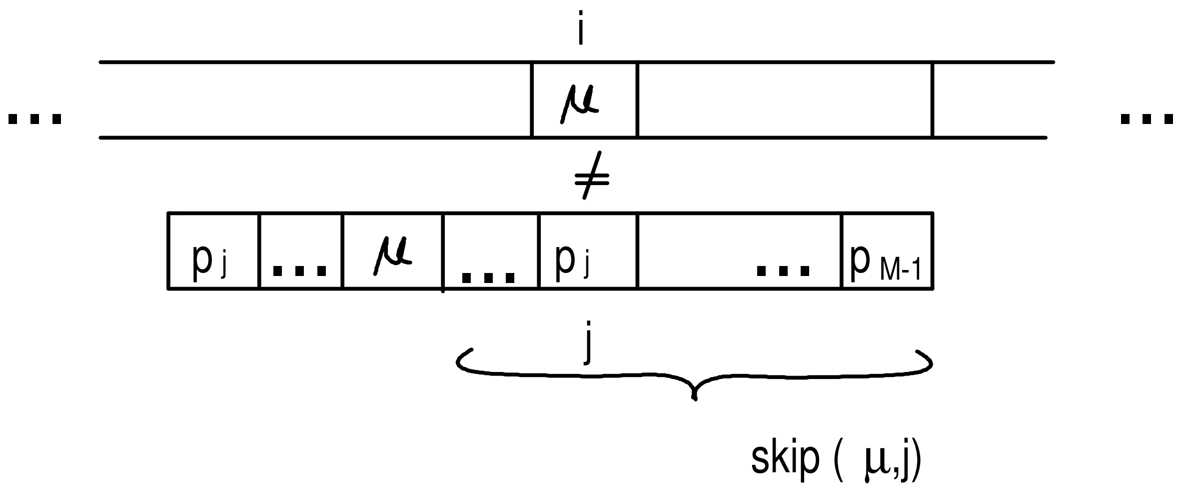


Das Mismatch-Zeichen befindet sich nur rechts in der Mismatch-Position j

⇨ neuer Textzeiger   i := i+M-j+1

⇨ neuer Musterzeiger j := M-1

Fall 2)



Das Mismatch-Zeichen μ befindet sich nur **links** in der Mismatch-Position j

⇨ neuer Textzeiger i := i+skip(μ,j)

⇨ neuer Musterzeiger j := M-1

Boyer-Moore, Programm

Eine mögliche Implementierung muß sich an folgenden Schritten orientieren:

- man benötigt ein Feld **skip**, welches für jedes Zeichen im Alphabet angibt, wie weit bei einem Mismatch zu springen ist
- i := index(*buchstabe*) gibt für jeden *buchstaben* seine Position i im Alphabet wieder
- *initskip*(p) besetzt alle diejenigen Positionen von **skip** mit M, deren Buchstabe nicht im Muster enthalten ist

initskip(p) besetzt die verbleibenden Position von skip wie folgt:

Für j = 0 bis M-1: skip[index[p[j]]] := M-j-1

Programm

```
int mischarsearch(char *p, char *a)
       {
              int i, j, t, M = strlen(p), N =
strlen(a);                    initskip(p);
              for (i = M-1, j = M-1; j>0; i--, j--
)
                     while (a[i] != p[j])
                            {
                                   t =
skip[index(a[i])];
                                   i += (M-j>t) ?
M-j : t;
                                   if (i>=N)
return N;
                                   j = M-1;
                            }
              return i;
}
```

Boyer-Moore, Komplexität

Größe des Alphabets: A

Bei der Suche gemäß dem Algorithmus von Boyer-Moore sind niemals mehr als N Zeichenvergleiche erforderlich plus A Operationen zum Aufbau der Tabelle, und es werden ungefähr N/M Schritte benötigt, wenn das Alphabet nicht klein und das Muster nicht lang ist.

In praktischen Fällen: Aufwand oft $\approx$ N/M << N+A

Im Beispiel für „aber“

Text/Muster = 50/4 = 12,5
tatsächlich: 17 Vergleiche

Zusammenfassung für das Boyer-Moore-Verfahren

- Die Verarbeitung des Musters (die skip-Tabelle), ist abhängig vom Alphabet - insbesondere bei binären Textdateien und entsprechenden Mustern ergeben sich kaum Vorteile.
- In der einfachen Form mit einer Sprungtabelle ergibt sich ein effizienter und kompakter Code.
- Ein praktisches Laufzeitverhalten 0(N/M) bei M<<N und hinreichend großen Alphabeten ist erreichbar.

6.5 Das Suchverfahren von Rabin und Karp

Alle bisher diskutierten Ansätze für die Mustersuche in Texten sind gegenüber dem „klassischen“ Algorithmen, die auf strukturierten Datensätzen operieren, deutlich anders aufgebaut. Unsere Begründung dafür lag hauptsächlich bei der „*Nicht-Struktur*“ der Texte.

Das Verfahren von Rabin und Karp greift dagegen eine Suchtechnik aus den klassischen Algorithmen auf, nämlich Hashing, allerdings nicht zur Lokalisierung von Speicherorten, sondern zum effizienten Erkennen von Match-Kandidaten.

Idee von Rabin-Karp

```
Muster                    p[0] p[1].......p[M−1]
       ↓
Schlüsselzahl                                       k
       ↓
Hashfunktion  h(k) = k mod q; q Primzahl
       ↓
Vergleich                 h(k) = h(k'ᵢ)
       ↑
Hashfunktion  h(k'ᵢ) = k'ᵢ mod q
       ↑
Schlüsselzahl                                       k'ᵢ
       ↑
Teiltext                         a[i] a[i+1]
.....a[i+M−1]
       ↑
Text                             a[0] a[1] .......a[i]
.......a[M−1]
```

Ist $h(k) = h(k'_i)$, so ist der gerade betrachtete Teiltext **möglicherweise** mit dem Muster identisch.

Der Weg vom Muster zur Hash-Funktion muß nur **einmal** durchlaufen werden. **Beim Text** ist die Angelegenheit **aufwendiger**, hier muß beim Weiterschalten des Textzeigers i := i+1 eventuell jedesmal die entsprechende Hash-Funktion für den Teiltext neu berechnet werden.

⇨ hoher Aufwand

Aber: Trick von Rabin-Karp:

1. Das Teilstück wird in eine Zahl verwandelt, dazu benötigt man eine **Basis** -> Zeichenumfang d des Textalphabets, also Basis d = $|\Sigma|$

2. Eigentliche Transformation durch die Zuordnung

a[i]	a[i+1]	....	a[i+M-2]	a[i+M-1]
$\Downarrow$	$\Downarrow$		$\Downarrow$	$\Downarrow$
d^{M-1}	d^{M-2}		d^1	d^0

3. Argument für die Hash-Funktion ist damit
   $k_2 = a[i]d^{M-1} + a[i+1]d^{M-2} + ...$
   $a[i+M-2]d^1 + a[i+M-1]d^0$

4. Was bedeutet dann das Weiterschalten des Textzeigers i := i+1?
   $k'_{i+1} = a[i+1]d^{M-1} + a[i+2]d^{M-2}$
   $a[i+M-1]d^{-1} + a[i+M]d^0$

5. Kann man k'_{i+1} aus k'_i berechnen?

   Multiplikation von k'_i mit der Basis d

   $k'_i * d = a[i]d^M + a[i+1]d^{M-1} + ...$

   $a[i+M-2]d^2 + a[i+M-1]d^1$

6. Wir bilden die Differenz

   $k'_{i+1} - k'_i *d = a[i+1]d^{M-1} + a[i+2]d^{M-2} + ...$

   $+ a[i+M-1]d^1 +$

   $a[i+M]d^0$

   $-a[i]d^M - a[i+1]d^{M-1} - a[i+2]d^{M-2}$

   $-a[i+M-1]d^1$

   und fassen zusammen

   $k'_{i+1} - k'_i *d = - a[i]d^M + a[i+M]$

7. Aus Punkt (6) folgt

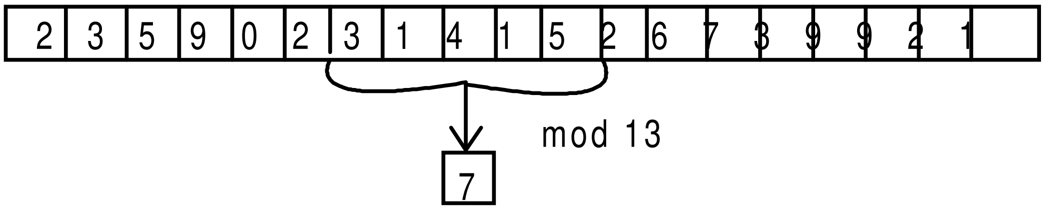
   $k'_{i+1} = k'_i *d - a[i]d^M + a[i+M]$

   $k'_{i+1} = (k'_i - a[i]d^{M-1})*d^1 + a[i+M]d^0$

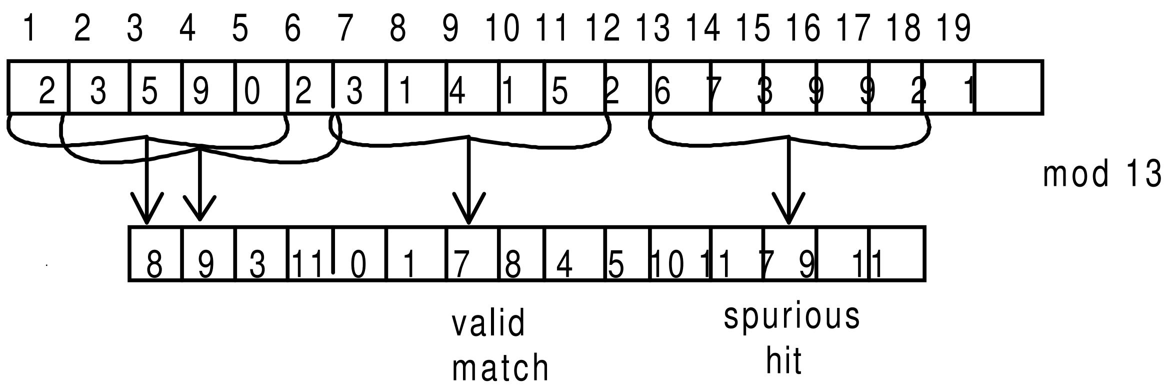
Zusammenfassung 1-7:

Die Hash-Funktion muß nur einmal für das Muster h(k) und nur einmal für die Teiltextkette a[0] a[1] ..a[M-1] berechnet werden, die Weiterschaltung erfolgt nach (7) sehr effizient. Der Aufwand zur Berechnung von k'_{i+1} aus k'_i ist O(1).

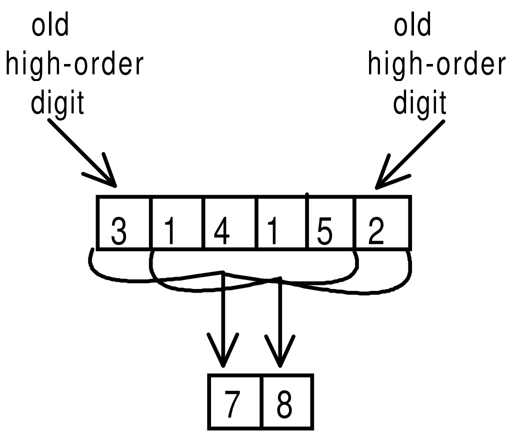
Algorithmus von Rabin-Karp, Beispiel



(a)



(b)



old high-order digit   shift   old high-order digit

$$
\begin{aligned}
14152 &\equiv (31415-3*10000)*10+2 \pmod{13} \\
&\equiv (7-3*3)*10+2 \pmod{13} \\
&\equiv 8 \pmod{13}
\end{aligned}
$$

(c)

Rabin-Karp, Programm

```
#define q 33554393
#define d 32
int rksearch(char *p, char *a)
        {
                int i, dM = 1, h1 = 0, h2 = 0;
                int M = strlen(p), N = strlen(a);
                for (i = 1; i<M; i++) dM = (d*dM)
%q;
                for (i = 0; i<M; i++)
                        {
                                h1 =
(h1*d+index(p[i])) %q;
                                h2 =
(h2*d+index(a[i])) %q;
                        }
                for (i = 0; h1 != h2; i++)
                        {
                                h2 = (h2+d*q-
index(a[i])*dM) %q;
                                h2 =
(h2*d+index(a[i+M])) %q;
                                if (i>N-M) return N;
                        }
                return i;
        }
```

Kommentare zum Programm

- `h1` ist der Hash-Wert h(k) für das Muster
- `h2` ist der Hash-Wert $h(k'_0)$ für den Textanfang
- in der letzten **for-Schleife** wird der Vergleich zwischen `h1` und `h2` gestartet und linear durch den Text gelaufen; `h2` ist der jeweilige aktuelle Hash-Wert $h(k'_{i+1})$
- `index(`*`buchstabe`*`)` gibt die Position i des *Buchstabens* im Alphabet an. Die Funkton index hat ein Zeichen (char) als Argument und gibt i für den i-ten Buchstaben des Alphabets zurück, für ein Leerzeichen 0.
- Die Primzahl `q` ist so gewählt, daß mit `d*q` kein Überlauf vorkommen kann
- Das Programm findet nur zwei übereinstimmende Hash-Werte an einer festen Textzeigerposition. Es muß jetzt noch einmal **zeichenweise** verglichen werden, ob nicht nur die Hash-Werte, sondern tatsächlich auch die Zeichen übereinstimmen.

Zusammenfassung des Rabin-Karp-Verfahrens

- Transparenter und eleganter Algorithmus ohne Bildung komplizierter Submuster
- Keine aufwendige Vorverarbeitung des Musters
- Der Text wird linear durchgemustert, deshalb ergibt sich Laufzeitverhalten O(N+M)