11 Suche in Zeichenketten

11.1 Problembeschreibung

Gegeben sind zwei Zeichenketten s (Text/String der Länge n) und p (Muster/Pattern der Länge m, wobei m < n)

```
s: array [1:n] of char
p: array [1:m] of char
```

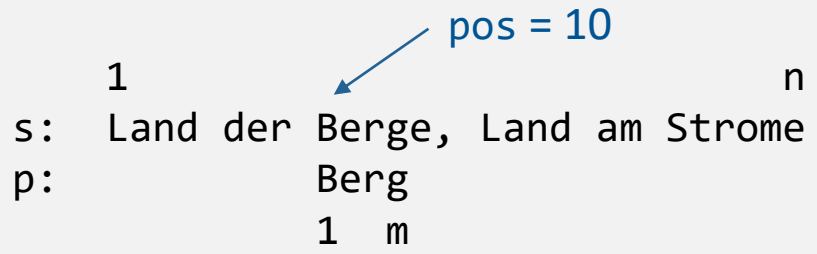
Gesucht ist die Position pos des ersten Auftretens von p in s, sodass

- `s[pos:pos + m - 1] = p[1:m]` wenn $p \in s$
- `pos = 0` wenn $p \notin s$

Schnittstelle

```
Search(↓s: array[1:n] of char ↓p: array[1:m] of char ↑pos: int)
```

Beispiel



Geschichte

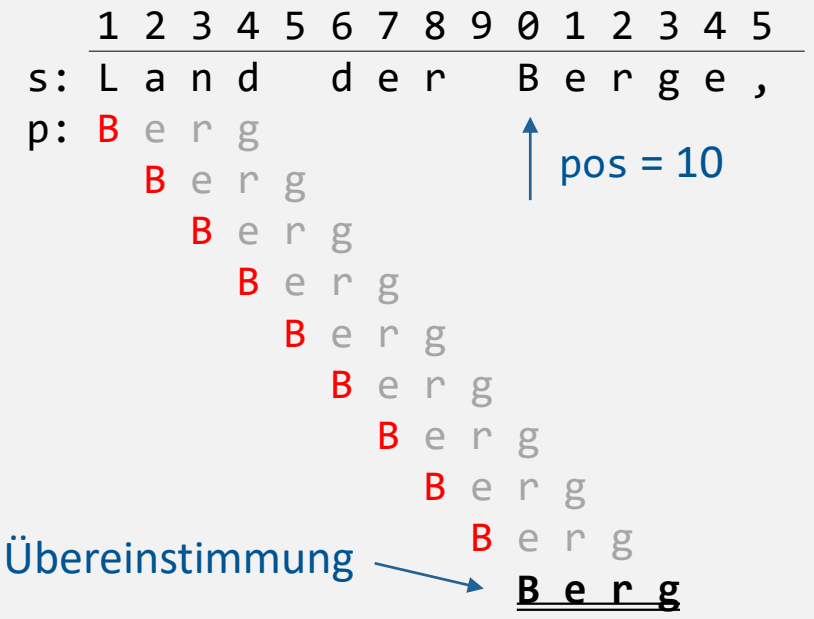
- Offensichtliche Lösung benötigt im ungünstigsten Fall $m \cdot n$ Vergleiche
- A. Cook hat bewiesen (1970), dass ein Algorithmus existieren muss, der mit $m + n$ Vergleichen auskommt (Problemkomplexität $O(m + n)$)
- D.E. Knuth und V.R. Pratt gelang es, einen solchen Algorithmus zu finden
- J.H. Morris hatte zur selben Zeit die gleiche Lösungsidee
- Knuth, Morris und Pratt veröffentlichten 1977 ihren Algorithmus
- R.S. Boyer und J.S. Moore veröffentlichten 1977 anderen Algorithmus, der ebenfalls mit $m + n$ Vergleichen auskommt
- R.M. Karp und M.O. Rabin schlugen 1980 einen auf der Idee des Hashings basierenden Algorithmus vor

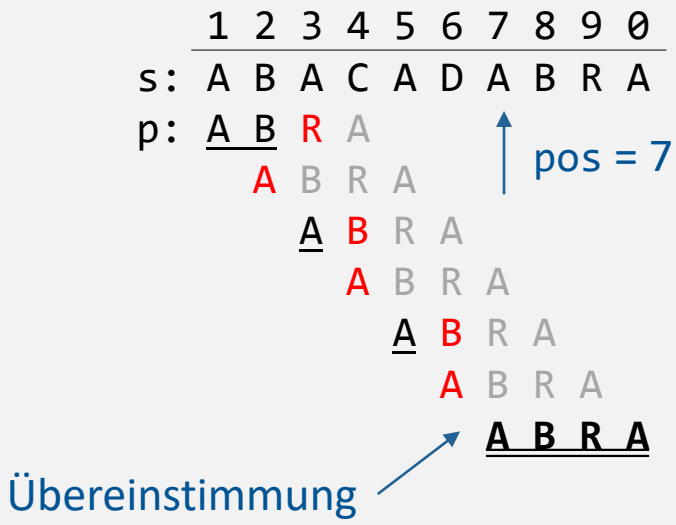
11.2 Ein einfacher Algorithmus `BruteForceMatching`

Lösungsidee

- Muster p solange über jede Position in Text s legen,
  - bis eine Übereinstimmung der Länge m gefunden ist,
  - bei einer Nichtübereinstimmung (s[i] ≠ p[j] mit j ≤ m, *mismatch*) wird p gegenüber s um eine Position nach rechts verschoben
- Sedgewick bezeichnet diese Lösung als *brute force* („rohe Gewalt“)

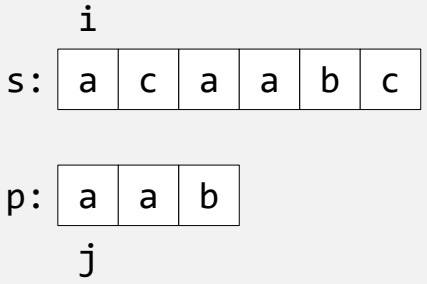
Beispiele



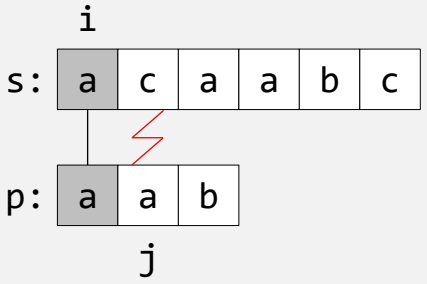


`BruteForceMatching` – Variante 1

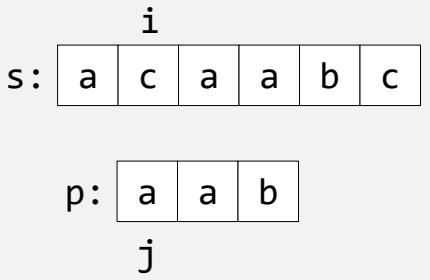
Muster wird von links nach rechts über Text gelegt



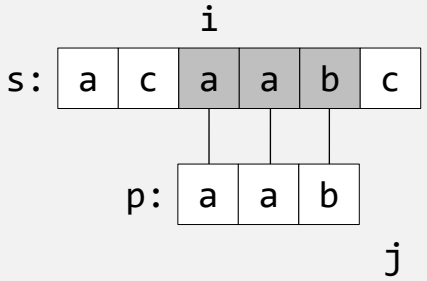
Beginn mit i = 1 und j = 1



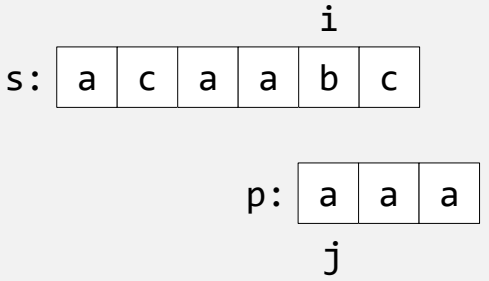
Bei Übereinstimmung wird j um 1 erhöht



Bei Nichtübereinstimmung wird Muster um eine Stelle nach rechts verschoben (i := i + 1, j := 1)



Muster in Text an Position i enthalten, wenn j > m



Muster in Text nicht enthalten, wenn i > (n – m + 1)

BruteForceMatching – Variante 1

Algorithmus

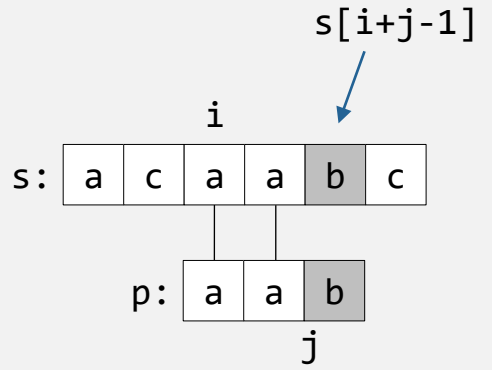
```
BruteForceMatching(↓s: array[1:n] of char
                   ↓p: array[1:m] of char
                   ↑pos: int)
  var i, j: int
begin
  i := 1
  pos := 0
  while (pos = 0) and (i ≤ n – m + 1) do
    j := 1
    while (j ≤ m) and (s[i + j - 1] = p[j]) do
      j := j + 1
    end -- while
    if j > m then
      pos := i
    else
      i := i + 1
    end -- if
  end -- while
end BruteForceMatching
```

solange nicht gefunden

solange Ende nicht erreicht

hier gilt: `s[i:i + j - 1] = p[1:j]`

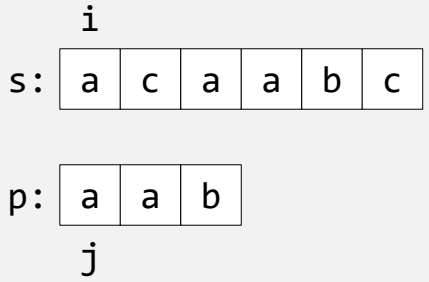
Muster p an Position i gefunden



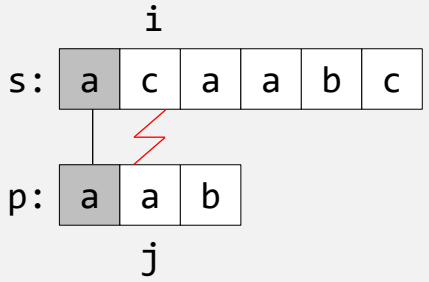
`BruteForceMatching` – Variante 2

In Variante 1 sind bei jedem Vergleich Indexberechnungen erforderlich

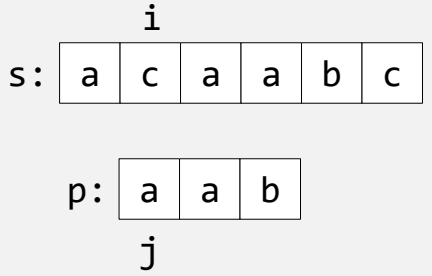
Überlegungen zu Variante 2 (…`s[i] = p[j]`…):



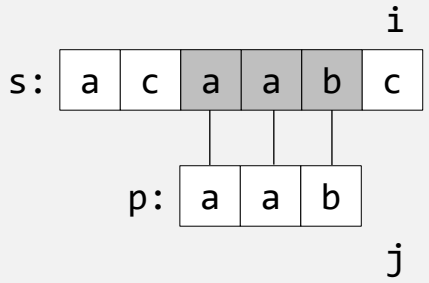
Beginn mit i = 1 und j = 1



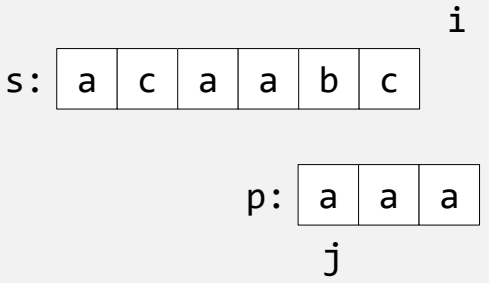
Bei Übereinstimmung werden i und j um 1 erhöht



Bei Nichtübereinstimmung wird Muster um eine Stelle nach rechts verschoben (i := i - j + 2, j := 1)



Muster in Text an Position i - m enthalten, wenn j > m



Muster in Text nicht enthalten, wenn i > n

BruteForceMatching – Variante 2

Algorithmus

```
BruteForceMatching2(↓s: array[1:n] of char
                    ↓p: array[1:m] of char
                    ↑pos: int)
  var i, j: int
begin
  i := 1
  j := 1
  repeat
    if s[i] = p[j] then
      i := i + 1; j := j + 1
    else
      i := i – j + 2; j := 1
    end -- if
  until (i > n) or (j > m)
  if j > m then
    pos := i - m
  else
    pos := 0
  end -- if
end BruteForceMatching2
```

hier gilt: `s[i-j+1:i] = p[1:j]`

Muster um 1 Stelle nach rechts verschieben (bezogen auf Position mit j = 1)

Muster p an Position i - m gefunden

Laufzeitkomplexität

Günstigster Fall bei erfolgreicher Suche: $O(m)$

- Muster wird bereits an erster Position in Text gefunden

Günstigster Fall bei erfolgloser Suche: $O(n)$

- Erstes Zeichen von Muster kommt in Text nicht vor

Ungünstigster Fall: $O(m \cdot n)$

- Muster der Länge m muss mit jeder Teilkette aus dem Text verglichen werden $\big(m \cdot (n - m + 1)\big)$ Vergleiche)

```
s := "aaaaaaaaaaaaaaaaaaaaaab"
p := "aaaab"
```

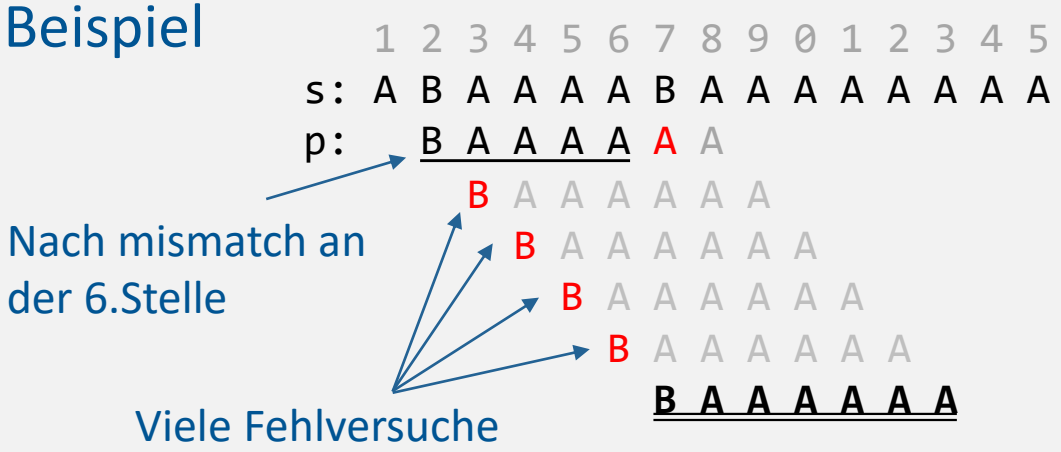
Bei großem Alphabet wird Muster nur selten vollständig mit Teilkette verglichen, Wahrscheinlichkeit $\left(\frac{1}{d}\right)^m$ mit d verschiedenen Zeichen

11.3 Knuth-Morris-Pratt Mustersuchalgorithmus

„this is one of the coolest algorithm that we'll cover in this course“ R. Sedgewick

- Beruht auf der Feststellung, dass wertvolle Zeit verloren geht, wenn bei jeder Verschiebung des Musters (relativ zum Text) stets wieder mit dem ersten Zeichen des Musters begonnen wird
- Wenn an Position j im Muster eine Nichtübereinstimmung festgestellt wird, kennt man die Zeichen p[1], … p[j]
- Aus diesem Wissen lässt sich die Information gewinnen, ob das Muster um mehr als eine Position nach rechts verschoben werden kann

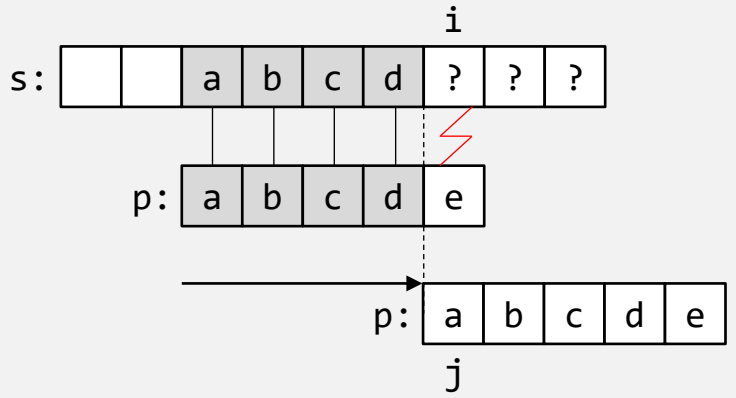
Beispiel



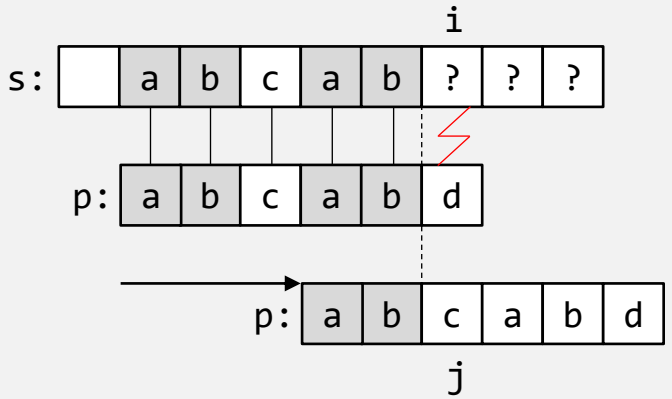
Idee: Wenn erstes Zeichen (hier B) im Rest des Musters nicht mehr vorkommt, können Zeichen AAAA keine gültigen Startpositionen sein

Knuth-Morris-Pratt Mustersuchalgorithmus

Lösungsidee für das Verschieben des Musters um mehr als eine Stelle



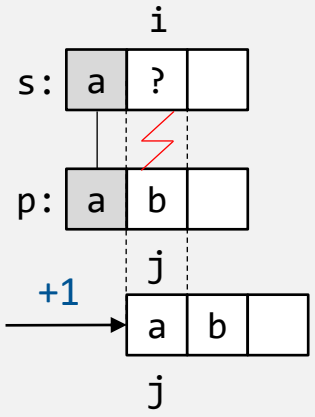
Fall 1: Erstes Zeichen in Muster kommt vor mismatch-Stelle nicht vor.



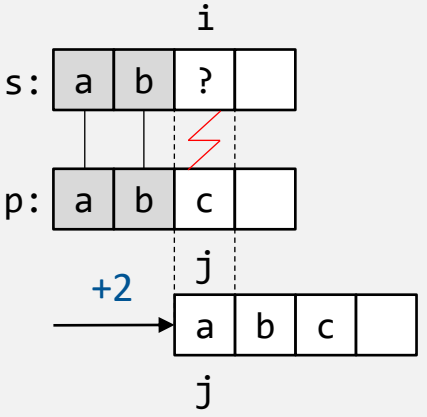
Fall 2: Teilkette vor mismatch-Stelle ist auch am Beginn des Muster zu finden.

- Ein ganzes Muster überspringen funktioniert nicht, wenn das Muster am Punkt des „mismatch“ auf sich selbst passt (Fall 2)
- Vor der Suche kann aufgrund des Musters `p` alleine bereits ermittelt werden, wie weit zurück die Suche wieder aufgesetzt werden muss
- Idee: Im Falle eines mismatch gibt Feld `next` Auskunft, wo im Muster die Suche fortgesetzt werden kann (dh nächster Wert für Position `j`)

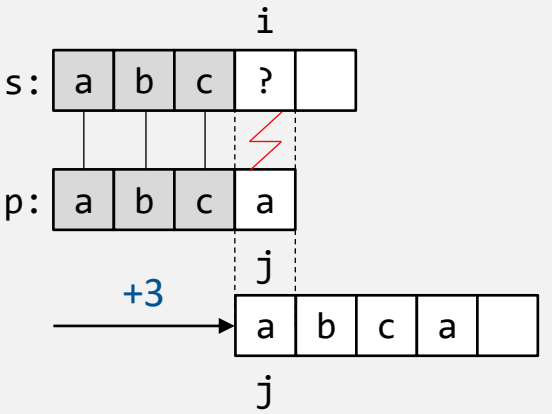
Ermittlung der Elemente des **next**-Felds



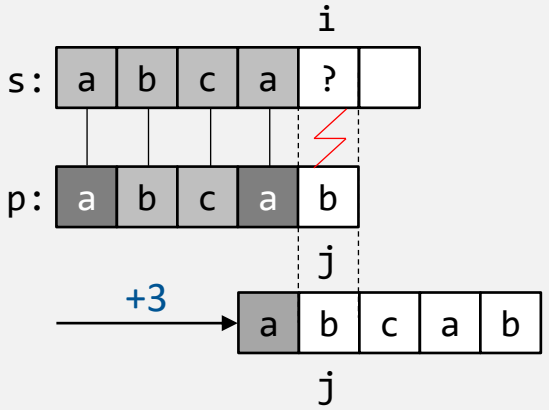
next[2] := 1



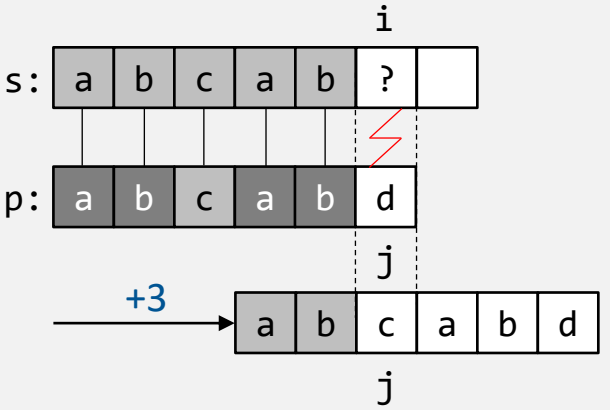
next[3] := 1



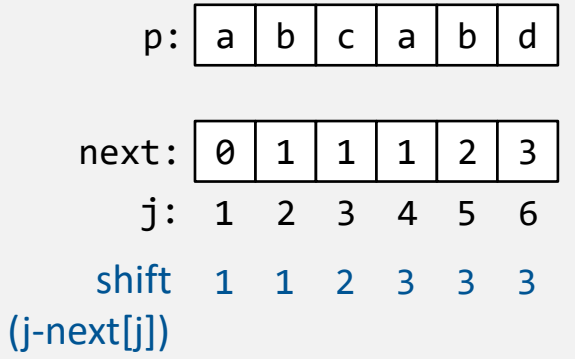
next[4] := 1



next[5] := 2



next[6] := 3

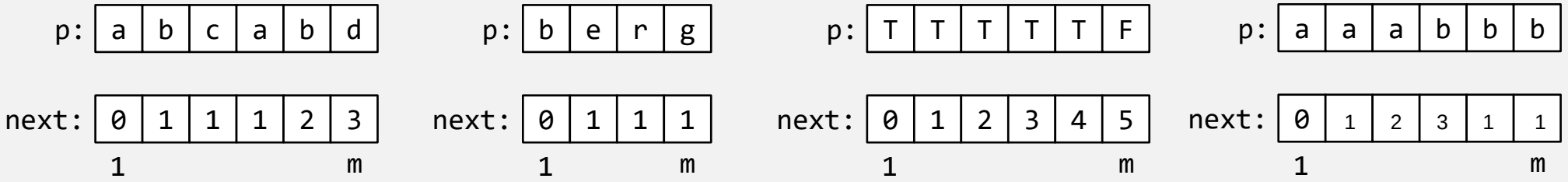


Ermittlung der Elemente des `next`-Felds

Im Falle eines mismatch gibt also `next` Auskunft, wo im Muster die Suche fortgesetzt werden kann

Bei mismatch an der Position j = 1 wird Muster um eine Position verschoben (dh i = i + 1 und j =1, dazu wird `next[1] = 0` gesetzt)

Beispiele



Unter der Annahme, dass wir `next` kennen, erhalten wir somit folgenden Algorithmus (siehe nächste Seite)

Algorithmus

```
KMPMatching(↓s: array[1:n] of char ↓p: array[1:m] of char ↑pos: int)
  var
    next: array[1:m] of int
    i, j: int
begin
  InitNext(↓p ↑next)
  i := 1; j := 1
  repeat
    if (j = 0) or (s[i] = p[j]) then
      i := i + 1
      j := j + 1
    else
      j := next[j]
    end -- if
  until (i > n) or (j > m)
  if j > m then
    pos := i - m
  else
    pos := 0
  end -- if
end KMPMatching
```

Variable i wird nicht verändert!

Muster p im Text s beginnend an Position pos

Muster p nicht im Text s enthalten

Aufbau der Tabelle `next`

Erfolgt nach dem gleichen Prinzip wie `KMPMatching` (weil es sich ja ebenfalls um eine Mustersuche handelt)!

```
InitNext(↓p: array[1:m] of char ↑next: array[1:m] of int)
  var i, j: int
begin
  i := 1;
  j := 0
  next[1] := 0
  repeat
    if (j = 0) or (p[i] = p[j]) then
      i := i + 1
      j := j + 1
      next[i] := j
    else
      j := next[j]
    end -- if
  until i ≥ m
end InitNext
```

Laufzeitkomplexität

- Die Autoren Knuth, Morris und Pratt zeigen, dass ihr Verfahren höchstens $m+n$ Vergleiche benötigt
- Der `KMPSearch` hat somit die asymptotische Laufzeitkomplexität $O(m+n)$, ist also linear
- Die asymptotische Laufzeitkomplexität des Algorithmus `InitNext` für die Ermittlung der Elementwerte des Felds `next` ist $O(m)$

11.4 Rabin-Karp Mustersuchalgorithmus

Zur Effizienzsteigerung (um unnötige Vergleiche zu vermeiden) werden Hashfunktionen herangezogen:

- Berechnung des Hashcodes h_p für das Muster `p` und für jede Teilkette h_s der Länge `m` aus dem Text `s`
- Wenn $h_p \neq h_s$ ist, dann kann an der entsprechenden Stelle in `s` das Muster `p` nicht beginnen, daher wird Hashcode für nächste Teilkette in `s` berechnet (d. h. Muster wird nach rechts verschoben)
- Wenn $h_p = h_s$ werden Muster und Teilkette zeichenweise verglichen (z. B. mittels `BruteForce`)
- Um Zeit zu sparen werden die Hashcodes inkrementell berechnet

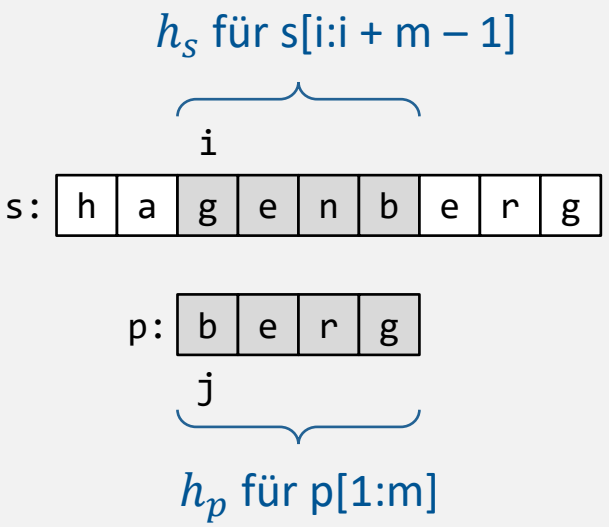
Vorteil:

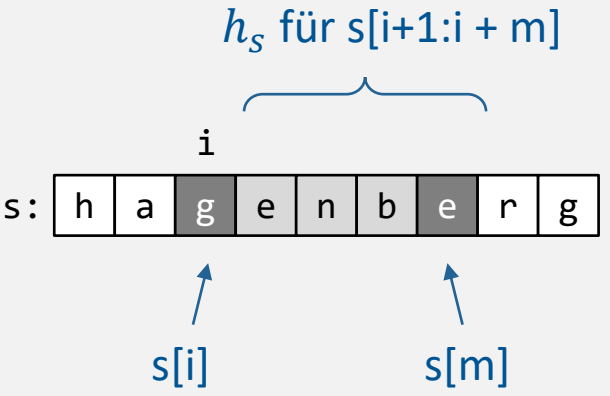
- Durch das frühe Erkennen einer Nichtübereinstimmung, sind weniger Vergleiche erforderlich

Inkrementelle Berechnung der Hashcodes

- Berechnung des Hashcode für Teilkette s[i], ... s[i + m - 1] soll möglich sein, ohne alle Zeichen s[i], ... s[i + m - 1] "besuchen" zu müssen (sonst könnte man auch vergleichen)
- Man berechnet einmal den Hashcode h_s für s[1..m]
- Aus h_s für s[i:i + m - 1] wird h_s für s[i + 1:i + m] berechnet
  - Anteil von s[i] aus Hashcode entfernen und
  - Anteil von s[i + m] hinzufügen

Beispiel





Inkrementelle Berechnung mit einfacher Hashfunktion

Wenn wir die Summe aller Ordinalwerte der einzelnen Zeichen heranziehen, ergibt sich der Hashcode für die Zeichenkette beginnend an Position i

$$h(i) \quad = s[i] + s[i+1] + \cdots + s[i+m-1]$$

und der Hashcode für die Zeichenkette beginnend an Position $i+1$

$$h(i+1) = \quad s[i+1] + \cdots + s[i+m-1] + s[i+m]$$

Daraus folgt:

$$h(i+1) = h(i) \; - s[i] + s[i+m]$$

Beispiel mit `s="Hagenberg"` , `i=3`, `m=4`:

$$h(3) = 103 + 101 + 110 + 98 = 412$$

$$h(3+1) = 101 + 110 + 98 + 101 = 410$$

$$h(3+1) = h(3) - 103 + 101 = 410$$

Hashfunktion von Rabin-Karp

Muster p[1], ... p[m] und Teilketten s[i], … s[i + m - 1] werden als m-stellige Zahlen zur jeweiligen Basis d (Größe des Alphabets) aufgefasst

```
h := 0
for i := 1 to m do
  h := (h * d + Int(↓key[i])) mod q
end -- for
```

Beispiel

- mit Alphabet = $\{A,B,C,D,E,F,G,H,I,J\}$, d = 10
- und Codierung $a \rightarrow 1, b \rightarrow 2, c \rightarrow 3$…
- $ACCD = 1 \cdot 10^3 + 3 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0 = 1334$
- Wenn wir eine Zeichenkette mit gleicher Länge, gleichen Zeichen aber anderer Anordnung haben, erhalten wir ein anderes Ergebnis:
- $ACDC = 1 \cdot 10^3 + 3 \cdot 10^2 + 4 \cdot 10^1 + 3 \cdot 10^0 = 1343$

Inkrementelle Berechnung mit Hashfunk. von Rabin-Karp

Hashfunktion von Rabin-Karp

```
h := 0
for i := 1 to m do
  h := (h * d + Int(↓key[i])) mod q
end -- for
```

Berechnung ($s[i]$ bedeutet hier `Int(↓key[i])`):

$$h(i) = s[i] \cdot d^{m-1} + s[i+1] \cdot d^{m-2} + \cdots + s[\mathrm{i}+m-1] \cdot d^0$$
$$h(i+1) = \quad s[i+1] \cdot d^{m-1} + \cdots + s[\mathrm{i}+m-1] \cdot d^1 + s[\mathrm{i}+m] \cdot d^0$$

Inkrementelle Berechnung von $h(i+1)$:

$h = h - \mathrm{s[i]} \cdot d^{m-1}$ ← entfernt „Anteil“ des 1. Elements

$h = h \cdot d$ ← erhöht Exponent

$h = h + s[i+m]$

$h = h \bmod q$ ← addiert neues Element

Probleme bei inkrementeller Berechnung

Problem 1: $h = h - \mathrm{s[i]} \cdot d^{m-1}$

- d^{m-1} kann den gültigen Wertebereich übersteigen
- Statt mit d^{m-1} wird mit $d^{m-1} \bmod q$ multipliziert

```
dm := 1
for i := 2 to m do
  dm := (dm * d) mod q
end
```

Problem 2: $h = h - \mathrm{s[i]} \cdot d^{m-1}$

- Vorangehendes h kann sehr klein sein, sodass Subtraktion zu negativen Wert führt
- Vor Subtraktion wird Wert addiert, der ein Vielfaches von q und sicher größer als $\mathrm{s[i]} \cdot d^{m-1}$ ist; da $d^{m-1} < q$ ist kann man $d \cdot q$ nehmen

Das führt zu folgender Berechnung:

-- es wird d*q addiert, damit der Wert sicher positiv ist. Man kann jedes Vielfache von q addieren ohne dass Ergebnis zu verändern (mod ist der Rest)

```
h := (h + d * q - dm * Int(↓s[i])) mod q
h := (h * d + Int(↓(s[i + m])) mod q
```

Der Rabin-Karp-Mustersuchalgorithmus

```
RKMatching(↓s: array[1:n] of char ↓p: array[1:m] of char ↑pos: int)
  const
    q = 8355967
    d = 256
begin
  dm := 1
  for i := 2 to m do
    dm := (d * dm) mod q
  end -- for
  hs := 0
  hp := 0
  for i := 1 to m do
    hs := (hs * d + Int(↓s[i])) mod q
    hp := (hp * d + Int(↓p[i])) mod q
  end -- for
  …
```

möglichst großer Wert in int (wsl 64bit?)

berechnet $dm = d^{m-1} \bmod q$

Hashcode für die ersten m Zeichen der Zeichenkette

Hashcode für Muster

da man immer wieder mod rechnet, dam it man im wertebereich bleibt, gibt es unterschiedliche wörter mit den selben hash funktionen
in der Theorie bei einem unendlichen Wertebeeriech würd es eine eindeutige Funktion geben

Der Rabin-Karp-Mustersuchalgorithmus

```
  …
  i := 1
  while i ≤ n – m + 1 do
    if hs = hp then
      j := 1
      while (j ≤ m) and (s[i + j - 1] = p[j]) do
        j := j + 1
      end -- while
      if j > m then
        pos := i
        return
      end -- if
    end -- if
    if i < n – m + 1 then
      hs := (hs + d * q – dm * Int(↓s[i])) mod q
      hs := (hs * d + Int(↓s[i + m]) mod q
    end -- if
    i := i + 1
  end -- while
  pos := 0
end RKMatching
```

Vergleich der Zeichen in Muster und Teilkette, wenn Hashcodes übereinstimmen

inkrementelle Berechnung des nächsten Hashwerts

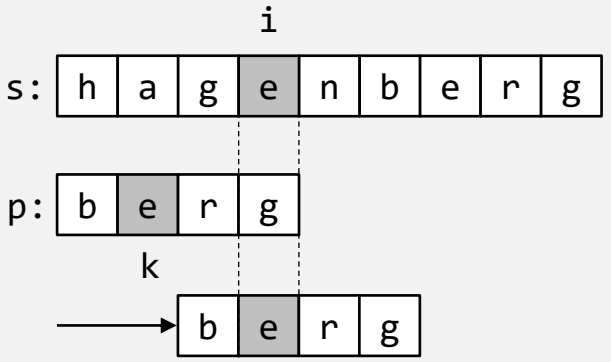
Laufzeitkomplexität: $O(n + m)$

11.5 Boyer-Moore Mustersuchalgorithmus

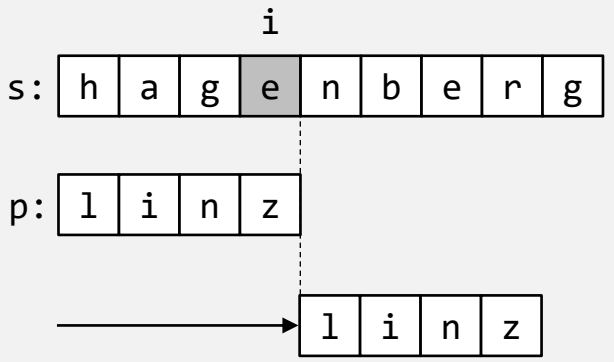
Muster wird von links nach rechts über Zeichenkette gelegt, der Zeichenvergleich erfolgt jedoch von rechts nach links (d. h. von hinten)

Bei mismatch an Position i Muster verschieben, uzw:

- Wenn mismatch-Zeichen `s[i]` in p an Position k vorkommt, wird Muster so über Text gelegt, dass `s[i]` und `p[k]` übereinander liegen
- Wenn mismatch-Zeichen `s[i]` in Muster p nicht vorkommt, scheiden die Positionen $\leq i$ als Startpositionen aus



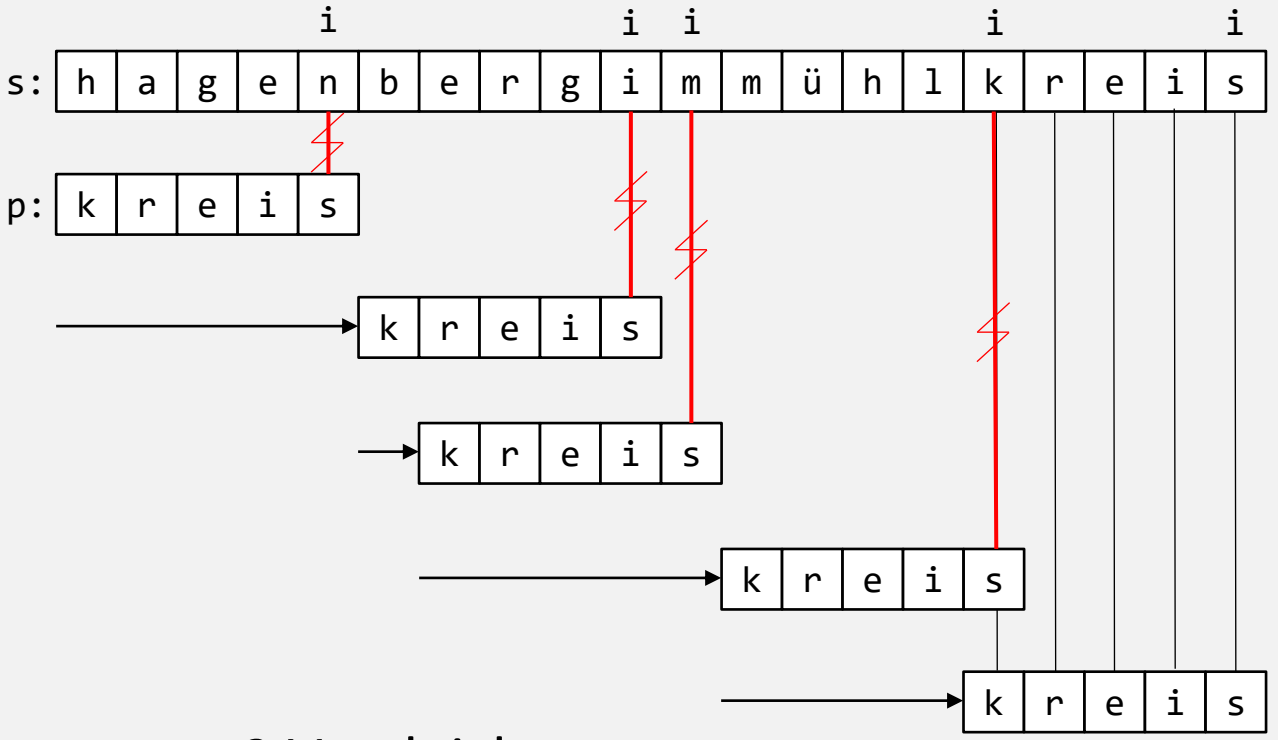
$s[i] \in p$



$s[i] \notin p$

Boyer-Moore Musteralgorithmus

Beispiel

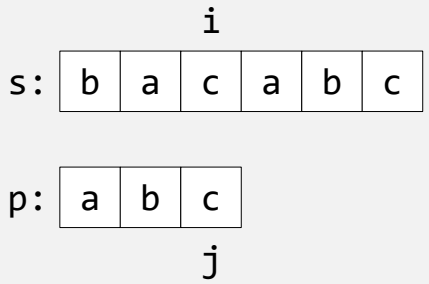


Insgesamt nur 9 Vergleiche

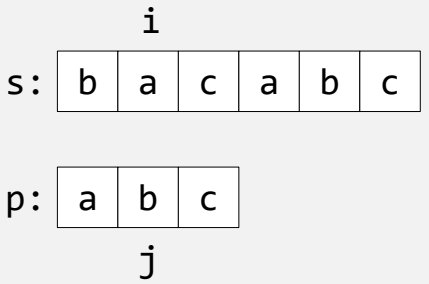
- 4 Vergleiche mit mismatch
- 5 Vergleiche mit match

`BruteForceMatching` Variante 3

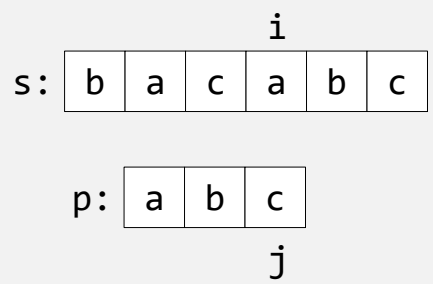
Muster p der Länge m wird von links nach rechts über Text s der Länge n gelegt und bei Nichtübereinstimmung nach rechts verschoben



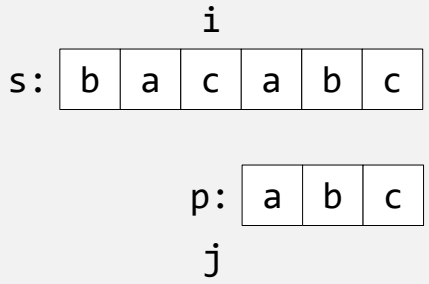
Beginn mit i = m und j = m



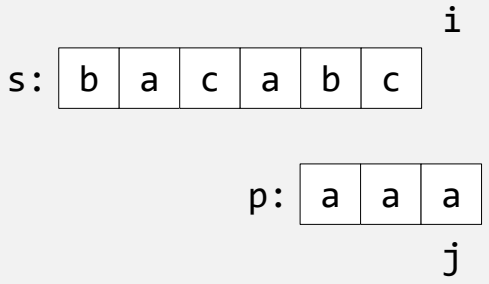
Bei Übereinstimmung werden i und j dekrementiert



Bei Nichtübereinstimmung wird Muster um eine Stelle nach rechts verschoben (i := i + m – j + 1, j := m)



Muster in Text an Position i + 1 enthalten, wenn j < 1



Muster in Text nicht enthalten, wenn i > n

BruteForceMatching Variante 3

Algorithmus

```
BruteForceMatching3(↓s: array[1:n] of char ↓p: array[1:m] of char ↑pos: int)
  var
    i, j: int
begin
  i := m
  j := m
  repeat
    if s[i] = p[j] then
      i := i - 1 ;  j := j - 1
    else
      i := i + m – j + 1;  j := m
    end -- if
  until (i > n) or (j < 1)
  if j < 1 then
    pos := i + 1
  else
    pos := 0
  end -- if
end BruteForceMatching3
```

Beginne rechts

Muster wird um 1 Stelle nach rechts verschoben

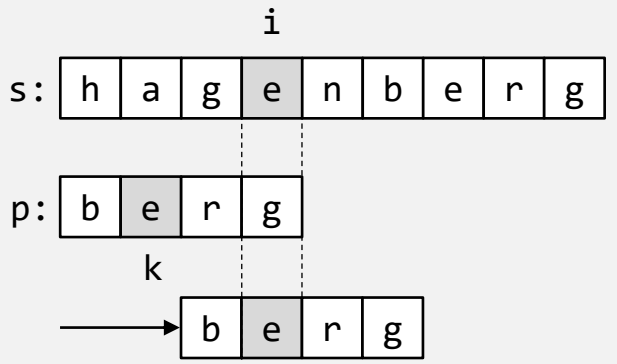
Muster gefunden

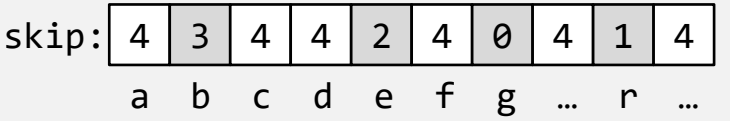
Der eigentliche Boyer-Moore-Mustersuchalgorithmus

- Neue Position des Musters nach mismatch hängt von `s[i]` und Musterinhalt ab
- Einrichtung einer Tabelle `skip` (über Alphabet), die die Verschiebungs-distanzen in Abhängigkeit vom mismatch-Zeichen `s[i]` enthält

```
var
  skip: array[char] of int
```

- Ist mismatch-Zeichen `c` nicht in `p` enthalten, ist `skip[c] := m`
- Ist mismatch-Zeichen `c` mehrfach in `p` enthalten, muss die am weitesten rechts liegende Position genommen werden
- Bei mismatch an Position `p[m]` wird `p` um eine Position verschoben





--> 0 heißt um eine Stelle weiter;
(glaub ich)

Aufbau der Tabelle skip

Algorithmus

```
InitSkip(↑skip: array[char] of int)
  var
    c: char
    j: int
begin
  for c := Char(↓0) to Char(↓255) do
    skip[c] := m
  end -- for
  for j := 1 to m do
    skip[p[j]] := m - j
  end -- for
end InitSkip
```

Beispiel: Einträge in die Tabelle `skip` für `p = "berg"` und `m = 4`

j	p[j]	m - j
1	b	3
2	e	2
3	r	1
4	g	0

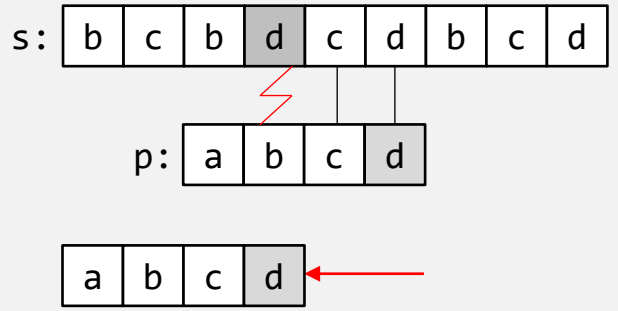
Algorithmus

```
BMMatching(↓s: array[1:n] of char ↓p: array[1:m] of char ↑pos: int)
  var
    i, j:Integer
    skip: array[char] of int
begin
  InitSkip(↑skip)
  i := m; j := m
  repeat
    if s[i] = p[j] then
      i := i - 1; j := j - 1
    else
      i := i + skip[s[i]]
      j := m
    end -- if
  until (i > n) or (j < 1)
  if j < 1 then
    pos := i + 1
  else
    pos := 0
  end -- if
end BMMatching
```

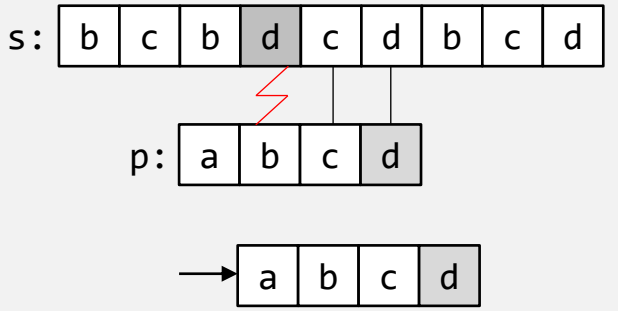
ersetzt i := i + m – j + 1

Problem

- Wenn mismatch-Zeichen `s[i]` auch rechts von der mismatch-Position auftritt, wird im bisherigen Algorithmus das Muster nach links verschoben



falsch



richtig

- Der Wert von `skip[s[i]]` wird nur verwendet, wenn rechts von `p[j]` kein `s[i]` steht
- Sonst wird Muster um eine Stelle nach rechts verschoben:

```
if (m – j + 1 > skip[s[i]]) then
  i := i + m – j + 1
else
  i := i + skip[s[i]]
end -- if
```

Laufzeitkomplexität

Günstigster Fall ($\frac{n}{m}$)

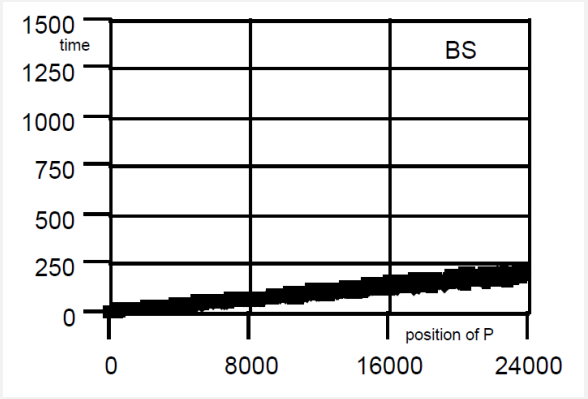
```
s := "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaabcdef"
p := "bcdef"
```

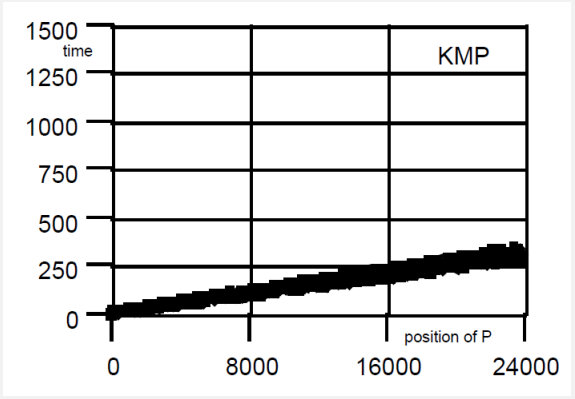
Ungünstigster Fall ($n \cdot m$ Vergleiche)

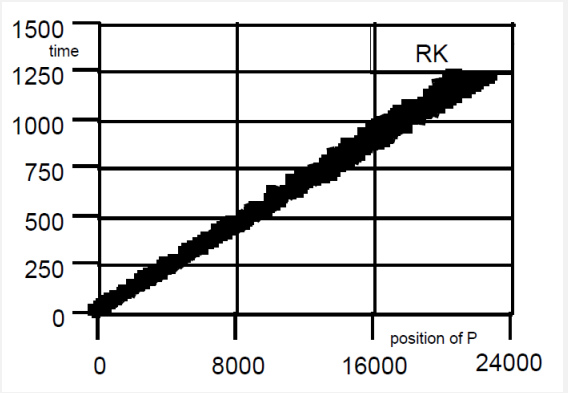
```
s := "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaabaaaa"
p := "baaaa"
```

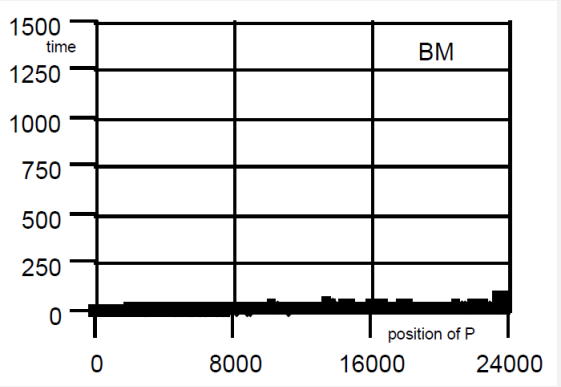
Für große Zeichensätze und nicht allzu lange Muster sind im Durchschnitt $\frac{n}{m}$ Vergleiche erforderlich

11.6 Vergleich der Laufzeiten









Pirklbauer: A Study of Pattern-Matching Algorithms, Structured Programming, Vol. 13, pp. 89-98, Springer-Verlag, Mai 1992.