Suche in Texten

- Naiver Algorithmus
- Knuth-Morris-Pratt-Algorithmus
- Karp-Rabin-Algorithmus

Suche in Texten

- Naiver Algorithmus
- Knuth-Morris-Pratt-Algorithmus
- Karp-Rabin-Algorithmus

Textsuchproblem (1)

Definitionen:

- Ein Alphabet ist eine endliche, nichtleere Menge von Zeichen.

- Sei A ein Alphabet. Ein Wort über A ist eine endliche Folge

$$w = w_0 \ldots w_{n-1} \quad \text{mit} \quad w_i \in A, \quad n \geq 0.$$

Hierbei ist $|w| = n$ die Länge des Wortes w.

- Das Wort der Länge 0 ist das leere Wort, es wird mit ε bezeichnet.

Der Begriff "Wort" nach dieser Definition bezeichnet also jede beliebige endliche Folge von Zeichen; die Bedeutung der Zeichen spielt keine Rolle.

Textsuchproblem (engl.: *string matching problem*):

> Gegeben ist ein Wort *t*, der Text, und ein kürzeres Wort *p*, der Suchbegriff oder das Muster. Gesucht sind alle Positionen *i*, an denen das Muster *p* im Text *t* vorkommt.

Textsuchproblem (2)

Definitionen:

Sei A ein Alphabet und seien $t = t_0 \ldots t_{n-1}$ und $p = p_0 \ldots p_{m-1}$ Wörter der Länge n bzw. m über A.

- Ein Textfenster w_i ist ein Teilwort von t der Länge m, die an Position i beginnt:

  $w_i = t_i \ldots t_{i+m-1}$.

- Ein Textfenster w_i, das mit dem Muster p übereinstimmt, heißt Vorkommen des Musters an Position i:

  w_i ist Vorkommen von $p \Leftrightarrow p = w_i$

- Ein Mismatch in einem Textfenster w_i ist eine Position j, an der das Muster mit dem Textfenster nicht übereinstimmt:

  j ist Mismatch in $w_i \Leftrightarrow p_j \neq (w_i)_j$.

Textsuchproblem

Eingabe: Text $t = t_0 \ldots t_{n-1}$ und Muster $p = p_0 \ldots p_{m-1}$ mit $m \leq n$

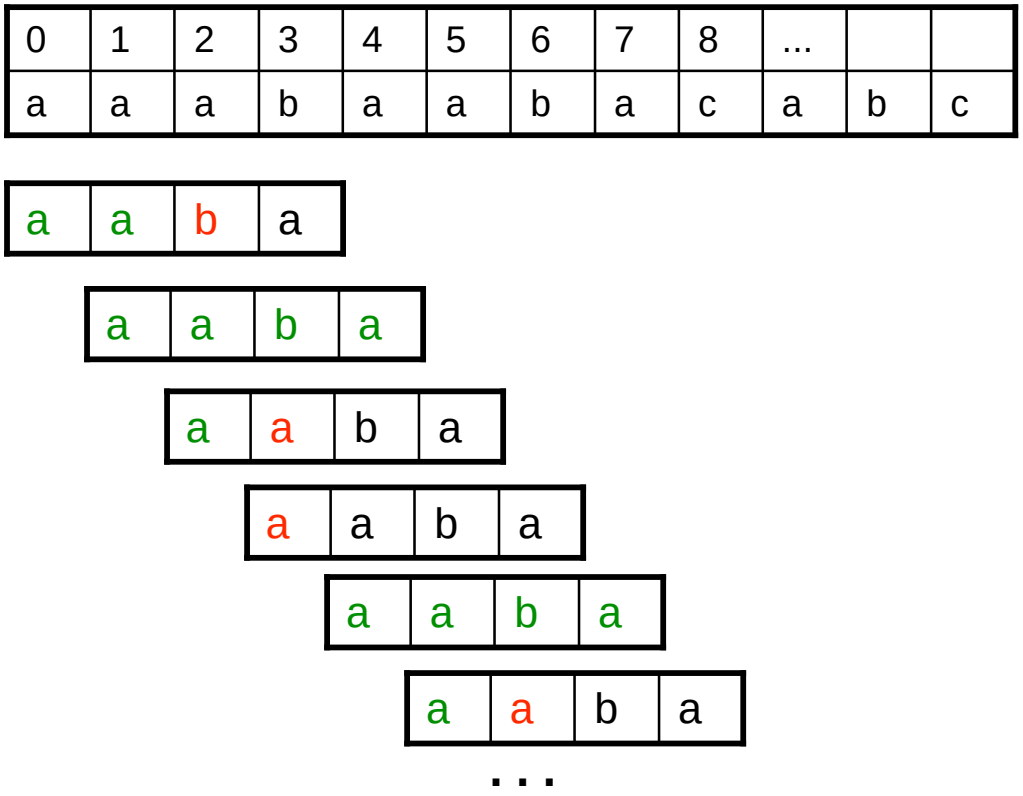
Ausgabe: Alle Positionen von Vorkommen von p in t, d.h. $\{ i \mid p = w_i \}$

Naiver Algorithmus

Idee:

Überprüfung des Textes an allen Positionen i auf Vorkommen des Musters, d.h. von Position i = 0 (linksbündig) bis i = n - m (rechtsbündig). An jeder Position wird das Muster zeichenweise von links nach rechts verglichen, bei Mismatch oder vollständiger Übereinstimmung wird Position um 1 nach rechts verschoben.

Beispiel:



Naiver Algorithmus - Analyse

```
void naiveSearch()
{
      int i = 0;
      while (i <= n-m)
      {
            int j = 0;
            while (j<m && p[j] == t[i+j])   j++;
            if (j == m) report(i);
            i++;
      }
}
```

- Äußere Schleife wird n - m + 1 mal durchlaufen
- Die innere Schleife wird höchstens m mal durchlaufen
- Die Maximalzahl an Vergleichen ist damit (n - m +1) * m, somit gilt im schlechtesten Fall T(n, m) = O(m*n)

Der schlechteste Fall tritt z.B. für t = aaaaaaa...aaa und p = aaaaab ein.

Verhalten im Durchschnitt: Sei h die Häufigkeit des häufigsten Zeichens in t. Dann ist die Anzahl der Vergleiche pro Position beschränkt durch

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

Damit ergibt sich durchschnittlich $T(n) = (n + m + 1) / (1 - h) \in O(n)$.
z.B. ist e im Deutschen am häufigsten (h = 0.13), => durchschnittlich 1/(1 - 0.13) = 1.15 Vergleiche pro Textzeichen.

Verbesserung durch Berücksichtigung der Häufigkeit

Idee:

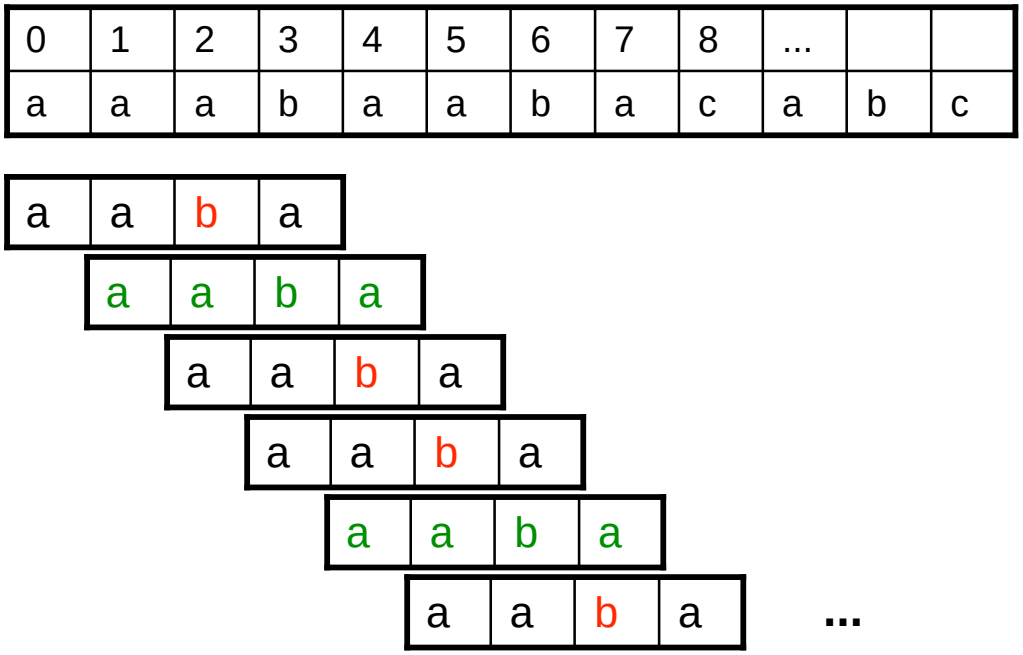
• Beim naiven Algorithmus ist es nicht notwendig, die Zeichen des Musters p in aufsteigender Reihenfolge p_0, ..., p_{m+1} mit dem Text zu vergleichen, die Reihenfolge ist beliebig.

• Am besten ist es, zuerst das seltenste Zeichen zu vergleichen, da so ein Mismatch am wahrscheinlichsten wird und damit die innere Schleife möglichst schnell abgebrochen wird.

• erfordert Kenntnis der Häufigkeitsverteilung der Zeichen im Text.

Beispiel:

b ist seltener als a



Verbesserter Algorithmus

```
void notsoNaiveSearch()
{
      int i = 0;
      while (i <= n-m)
      {
            int j = 0;
            while (j<m && p[ g[j] ] == t[ i+ g[j] ])
                  j++;
            if (j == m) report(i);
            i++;
      }
}
```

• Die geänderte Vergleichsreihenfolge ist eine Permutation der Indizes von p: Dem Zeichen No. k in p wird ein Platz g[k] ∈ 1..m-1 zugewiesen. Beim naiven Algorithmus ist g[k] = k.

• g[k] muß im Vorfeld durch ein Sortierverfahren aus den Häufigkeiten berechnet werden (O(m log m))

Analyse:

Sei h_j die Häufigkeit des Zeichens an der Auswerteposition g[j]. Pro Textposition ergeben sich

$$v = 1 + h_0 + h_0\, h_1 + \ldots + h_0\, h_1 \ldots h_{m-2}$$

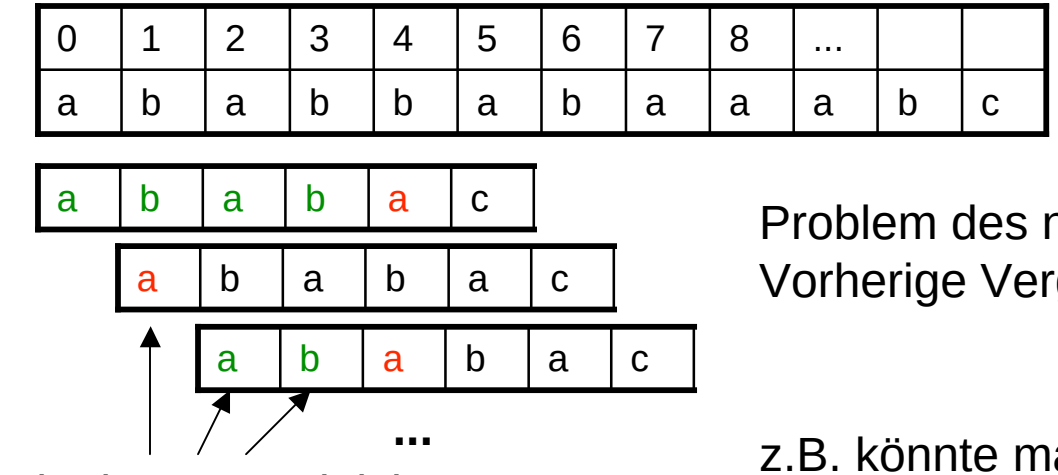
Vergleiche.

Sind alle Zeichen gleich häufig, so ergibt sich kein Vorteil gegenüber dem naiven Algorithmus. Ist ein Zeichen in p sehr selten, erhält man eine signifikante Einsparung.

Suche in Texten

- Naiver Algorithmus
- Knuth-Morris-Pratt-Algorithmus
- Karp-Rabin-Algorithmus

Knuth-Morris-Pratt-Algorithmus



Problem des naiven Algorithmus:
Vorherige Vergleiche werden vergessen.

z.B. könnte man Suchfenster gleich um 2 weiterschieben und an Pos. 4 weitervergleichen, da man bereits weiß, das Pos. 2 und 3 in t und Pos. 0 und 1 in p die Folge ab enthält.

Idee:

In einer Vorlaufphase wird das Suchmuster analysiert und Information über seine Struktur abgespeichert. Aus der Art und Position des Mismatches wird über die Strukturinformation berechnet, wie weit man springen darf, bevor weiterverglichen werden muß.

Präfix, Suffix und Rand von Zeichenketten

Definitionen:
Sei A ein Alphabet und $x = x_0 \ldots x_{k-1}$ ein Wort der Länge k über A.

• Ein Präfix von x ist ein Teilwort u mit $u = x_0 \ldots x_{b-1}$ wobei $b \in \{0, \ldots, k\}$, also ein Anfangswort der Länge b von x.

• Ein Suffix von x ist ein Teilwort u mit $u = x_{k-b} \ldots x_{k-1}$ wobei $b \in \{0, \ldots, k\}$, also ein Endwort der Länge b von x.

• Ein Präfix u von x bzw. Suffix heißt echt, wenn $u \neq x$ ist, d.h. wenn die Länge $b < k$ ist.

• Ein Rand von x ist ein Teilwort r mit $r = x_0 \ldots x_{b-1}$ und $r = x_{k-b} \ldots x_{k-1}$ wobei $b \in \{0, \ldots, k-1\}$. Ein Rand ist also ein Wort, das gleichzeitig ein echtes Präfix und ein echtes Suffix von x ist. Die Länge b ist die Breite des Randes r.

Beispiel: x = abacab

Echte Präfixe: ε, a, ab, aba, abac, abaca
Echte Suffixe: ε, b, ab, cab, aca, bacab
Ränder: ε, ab mit Breite 0 bzw. 2

Schiebedistanz aus dem Rand des Präfix

Beispiel:

0	1	2	3	4	5	6	7	8	9	...	
a	b	c	a	b	c	a	b	d	b	b	c

a	b	c	a	b	d

Muster kann bis Pos. 3 geschoben werden →

a	b	c	a	b	d

Vergleich wird ab Pos. 5 fortgesetzt

Sowohl Schiebedistanz als auch Fortsetzungspunkt können berechnet werden, ohne nochmals den Text anzuschauen.

Übereinstimmendes Präfix:	abcab	(Länge j = 5)
Breitester Rand:	ab	(Breite b = 2)
Schiebedistanz:	j - b = 5 - 2 = 3	

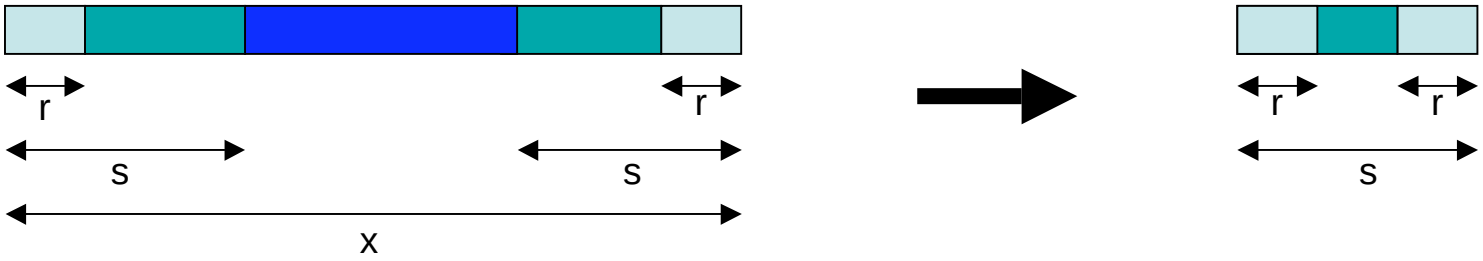
Grundprinzip:

- Schiebedistanz richtet sich nach dem breitesten Rand des mit dem Text übereinstimmenden Präfixes des Musters
- In einem Vorlauf werden für jedes Präfix des Musters die Breite seines breitesten Randes bestimmt.

Ränder von Rändern...

Satz:

Seien r und s Ränder eines Wortes x, wobei die Breite von r kleiner als die Breite von s ist. Dann ist r ein Rand von s.



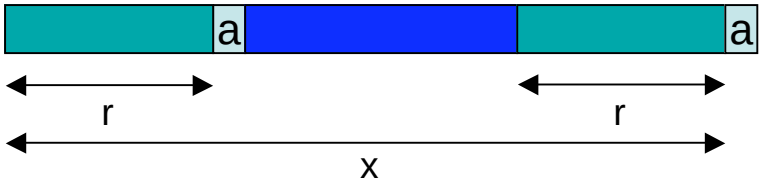
Beweis:

Als Rand von x ist r ein Präfix von x und damit auch ein echtes Präfix von s, weil r schmaler als s ist. Aber r ist auch Suffix von x und damit ein echtes Suffix von s. Also ist r ein Rand von s. □

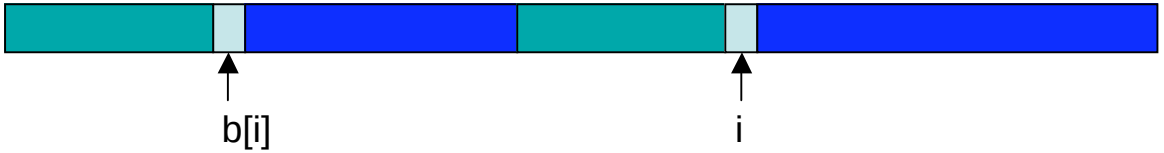
Ist s der breiteste Rand von x, so ergibt sich der nächstschmalere Rand r von x als breitester Rand von s usw.

Vorlaufalgorithmus (1)

Definition:
Sei x ein Wort und a ein Zeichen ein Rand r von x läßt sich durch a fortsetzen, wenn ra ein Rand von xa ist.



Ein Rand r von x der Breite j läßt sich durch a fortsetzen, wenn x_j = a ist.



- In der Vorlaufphase wird ein Array b der Länge m + 1 berechnet, bei dem der Eintrag b[i] die Breite des breitesten Randes des Präfix der Länge i enthält.
- Das Präfix der Länge i = 0 hat keinen Rand, daher wird b[0] = -1 gesetzt.
- Sind b[0], ..., b[i] bereits bekannt, wird geprüft, ob sich ein Rand des Präfixes der Länge i durch p_i fortsetzen läßt, d.h. $p_{b[i]} = p_i$.
- Die zu prüfenden Ränder ergeben sich nach obigem Satz in absteigender Breite aus b[i], b[b[i]], ...

Vorlaufalgorithmus (2)

```
void kmpPreprocess()
{
      int i = 0, j = -1;
      b[0] = -1;
      while (i < m)
      {
            while (j>=0 && p[i] != p[j])   j = b[j];
            i++; j++
            b[i] = j;
      }
}
```

Überprüfung, ob Rand sich fortsetzen läßt.

Falls nicht, gehe zum nächstschmaleren Rand

Schleife endet, wenn es keinen fortsetzbaren Rand mehr gibt, d.h. j = -1

Beispiel:

Randbreiten für das Muster p = ababaa, z.B. ist b[5] = 3, weil das Präfix ababa der Länge 5 einen Rand aba der Breite 3 hat.

j:	0	1	2	3	4	5	6
p[j]:	a	b	a	b	a	a	
b[j]:	-1	0	0	1	2	3	1

Knuth-Morris-Pratt-Suchalgorithmus (1)

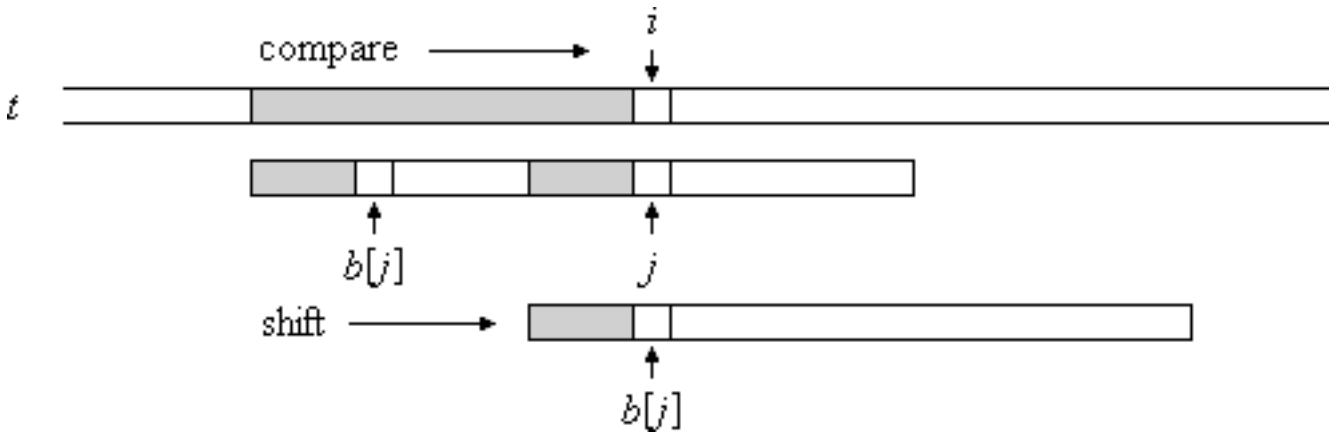
```
void kmpSearch()
{
      int i = 0, j = 0;
      while (i < n)
      {
            while (j>=0 && t[i] != p[j])   j = b[j];
            i++; j++
            if (j == m)
            {
                  report(i - j);
                  j = b[ij;
            }
      }
}
```

In der inneren Schleife wird bei einem Mismatch an Position j der breiteste Rand mit Breite b[j] des übereinstimmende Präfixes betrachtet

Fortsetzung des Vergleichs an Pos. b[j]

Abbruch wenn kein Rand mehr vorhanden ist (j = -1), d.h. keine Sprünge mehr gemacht werden können, Übergang zur nächsten Pos. i in t.

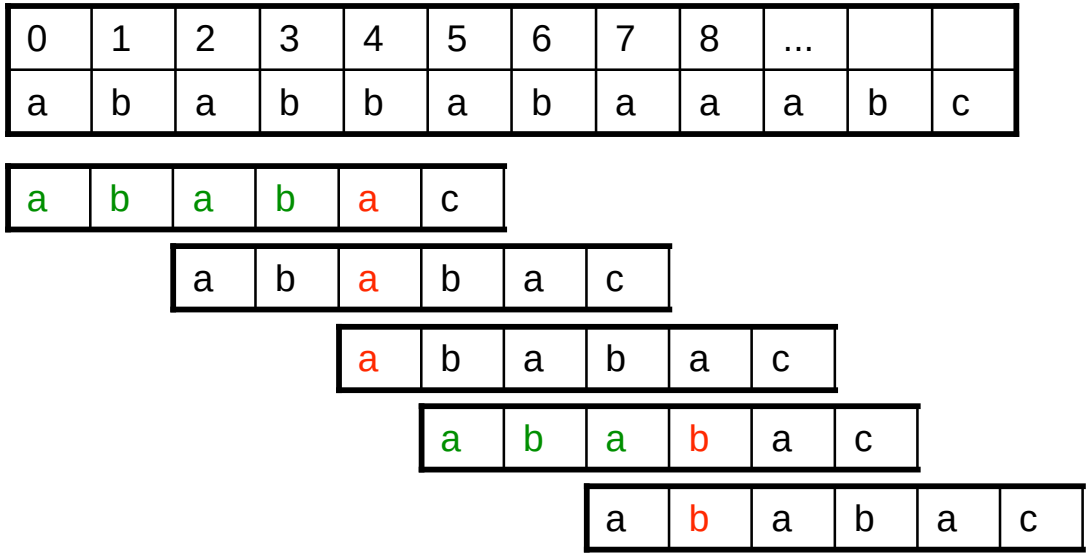
Übereinstimmung gefunden, springe so weit, wie es der breiteste Rand zuläßt.



[Lang, 2006]

Knuth-Morris-Pratt-Suchalgorithmus (2)

Beispiel:



Analyse: ...

Anschaulich: die durchgeführten Vergleiche (rote und grüne Felder) bilden eine Treppe. Jede Stufe kann höchsten so hoch sein, wie sie breit ist, somit werden maximal 2n Vergleiche durchgeführt => Suche ist O(n).

Der Vorlaufalgorithmus ist O(m) (s. Lang, 2006), es gilt m < n, daher hat der gesamte Algorithmus Laufzeit O(n). Der Gewinn gegenüber dem naiven Algorithmus liegt in der Nutzung bereits vorher durchgeführter Vergleiche.

Suche in Texten

- Naiver Algorithmus
- Knuth-Morris-Pratt-Algorithmus
- Karp-Rabin-Algorithmus

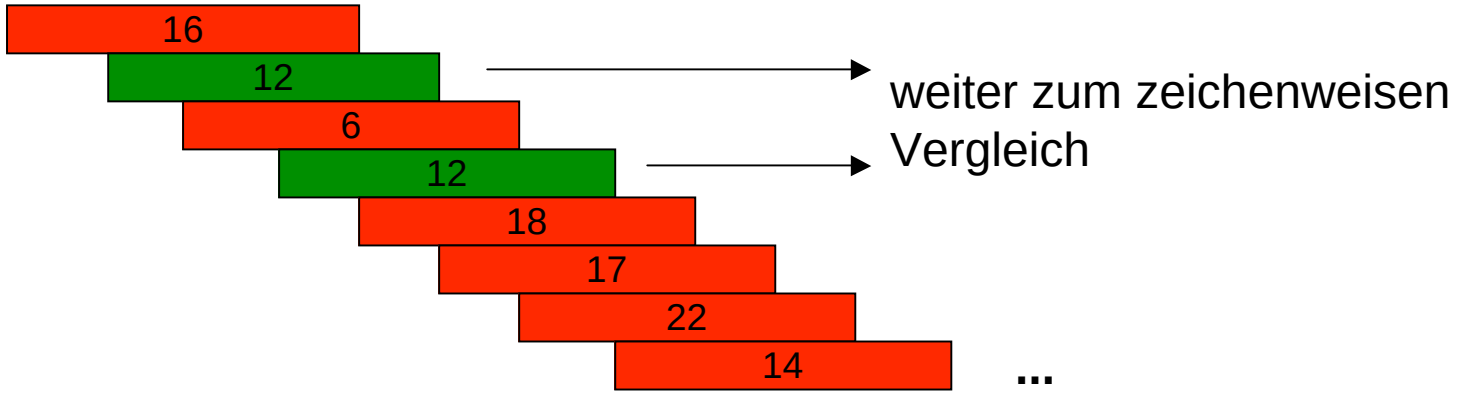
Algorithmus von Karp-Rabin

Idee:

Grundaufbau ähnlich wie beim naiven Algorithmus: das Muster wird mit jedem Textfenster verglichen. Aber statt an jeder Position zeichenweise zu vergleichen, wird nur ein einziger Vergleich durchgeführt, und zwar zwischen 2 Signaturen (eindeutige numerische Merkmale) von Textfenster und Muster.

Beispiel: Alphabet A = {0, ..., 9}, Muster p = 1 3 0 8
Signaturfunktion: Quersumme
Quersumme von p ist 12

0	1	2	3	4	5	6	7	8	...		
7	6	2	1	3	0	8	7	2	5	0	8



Signaturfunktion

Definitionen:

• Sei A ein Alphabet und M eine Menge. Eine Signaturfunktion ist eine Abbildung s: $A^* \rightarrow M$, die jedem Wort $x \in A^*$ einen Wert $s(x) \in M$ zuordnet. Der Wert s(x) ist die Signatur von x.

• Sei eine Signaturfunktion. Eine Kollision ist ein Paar von Wörtern (x, x‘) mit $x \neq x'$, aber $s(x) = s(x')$.

Anforderungen an Signaturfunktion (vgl. Hashfunktion):

- Kollisionen sollten möglichst ausgeschlossen sein
- Signatur muß in konstanter Zeit berechenbar sein.

Beispiel:
Quersumme hat häufig Kollisionen, ist aber sehr effektiv berechenbar: die Zahl, die das Textfenster verläßt, wird subtrahiert, die neu hinzukommende Zahl addiert.

Signaturfunktion des Karp-Rabin-Algorithmus

Grundidee: Bildung einer Dezimalzahl aus den Ziffern der Zeichenreihe, z.B. 1308 aus der Folge 1 3 0 8 (bei A = {0,..,9} sind Kollisionen so ausgeschlossen).

Für schnelle Berechenbarkeit sollten die resultierenden Zahlen in ein Maschinenwort passen: Zahlen mit höchstens 32 Bit in Binärdarstellung passen in ein Maschinenwort, bei mehr als 32 Bit werden nur die letzten 32 Bit berücksichtigt (=> Gefahr von Kollisionen). Ebenso erhöht sich die Kollisionswahrscheinlichkeit durch die Verwendung des Binärsystems statt eines Systems mit Basis 10 oder 26.

Signaturfunktion (Karp, Rabin): $$s(x_0 x_1 ... x_{m-1}) = \sum_{j=0,...,m-1} 2^j \cdot x_{m-j-1} \bmod 2^{32}$$

Die Signatur s‘ eines neuen Fensters t_{i-m+1} ... t_i wird aus der Signatur s des vorhergehenden Fensters t_{i-m} ... t_{i-1} wie folgt berechnet:

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

Im schlimmsten Fall (100% Kollisionen z.B. bei p = aaa... und t = aaa...) ist die Laufzeit O(nm), im Durchschnitt aber O(n).

Karp-Rabin-Algorithmus

```
void krPreprocess()
{
      int i = 0;
      sp = p[0];

      for (i = 1; i < m; i++)
            sp = (sp << 1) + p[i];
}
```

sp, p, t seien innerhalb des zugehörigen Objekts deklariert

Vorlauf: Berechnung der Signatur des Musters

mod 2^{32} wird implizit durch die 32-Bit-Zahlendarstellung bewirkt.

```
void krSearch()
{
      int i = 0, st = 0;

      for (i = 1; i < m; i++)
            st = (st << 1) + t[i];

      for (i = m; i < n; i++)
      {
            if ( sp == st && matchesAt(i-m) )  report(i-m);
            st = ( (st - (t[i-m] << m-1)) << 1) + t[i];
      }
      if ( sp == st && matchesAt(n-m) )  report(n-m);
}
```

Initialisierung der Signatur mit erstem Suchfenster

Funktion matchesAt überprüft zeichenweise, ob Fenster und Muster übereinstimmen.

Achtung: für m-1 > 32 muß die Multiplikation mit 2^{m-1} mod 2^{32} evtl. anders realisiert werden.