Algorithmische Bioinformatik

Exaktes Stringmatching: Z-Box Algorithmus

Ulf Leser

Wissensmanagement in der Bioinformatik

Ziele für heute

- Analyse des naiven Stringmatching-Algorithmus
- Verständnis der Funktionsweise, Komplexität und Korrektheit des Z-Box Algorithmus
- Ins rechte Licht rücken: Worst case, average case im String Matching

Inhalt dieser Vorlesung

- Naiver Algorithmus für exaktes Stringmatching
- Z-Box Algorithmus
- Berechnung von Z-Boxen
- Average-Case Komplexitäten

Exaktes Matching

- Gegeben: P (Pattern) und T (Text)
  - Trivialerweise verlangen wir $|P| \leq |T|$
- Gesucht: Sämtliche Vorkommen von P in T
- Beispiel: Erkennungssequenzen von Restriktionsenzymen

Eco RV - GATATC

```
tcagcttactaattaaaaattctttctagtaagtgctaagatcaagaaaataaattaaaaataatggaacatggcacattttcctaaactcttcacagattgctaatga
ttattaattaaagaataaatgttataatttttatggtaacggaatttcctaaaatattaattcaagcaccatggaatgcaaataagaaggactctgttaattggtact
attcaactcaatgcaagtggaactaagttggtattaatactctttttacatatatatgtagttattttaggaagcgaaggacaatttcatctgctaataaagggattac
atatttattttgtgaatataaaaaatagaaagtatgttatcagattaaacttttgagaaaggtaagtatgaagtaaagctgtatactccagcaataagttcaaataggc
gaaaaacttttaataacaaagttaaataatcattttgggaattgaaatgtcaaagataattacttcacgataagtagttgaagatagtttaaatttttctttttgtatt
acttcaatgaaggtaacgcaacaagattagagtatatatggccaataaggtttgctgtaggaaaattattctaaggagatacgcgagagggcttctcaaatttattcaga
gatggatgttttagatggtggtttaagaaaagcagtattaaatccagcaaaactagaccttaggtttattaaagcgaggcaataagttaattggaattgtaaaagatat
ctaattcttcttcatttgttggaggaaaactagttaacttcttaccccatgcagggccatagggtcgaatacgatctgtcactaagcaaaggaaaatgtgagtgtagact
ttaaaccattttattaatgactttagagaatcatgcatttgatgttactttcttaacaatgtgaacatatttatgcgattaagatgagttatgaaaaaggcgaatatat
tattcagttacatagagattatagctggtctattcttagttataggacttttgacaagatagcttagaaaataagattatagagcttaataaaagagaacttcttggaat
tagctgcctttggtgcagctgtaatggctattggtatggctccagcttactggttaggttttaatagaaaaattccccatgattgctaattatatctatcctattgagaa
caacgtgcgaagatgagtggcaaattggttcattattaactgctggtgctatagtagttatccttagaaagatatataaatctgataaagcaaaatcctggggaaaatat
tgctaactggtgctggtagggtttggggattggattatttcctctacaagaaatttggtgtttactgatatccttataaataatagagaaaaaattaataaagatgatat
```

Naiver Ansatz

1. P und T an Position 1 ausrichten
2. Vergleiche P mit T von links nach rechts (innere Schleife)
   - Zwei ungleiche Zeichen ⇒ Gehe zu 3
   - Zwei gleiche Zeichen
     - P noch nicht durchlaufen ⇒ Verschiebe Pointer nach rechts, gehe zu 2
     - P vollständig durchlaufen ⇒ Merke Vorkommen von P in T
3. Verschiebe P um 1 Zeichen nach rechts (äußere Schleife)
4. Solange Startposition <= |T|-|P|, gehe zu 2

```
T   ctgagatcgcgta
P   gagatc
     gagatc
      gagatc
       gagatc
        gagatc
         gatatc
          gatatc
           gatatc
```

Naiver Ansatz (cont.)

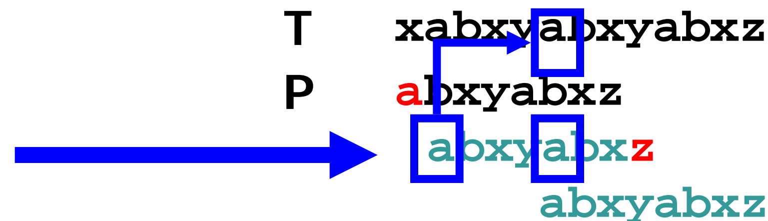
```
for i = 1 to |T| - |P| + 1
      match := true;
      j := 1;
      while ((match) and (j <= |P|))
            if (T[i+j-1] <> P[j]) then
                  match := false;
            else
                  j++;
      end while;
      if (match) then
            -> OUTPUT i
end for;
```

```
T    aaaaaaaaaaaaaa
P    aaaaat
      aaaaat
       aaaaat
        aaaaat
              ...
```

Vergleiche : n * (m-n+1) => O(m*n)

Optimierungsidee

- Anzahl der Vergleiche reduzieren
  - P um mehr als ein Zeichen verschieben
  - Aber nie soweit, dass ein Vorkommen von P in T nicht erkannt wird
- Idee am Beispiel



- Vorkommen in T muss mit a beginnen
- Nächstes a in T erst an Position 6 – springe 4 Positionen
- Vorkommen von Buchstaben in T durch Preprocessing lernen
  - Naiv: Von T
  - Besser: Von P

Erweiterung auf Substrings

```
T     xabxyabxyabxz
P     abxyabxz
       abxyabxz
           abxyabxz
```

- abx ist doppelt in P - interne Struktur von P erkennen
  - P[1..3] = P[5..7]
  - Kein Vorkommen dazwischen
- Vergleich findet: P[1..7] = T[2..8]
- Daher
  - P[1..3] = T[6..8]; zwischen 2 und 6 kann in T kein Treffer liegen
  - 4 Zeichen schieben und erst ab Position 4 in P weiter vergleichen

Inhalt dieser Vorlesung

- Naiver Algorithmus für exaktes Stringmatching
- Z-Box Algorithmus
- Berechnung von Z-Boxen
- Average-Case Komplexitäten

Z-Box Algorithmus

- Grobaufbau
  - Konstruktion eines „geeigneten“ Strings S aus P und T
  - Berechnung von Z-Boxen an jeder Position i von S: Längster Substring, der an Position i startet und auch Präfix von S ist
    - Längstes x mit $S[i..i+|x|-1]=S[1..|x|]$
  - Alle Z-Boxen einer bestimmten Länge sind Matches

- Wichtig: Z-Boxen müssen schnell berechnet werden
  - Lineare Komplexität

Z-Boxen

- Definition Z-Box
  - *Für i>1 sei $Z_i(S)$ die Länge des längsten Substrings x von S mit*
    - *x=S[i..i+|x|-1]* *(x startet an Position i in S)*
    - *S[i..i+|x|-1] = S[1..|x|]* *(x ist auch Präfix von S)*
  - *Dann nennen wir x die Z-Box von S an Position i mit Länge $Z_i(S)$*
- Anmerkung
  - Wir bezeichnen mit Z-Box oft auch den String x selber (statt seiner Länge)

Beispiele

```
S = aabcaabxaaz
     1(a)
      0
       0
        3(aab)
         1(a)
          0
           0
            2(aa)
             1(a)
              0
```

```
S = aaaaaa
```

```
S = baaaaa
```

Beispiel 2

	A	C	A	T	A	C	A	C	A	T	A	G	
Z_2		C	A	T	A	C	A	C	A	T	A	G	0
Z_3			A	T	A	C	A	C	A	T	A	G	1
Z_4				T	A	C	A	C	A	T	A	G	0
Z_5					A	C	A	C	A	T	A	G	3
Z_6						C	A	C	A	T	A	G	0
Z_7							A	C	A	T	A	G	5
Z_8								C	A	T	A	G	0
Z_9									A	T	A	G	1
Z_{10}										T	A	G	0
Z_{11}											A	G	1
Z_{12}												G	0

Linearer Stringmatching Algorithmus

- Annahme: Z-Boxen lassen sich in O(|S|) berechnen
  - Wie? Später
- Verwendung der Z-Boxen für exaktes Stringmatching
  - Wie muss S aussehen, um unser Problem zu lösen?

```
S := P||‘$‘||T;                    // ($ ∉ Σ)
compute Z-Boxes for S;
for i = |P|+2 to |S|-|P|-1
       if (Zi(S)=|P|) then
              print i-|P|-1;  // P in T at position i
       end if;
end if;
```

- Komplexität?
  - Berechnung Z-Boxen: Unklar
  - Schleife wird |T|-|P|-mal durchlaufen => O(m)

Inhalt dieser Vorlesung

- Naiver Algorithmus für exaktes Stringmatching
- Z-Box Algorithmus
- Berechnung von Z-Boxen
- Average-Case Komplexitäten

Berechnung der Z-Boxen

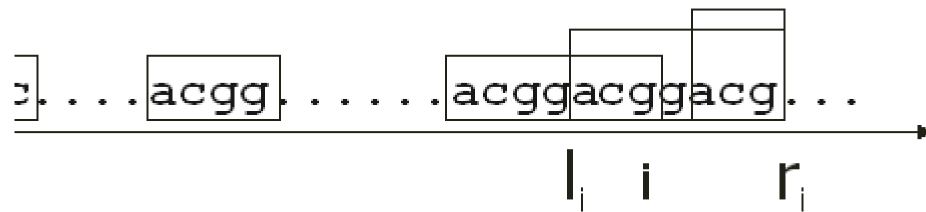
- Naiver Algorithmus: $O(|S|^2)$

```
for i = 2 to |S|
        Zi := 0;
        j := 1;
        while (((S[j] = S[i + j – 1])) and (i+j <= |S|))
                Zi := Zi + 1;
                j := j + 1;
        end while;
end for;
```

- Das wäre schlechter als der naive Algorithmus
  - $O((m+n)^2) + O(m) \sim O(m^2)$

Vorarbeiten

- Definition
  - *Für i > 1 ist*
    - *r_i der rechteste Endpunkt aller Z-Boxen, die bei oder vor i beginnen*
    - *l_i ist die Startposition der längsten Z-Box, die bei r_i endet*
- l_i eindeutig, da an jeder Position nur eine Z-Box beginnt
- $S[l_i..r_i]$ ist die Z-Box, die die Position i von S enthält, am weitesten nach rechts reicht und am längsten ist



Berechnung der Z_i Werte

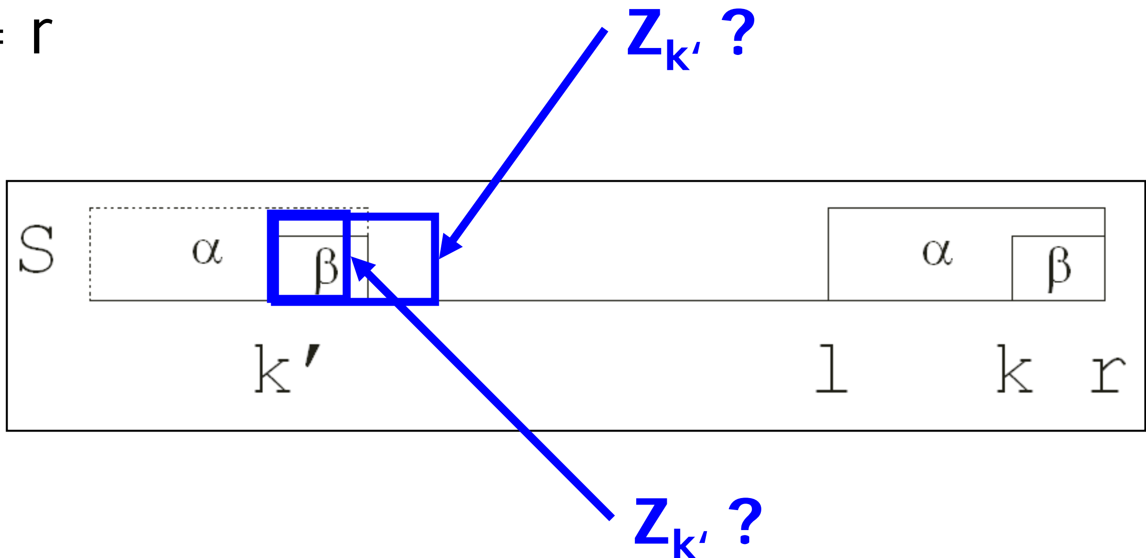
- Idee: Verwende bekannte Z_i zur Berechnung von Z_k $(k > i)$
- Grundaufbau
  - Einmaliges Durchlaufen von S (Laufvariable k)
  - Kontinuierliches Vorhalten der aktuellen Werte $l=l_k$ und $r=r_k$
  - Größe der Z-Box an Position k ergibt sich mit einigen Tricks in insgesamt linearer Zeit
- Induktive Erklärung
  - Induktionsanfang: Position $k=2$
    - Berechne Z_2
    - Wenn $Z_2 > 0$, setze $r=r_2$ $(=2+Z_2-1)$ und $l=l_2$ $(=2)$,
    - sonst $r=l=0$
  - Induktionsschritt: Position $k>2$
    - Bekannt sind r, l und $\forall j<k$: Z_j

Z-Algorithmus, Fall 1

- Möglichkeit 1: k > r
  - D.h., dass es keine Z-Box vor k gibt, die k überdeckt
  - Wir wissen also nichts über den Bereich ab k
  - Dann gehen wir naiv vor
    - Berechne Z_k durch Zeichen-für-Zeichen Matching
    - Wenn $Z_k>0$, setze $r=r_k$ und $l=l_k$

Z-Algorithmus, Fall 2

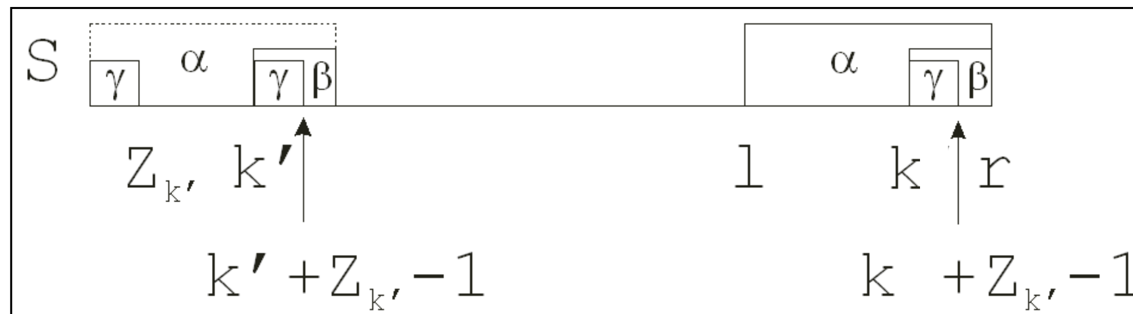
- Möglichkeit 2: k <= r
  - Die Situation



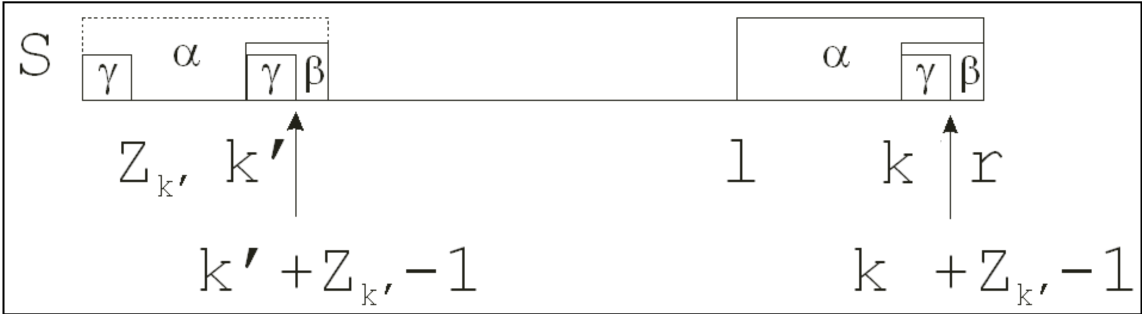
- Also
  - Z-Box Z_l ist Präfix von S
  - Substring β=S[k..r] kommt auch an Position k'=k-l+1 vor
  - Was wissen wir über S[k'..]? Natürlich: $Z_{k'}$
  - $Z_{k'}$ und Z_k können aber länger oder kürzer als |β|=r-k+1 sein
  - S[r+1..] kennen wir noch nicht; S[k'+1..] schon

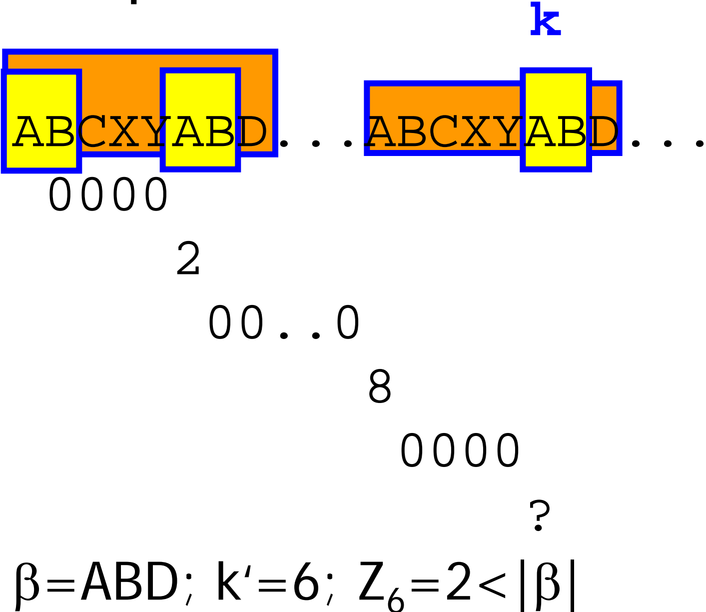
Z-Algorithmus, Fall 2.1

- Möglichkeit 2.1: $Z_{k'} < |\beta|=r-k+1$
  - Also ist das Zeichen an $k'+Z_{k'}$ ein Mismatch bei der Präfixverlängerung
  - Da $S[k+Z_{k'}]=S[k'+Z_{k'}]$, erzeugt $S[k+Z_{k'}]$ den gleichen Mismatch
  - Also muss gelten: $Z_k=Z_{k'}$; r und l unverändert



Beispiel

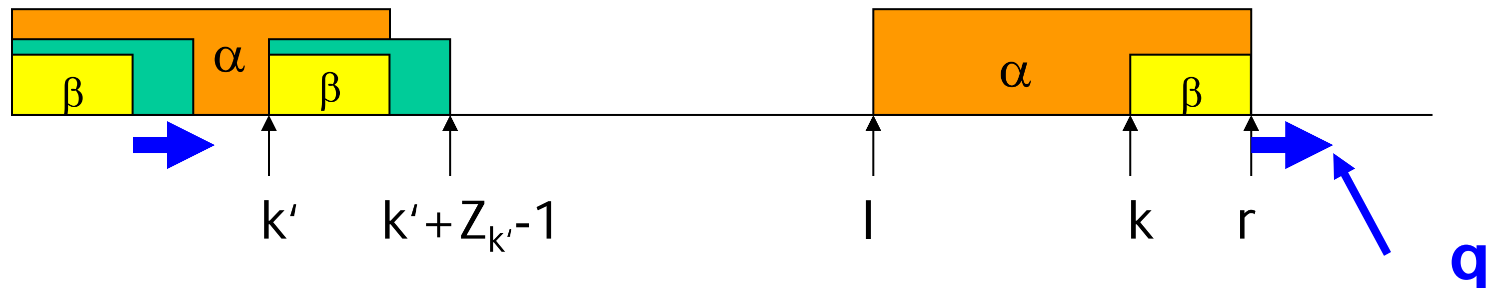




β=ABD; k'=6; Z_6=2<|β|

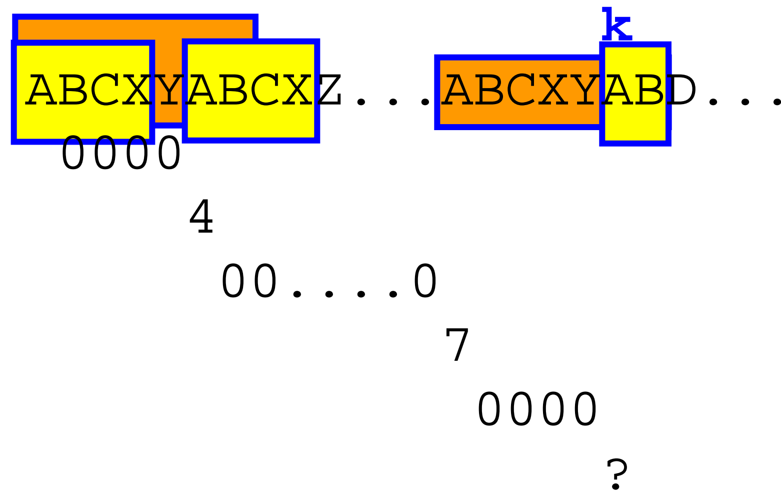
Z-Algorithmus, Fall 2.2

- Möglichkeit 2.2: $Z_{k'} \geq |\beta|$
  - β ist Präfix von S, das sich vielleicht (hinter r) verlängern lässt
  - Wenn $Z_{k'} > |\beta|$, dann wissen wir: $S[|\beta|+1]=S[k'+|\beta|]$
  - Wir wissen aber wenig über $S[r+1]$
    - Wurde bisher höchstens in Mismatches betrachtet
  - Also
    - Matche Zeichen für Zeichen $S[r+1..]$ mit $S[|\beta|+1..]$
    - Sei der erste Mismatch an Position q
    - Dann: $Z_k=q-k$; $r=q-1$; wenn $q \neq r+1$: $l=k$



Beispiel

Beispiel



β=AB; k'=6; $Z_6=4>|\beta|$

Algorithmus

```
match Z2; set l,r;
for k = 3 to |S|
        if k>r then
                match Zk;
                set r,l;
        else
                k‘ := k-l+1;
                b := r-k+1;                         // This is |β|
                if Zk‘< b then
                        Zk := Zk‘;
                else
                        match S[r+1.. ] with S[b+1.. ] until q;
                        if q!=r+1 then
                                Zk := q-k; r := q-1; l := k;
                        else
                                Zk := Zk‘;
                end if;
        end if;
end for;
```

Komplexität

- Theorem
  - *Der Z-Box Algorithmus berechnet alle Z-Werte in O(|S|)*
- Beweis
  - Wir zählen a=„Anz. Matches“ und a‘=„Anz. Mismatches“
    - Erst a‘. Wie viele Mismatches gibt es pro k?
      - Induktionsanfang – Maximal einen
      - Fall 1: Maximal einen
      - Fall 2.1: 0; Es werden überhaupt keine Zeichen verglichen
      - Fall 2.2: Maximal einen
    - Also kann pro Position in S maximal ein Mismatch auftreten
    - Also gilt: $a' \leq |S|$

Komplexität 2

- Fortsetzung
  - Jetzt a. Wann führt der Algorithmus Matches aus?
    - Induktionsanfang – maximal $|S|-1$ Matches
    - Fall 1: Maximal $|S|-2$
    - Fall 2.1: Es werden keine Zeichen verglichen
    - Fall 2.2: Maximal $|S|-2$
  - Aber
    - Jeder Match verschiebt r
    - Wir vergleichen immer nur rechts von r
  - Also kann ein Zeichen von S höchstens einen Match erzeugen
  - Also gilt: $a \leq |S|$
- Also gilt: $a+a' \leq 2*|S| = O(|S|)$
- Qed.

Alles zusammen

- Z-Boxen kann man in $O(|S|)=O(m+n)$ berechnen
- Danach in $O(|S|)$ alle passenden Z-Boxen suchen
- Damit löst der Z-Box Algorithmus das exakte Stringmatchingproblem in $O(m+n)$

```
123456789012345678901
abxyabxz$xabxyabxyabxz
```

```
k‘ := k-l+1; b := r-k+1;
Zk := q-k; l := k; r := q-1;
```

k	Bemerkung	Z_k	l	r
2	Induktionsanfang	0	0	0
3	k>r; Neues Matching, 1 Mismatch	0	0	0
4	k>r; Neues Matching, 1 Mismatch	0	0	0
5	k>r; Neues Matching, 3 Matches, 1 Mismatch	3	5	7
6	$6 \leq 7$; k‘=2;b=2; Z_2=0; Also $Z_{k'}$<b, damit $Z_k=Z_{k'}$	0	5	7
7	$7 \leq 7$; k‘=3;b=1; Z_3=0; Also $Z_{k'}$<b, damit $Z_k=Z_{k'}$	0	5	7
8	8>7; Neues Matching, 1 Mismatch	0	5	7
9	9>7; Neues Matching, 1 Mismatch	0	5	7
10	10>7; Neues Matching, 1 Mismatch	0	5	7
11	11>7; Neues Matching, 7 Matches, 1 Mismatch	7	11	17
12	$12 \leq 17$;k‘=2;b=6,Z_2=0; $Z_{k'}$<b, damit $Z_k=Z_{k'}$	0	11	17
13	$13 \leq 17$,k‘=3;b=5;Z_3=0; $Z_{k'}$<b, damit $Z_k=Z_{k'}$	0	11	17
14	$14 \leq 17$,k‘=4;b=4;Z_4=0; $Z_{k'}$<b, damit $Z_k=Z_{k'}$	0	11	17
15	$15 \leq 17$;k‘=5;b=3;Z_5=3; Also $Z_{k'} \geq$b; matche S[18..] mit S[4..]; 5 Matches und Erfolg			

```
1234567890123456
aaaat$aaaaaaaaaa
```

```
k' := k-l+1; b := r-k+1;
```

```
Zk := q-k; l := k; r := q-1;
```

---

k	Bemerkung	Z_k	l	r
2	Induktionsanfang	3	2	4
3	k<r; k'=2;b=2;Z_2=3; $Z_{k'}$≥b; matche S[5..] mit S[3..];1 Mismatch;q=5	2	3	4
4	k≤ r; k'=3;b=1;Z_3=2; $Z_{k'}$≥b; matche S[5..] mit S[3..];1 Mismatch;q=5	1	4	4
5	k>r; Neues Matching, 1 Mismatch	0	4	4
6	k>r; Neues Matching, 1 Mismatch	0	4	4
7	k>r; Neues Matching, 4 Matches, 1 Mismatch	4	7	10
8	8 ≤ 10; k'=2;b=3;Z_2=3; $Z_{k'}$≥b; matche S[11..] mit S[4..];1 / 1;q=12	4	8	11
9	9 ≤ 11; k'=2;b=3;Z_2=3; $Z_{k'}$≥b; matche S[12..] mit S[4..];1 / 1;q=12	4	9	12
10	10 ≤ 12; ...	4	10	13
..	...	...	...	...

Inhalt dieser Vorlesung

- Naiver Algorithmus für exaktes Stringmatching
- Z-Box Algorithmus
- Berechnung von Z-Boxen
- Average-Case Komplexitäten

Komplexitäten des Z-Box Algorithmus

- Bisher haben wir nur den Worst-Case betrachtet
- Was ist die Average-Case Komplexität?
  - Auch O(m+n), weil die äußere Schleife S komplett durchläuft
    - Algorithmus ist $\Omega(|S|)$
  - Der Z-Box Algorithmus kennt keine „guten“ oder „schlechten“ Stringpaare

Naiver Algorithmus: Average-Case

```
1.  for i = 1..|T|-|P| do
2.    match := true;
3.    j := 1;
4.    while match
5.      if T[i+j-1]=P[j] then
6.        if j=|P| then
7.          print i;
8.          match := false;
9.        end if;
10.       j := j+1;
11.     else
12.       match := false,
13.     end if;
14.   end while;
15. end for;
```

- Worst-Case ist O(n*m)
  - Z.B. $T=a^m$; $P=a^n$
- Was ist der Average-Case?
  - Äußere Schleife wird immer m mal durchlaufen
  - Innere Schleife: Hängt von P bzw. T ab
- Annahme: Zufällige Strings über Σ
  - Test in L5 geht mit $p=1/|\Sigma|$ gut aus
  - Erwartete Zahl Vergleiche:
    - $1(1-p)+2*p^1(1-p)+\ldots+n*p^{n-1}(1-p)=$
      $1-p + 2p-2p^2 + \ldots n*p^{n-1}-n*p^n=$
      $1 + p + p^2 + \ldots + p^{n-1} - n*p^n =$

$$-np^n + \sum_{i=0}^{n-1} p^i$$

Beispiele

- Deutsche Texte: |T|=50.000, P=|8|, |Σ|=28
  - Worst-case: 400.000
  - Average-case: ~51.851
    - Mismatch nach durchschnittlich ~1,03 Vergleichen
  - Z-Box: ~50008
- DNA: |T|=50.000, P=|8|, |Σ|=4
  - Worst-case: 400.000
  - Average-case: 65.740
  - Mismatch nach durchschnittlich ~1,35 Vergleichen
  - Z-Box: ~50.008
- Vorsicht
  - Wir ignorieren konstante Faktoren
  - Sind deutsche Wörter / DNA Sequenzen zufällige Strings?

Fazit

- Z-Box Algorithmus
  - Berechnung der Z Werte für P$T in linearer Zeit
  - Danach alle Vorkommen von P in T in linearer Zeit
  - Komplexität O(m+n)
- Als Worst-Case ist das bereits optimal
- Folgende Verfahren
  - Boyer-Moore: Average Case sublinear
  - Knuth-Morris-Pratt: Elegant erweiterbar zu vielen P

Selbsttest

- Erklären Sie den Z-Box Algorithmus Schritt für Schritt
- Beweisen Sie die Komplexität des Z-Box Algorithmus
- Warum sind Average Case Analysen beim String-Matching fragwürdig?
- Was unterscheidet natürliche Sprache von zufälligen Strings?
- Wie könnte man das zum Stringmatching ausnutzen?