Bioinformatik

Knuth-Morris-Pratt Algorithmus
„Natürliche“ Erweiterung des naiven Matching

Ulf Leser

Wissensmanagement in der
Bioinformatik

Algorithmus Gerüst

1. Anordnung der Strings P und T
2. Matchen von rechts nach links
    3. Erster Vergleich ist T(n):P(n)
    4. Dann T(n-1):P(n-1), T(n-2):P(n-2), ... T(1):P(1)
3. Bei Mismatch oder Match für ganz P
    6. P um k Zeichen nach rechts verschieben
    7. Gehe zu 2
8. Bei Match
    9. Weiter matchen nach links
    10. Gehe zu 2

- Wie wird „k“ berechnet?
    - Bad Character Rule
    - Good Suffix Rule

Bad Character Rule

- Beobachtung
  - Wir vergleichen und haben gerade P(n) mit T(k) aligniert; $k \geq n$
  - Sei der erste Mismatch an Position i von P
    - Also nach n-i+1 Matches
  - Sei x das Zeichen an Position k-n+i in T
  - Welches sind Kandidaten für einen Match mit x in P?
    - Fall 1: x kommt in P nicht vor – verschiebe P um i Zeichen
      - Wir springen bis nach Position von x in T

```
T    xabxfabzzabxzzbzzb
P    abwxyabzz
```

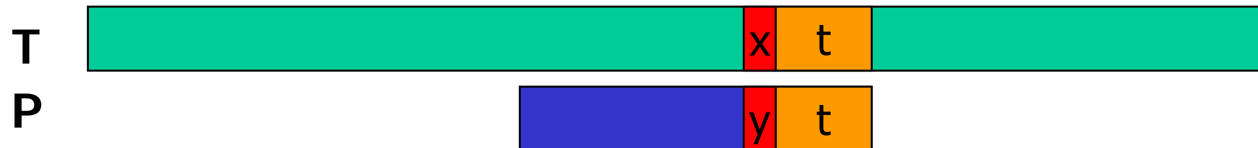
```
T    xabxfabzzabwzzbzzb
P          abwxyabzz
```

Wie weit können wir jetzt schieben ?

Zusammengenommen

- Definition:
  Gegeben Pattern P. Dann sei R(x), $x \in \Sigma$:
  - *0, wenn $x \notin P$*
  - *Sonst: Position des am weitesten rechts liegenden Auftretens von x in P*
- Berechnung leicht in O(n) möglich
  - Wie?
- Dann
  - Sei i die Position des ersten Mismatch in P
  - Sei x das Zeichen in T an der entsprechenden Position
  - Verschiebe P um max(1, i - R(x))
- Problem: Bei kleinem Alphabet (DNA) wird es meistens Auftreten von x rechts von i geben

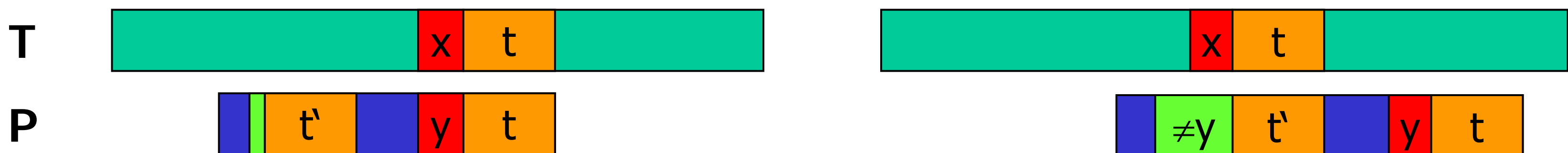
Good-Suffix Rule 2



- Substring t war ein Match, $x \neq y$ ein Mismatch
- Dann können wir wie folgt verschieben
  - Wenn t noch mal in P vorkommt, dann verschiebe bis zum am weitesten rechts liegenden t in P
  - Wenn t nicht mehr in P vorkommt, dann verschiebe P bis nach dem linken Ende von t in T

Fall 1

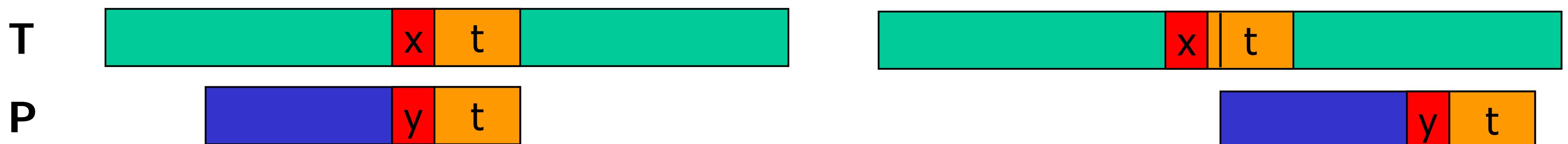
- Für die Mismatchposition i in P und t=P[n-i+1,..], sei k das rechte Ende des am weitesten rechts liegenden Vorkommen von t in P mit k<n und P(k-|t|) ≠ P(n-|t|) (‚y'), sonst 0
- Dann
  - Wenn k≠0: Verschiebe P um n-k



- Warum fordern wir nicht P(k-|t|)=‚x' ?

Fall 2

- Für die Mismatchposition i in P und t=P[n-i+1,..], sei k das rechte Ende des am weitesten rechts liegenden Vorkommen von t in P mit k<n und P(k-|t|) ≠ P(n-|t|) ('y'), sonst 0
- Dann
  - Wenn k≠0: Verschiebe P um n-k
  - Wenn k=0 und P≠t: Verschiebe P um n-|t|+1



- Man kann unter Umständen weiter verschieben
  - Später mehr

Anders gesagt

- Gesucht: Zu jedem Suffix von P suchen wir den nächsten (von rechts nach links) identischen Substring von P, den man nicht weiter nach links verlängern kann
- Erinnerung Z-Boxen: „… zu jedem Präfix von P alle identischen Substrings von P (von links nach rechts) …"

➢ Das Boxer-Moore Preprocessing ist also (fast) eine „invertierte" Z-Box Berechnung

Zwischenschritt

- Definition
  N(j) für Index j von P ist die Länge des längsten Suffix von P[1..j], das auch Suffix von P ist
- Beispiel

```
      dcabcabdabdab
N(j)=0002002005000
```

- j ist also eventuell das gesuchte rechte Ende
- Berechnung der N(j) Werte
  – N(j) symmetrisch zu Z_i Werten des Z-Box Algorithmus
  – Berechnung durch Z-Box Algorithmus auf umgedrehtem P (=P^r)

Zusammen: Preprocessing

- Gegeben: P
- Berechne N-Werte durch Z-Box auf P^r
- Berechne L`-Werte durch

```
for i=1 to n
        L`(i):= 0;
for j=1 to n-1
        i := n-N(j)+1;
        if (i≤n) then
                L`(i) := j;
end for;
```

N(j)=n-i+1

- Komplexität: O(n)
  - Z-Box ist O(n)
  - L`-Werte ist O(n)
  - Wir machen hier nur Preprocessing des Pattern; T bleibt unberührt

Vorarbeiten

- Erinnerung
  - N(j) ist die Länge des längsten Suffix von P[1..j], das auch Suffix von P ist
- Boyer-Moore Grundgerüst: Shift/compare Phasen
  - Eine Phase besteht aus Verschieben von P (shift) und matchen von rechts nach links bis Mismatch oder vollständiger Match (compare)
  - Dann Shift berechnen und nächste Phase starten, bis Ende von T erreicht
- Was wir noch brauchen
  - M ist ein Array int[m], initialisiert mit -1

M: Suffixe matchen Suffixe

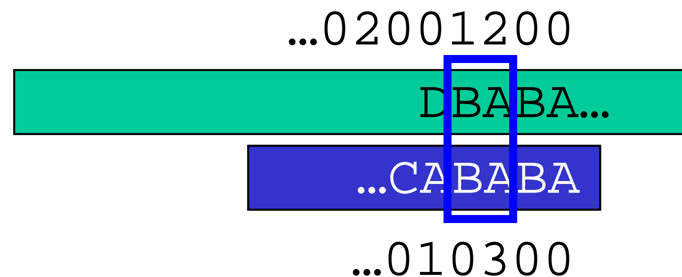
- M wird wie folgt benutzt
  - Wir beginnen eine Phase
  - Sei k das rechte Ende von P in T
  - Wir matchen nach links
  - Sei i die Position des (ersten) Mismatches in P
  - Dann matched das Suffix von P der Länge n-i mit einem Suffix des Substrings T[..k]
    - P[i+1..]=T[k-n+i..k]
  - Das merken wir uns: M[k] = x (Details später)
  - Später vergleichen wir M und N(i)
    - N(i) sind Suffixe von P in P
    - M sind Suffixe von P in T
    - Da wir immer nach links vergleichen, aber nach rechts schieben, kennen wir in einer Phase an Position k schon alle M[i] Werte mit i<k

Details – 1

- Wir beginnen eine Phase an Position k in T
  - Wir schieben immer wie beim „normalen“ Boyer-Moore
  - Wir sparen nur Vergleiche in jeder Phase
- Wir laufen nach links
  - Mit Variablen h durch T und i durch P
- Fall 1
  - Wenn M(h)=-1 oder M(h)=N(i)=0, dann
    - T(h)=P(i) und i=1: Das ist ein vollständiger Match; beende Phase
    - T(h)=P(i) und i>1: Weiter nach links matchen (h--; i--)
    - T(h)≠P(i): Kein Match; Setze M(k)=k-h; beende Phase

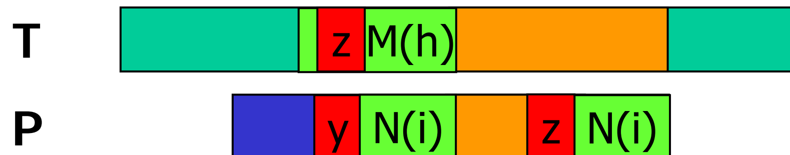
Details – 2

- Wenn M(h) < N(i)
  - Also matched P mit T ab h bis h-M(h)+1
    - Das genau haben wir uns in M(h) gemerkt
  - Diese Matches sparen wir uns
  - h := h-M(h); i := i-M(h)
  - Dann weiter nach links matchen
    - Phase läuft weiter



Details – 4

- Wenn M(h) > N(i) und N(i) < i
  - Wir können wieder Matches überspringen
    - Denn P[i-N(i)+1..] matched mit dem Substring in T, der an Position h endet
  - Danach kommt garantiert ein Mismatch
    - Denn wegen M(h) kommt ein Zeichen, das mit der Verlängerung des Suffixes von P matched
    - Aber wegen N(i) können wir den Substring in P, der an i endet, nicht weiter nach links mit Suffix von P matchen
  - Setze M(k) := k-h+N(i)
    - Gusfield setzt auf k-h (kürzer ist nie schlimm) wg Beweis
  - Phase beenden (mit Mismatch)



Inhalt dieser Vorlesung

- Knuth-Morris-Pratt
- Komplexität
- Vergleich mit anderen Algorithmen

Warum noch ein String Algorithmus?

- Klassischer Algorithmus
- Intuitive Erweiterung des naiven Algorithmus
- Schöner Beweis der Korrektheit
- Erweiterung zum linearen Matching mit mehrerer Pattern – Aho-Corasick

Grundidee

- Erinnerung an den naiven Algorithmus

```
ctgagatcgcgta

gagatc
 gagatc
  gagatc
   gagatc
    gagatc
     gatatc
      gatatc
       gatatc
```

```
abcxabcgedsc

abcxabcde
```

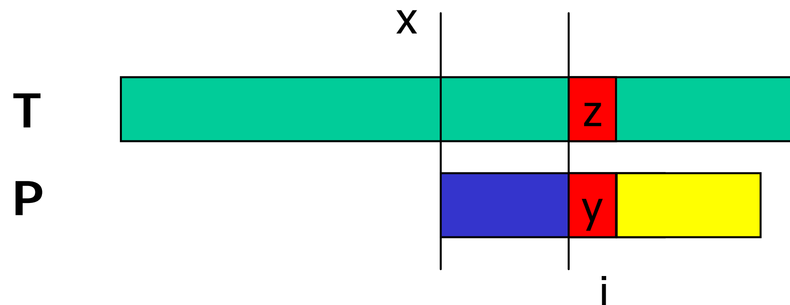
Wie weit kann man jetzt sicher springen ?
- T beginnt mit abcxabc
- Das zweite „a" in P kommt an Position 5

Also: 4 Zeichen (mind.) schieben

- Idee: Ausnutzen von
  - Wissen über die gematchten Zeichen benutzen
  - Preprocessing von P

Preprocessing

- Was wollen wir wissen?
  - Sei P mit T an Startposition k aligniert.
    Bei einem Mismatch an Position i (in P) gilt:
    T[k..k+i-2] = P[1..i-1]



  - Wir suchen nach einem möglichst großen Shift
  - Wir untersuchen das schon gematchte Präfix von P
  - Kommt der Anfang von P noch mal in P[1 .. i-1] vor?
  - Wenn ja – schiebe bis dahin
    - mehrmaliges Vorkommen später
  - Wenn nein – schiebe um i-1
    - den Teil von T kennen wir noch nicht

Definitionen

- Definition
  - *Sei sp_i die Länge des längsten echten Suffix von P[1..i], das mit einem Präfix von P matched*
  - *Sei sp_i' die Länge des längsten echten Suffix von P[1..i], das mit einem Präfix von P matched und für das gilt: $P(i+1) \neq P(sp_i+1)$; sonst 0*
- Beispiel

```
P:      abcaeabcabd
sp_i:   00010123420
        abca
        abcaeab
        abcaeabc
        abcaeabca
```

```
P:      abcaeabcabd
sp_i:   00010123420
sp_i':  00010000420

        abcaeabc
        abcaeabcab
```

KMP im Überblick

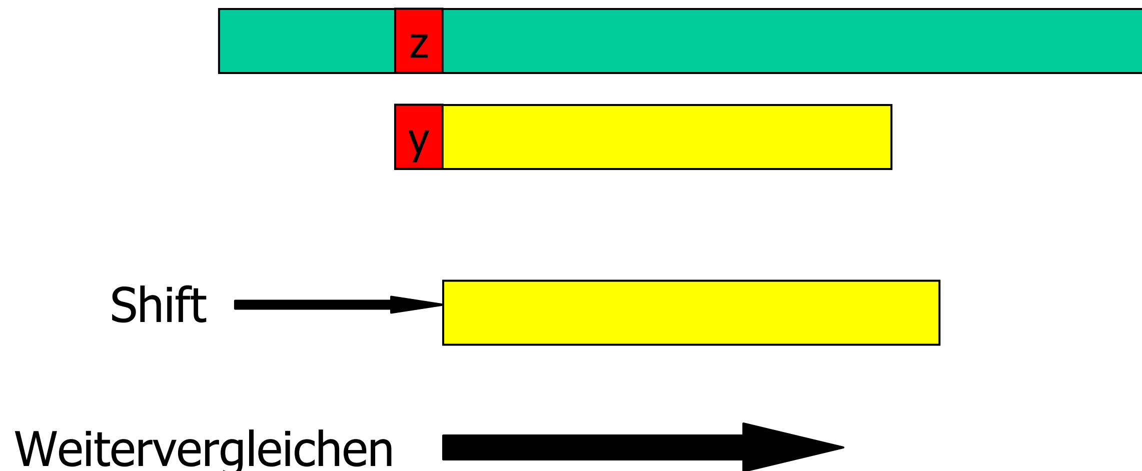
- Phasen
  - Preprocessing von P: Berechne sp_i und sp_i`
  - Matching von links nach rechts wie in naivem Algorithmus
  - Bei Match
    - Nächstes Zeichen matchen
  - Bei Mismatch (an Position i in P)
    - Schiebe P um ... (Länge durch sp_i` und i bedingt - später)
    - Weitervergleichen ab ? (später)
  - Bei vollem Match (endet an Position i=n)
    - Schiebe P um ... (Länge durch sp_n` und n bedingt)
    - Weitervergleichen ab ? (später)
- Zwei Gewinne gegenüber naivem Matching
  - Schiebe immer um mindestens 1 Position, oft aber mehr
  - Vergleiche starten „meistens" bei i+1 (und nicht bei 1)

KMP im Überblick

- Offene Punkte
  - Bestimmung der Länge der Verschiebung
  - Korrektheit des Algorithmus
    - Schiebt man nicht zu weit?
  - Komplexität der Suchphase
  - Komplexität des Preprocessing

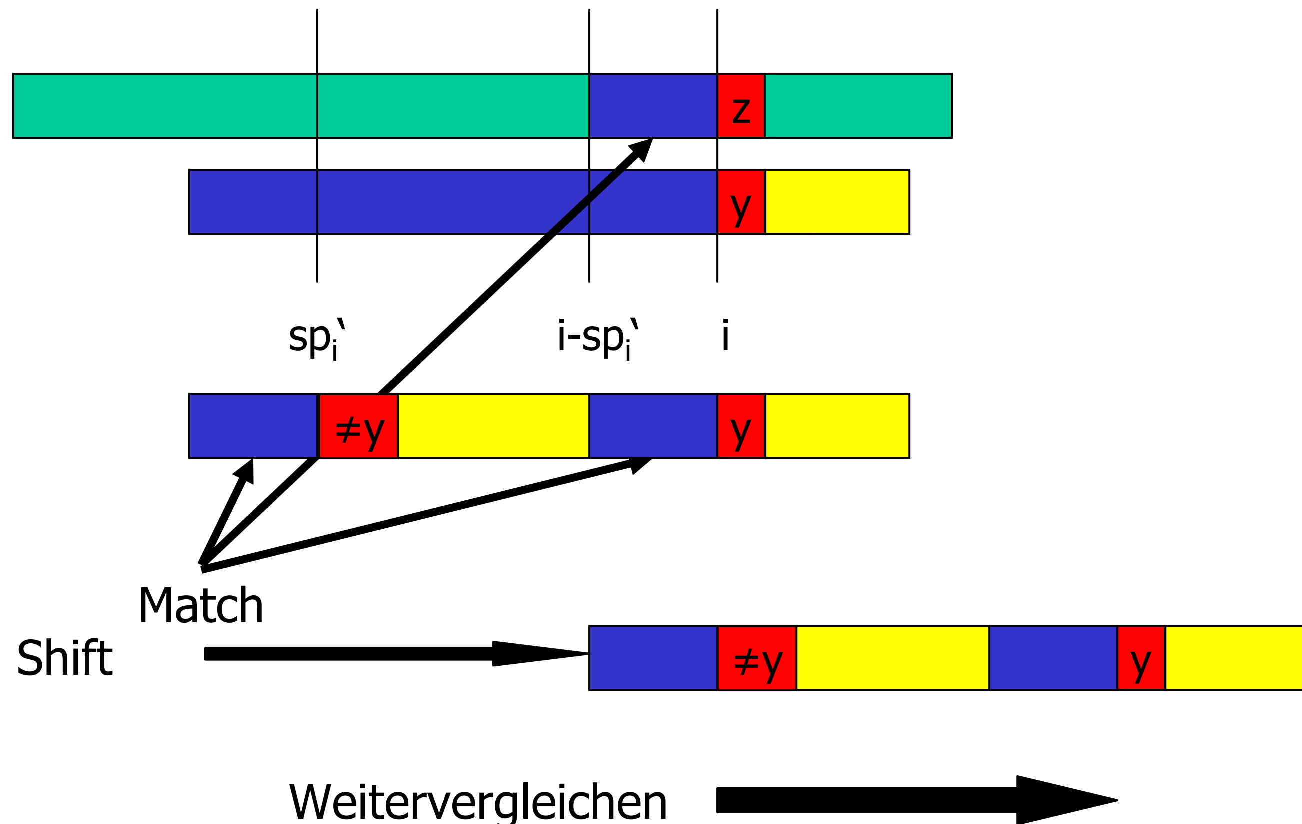
Shift Regel 1

- Wenn P(1) und T(k) sofort mismatchen
  - Schiebe P um 1 Positionen nach rechts
  - Vergleiche weiter ab P(1)



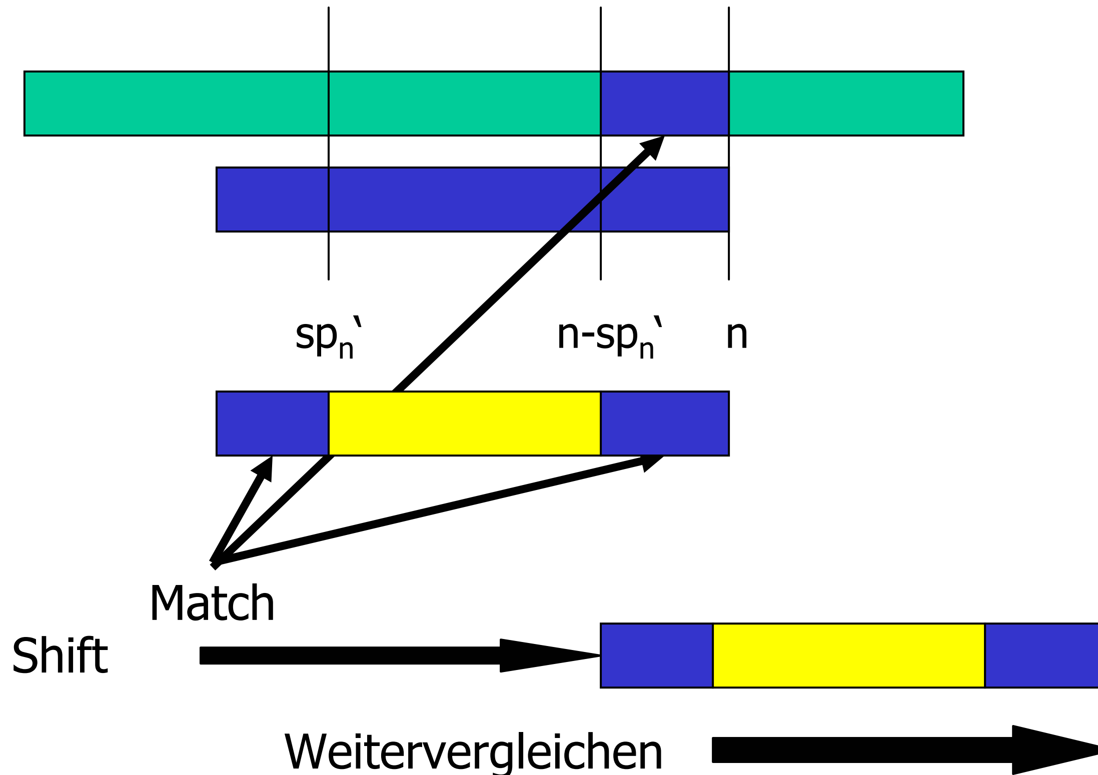
Shift Regel 2

- Wenn bei i+1 in P der erste Mismatch vorkommt
  - Schiebe P um i-sp_i\` Positionen nach rechts
  - Vergleiche weiter ab P(sp_i\`+1)



Shift Regel 3

- Wenn ein kompletter Match gefunden wird
  - Schiebe P um n-sp_n` Positionen nach rechts
  - Vergleiche weiter ab P(sp_n`+1)

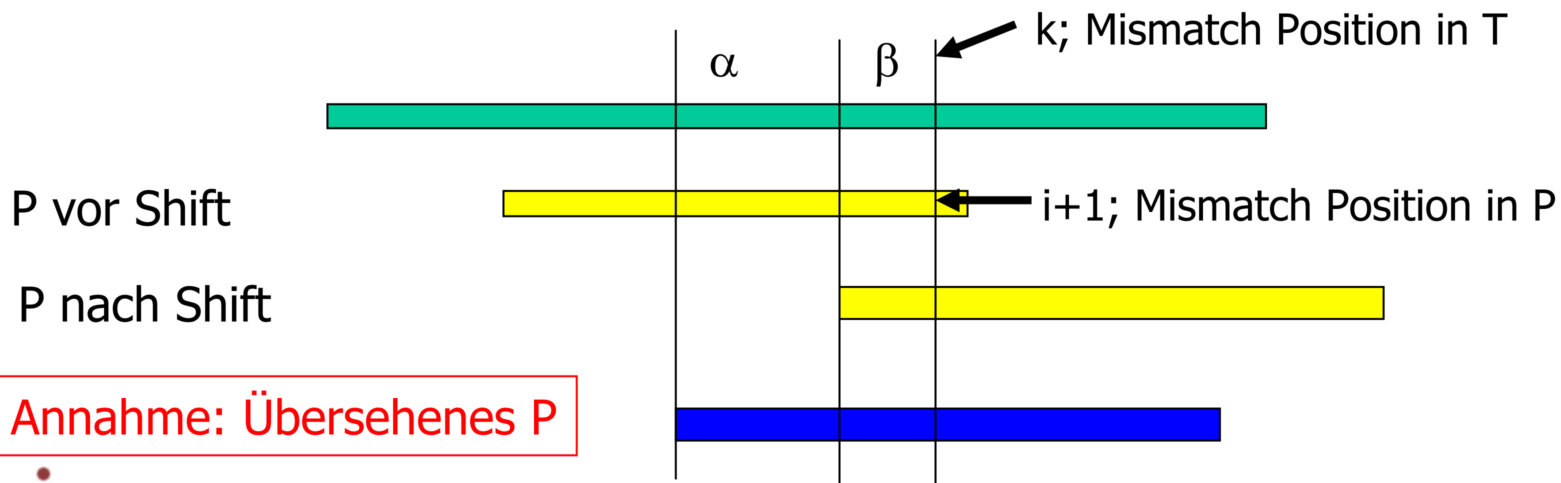


Warum sp_i` (und nicht sp_i)?

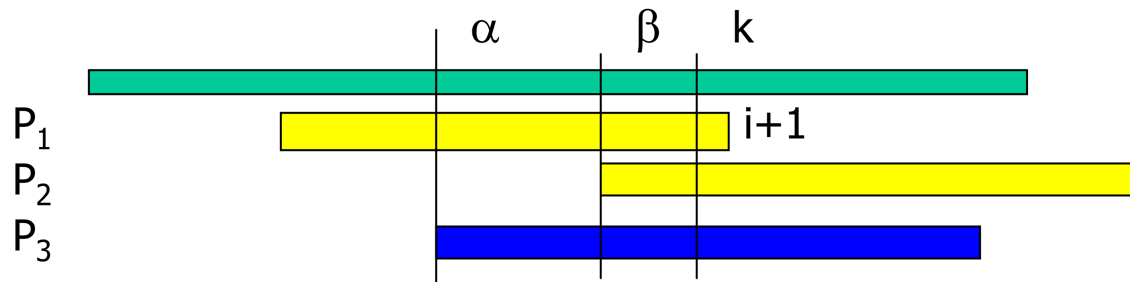
- Beide machen keinen Fehler (Beweis später)
- Beobachtung
  - Es gilt: sp_i` <= sp_i
    - Präfixe, die sp_i genügen, sind mindestens solange wie Präfixe für sp_i`
  - Man schiebt um i - sp_i(`)
  - Also: sp_i` erlaubt längere Verschiebungen als sp_i
- Was spart man?
  - Sei $sp_i \neq sp_i$` für ein i und bei i+1 tritt Mismatch auf
    - Sei P gerade mit Position k in T aligniert
  - Dann ist $P(i+1) \neq T(k+i+1)$
  - Wegen $sp_i \neq sp_i$` gilt: $P(i+1) = P(sp_i+1)$
  - Der nächste Vergleich ($P(sp_i+1)$ mit $T(k+i+1)$) ist also überflüssig
  - Außerdem: Oft ist sp_i`$=0 \neq sp_i$; dann kann man mit $sp_i$` weiter schieben

Korrektheit der Shift-Regel

- Schiebt man nicht eventuell zu weit?
- Theorem
  - *Die Shift-Regel verschiebt nie soweit, dass ein Vorkommen von P in T übersehen wird*
- Beweis
  - Wenn das anders wäre, hätten wir:

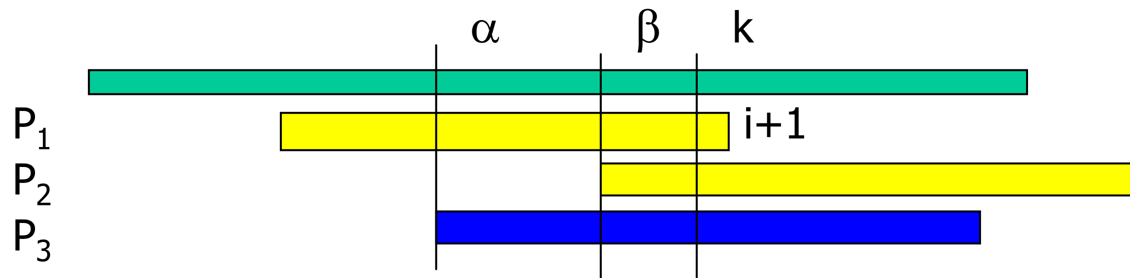


Beweis 2



- Wir zeigen, dass diese Situation im Widerspruch zur Definition von sp_i‘ steht
  - β ist Präfix von P; |β|= sp_i‘ (Def. sp_i‘)
  - P_1 und P_3 matchen mit T bis k-1; also ist
    - αβ Suffix von P(1..i) (in P_1)
    - αβ Präfix von P (in P_3)
  - αβ erfüllt die Bedingung für ein sp_i‘
    - $P(i+1) \neq P(|\alpha\beta|+1)$ (sonst kein Mismatch in Position k)

Beweis 3



- Weiterhin muss gelten $|\alpha|>0$
  - sonst ist $P_2=P_3$ und wir haben nichts übersehen
- Damit
  - $\alpha\beta$ ist Präfix von P
  - $\alpha\beta$ ist Suffix von P(1..i)
  - $P(i+1) \neq P(|\alpha\beta|+1)$
  - $|\alpha\beta| > |\beta| = sp_i`$
- $\alpha\beta$ erfüllt alle Voraussetzungen für $sp_i`$, ist aber länger
- Widerspruch zur Definition von $sp_i`$
- Also kann dieser Fall nicht eintreten
- qed.

Komplexität von KMP

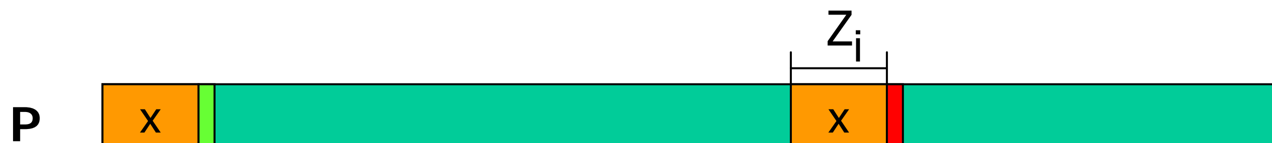
- Theorem
  - *KMP benötigt höchstens 2m Zeichenvergleiche*
- Beweis
  - Jeder Vergleich beginnt entweder
    - Am letzten Zeichen des letzten Vergleichs (bei Mismatch)
    - Am Zeichen rechts vom letzten Zeichen des letzten Vergleichs (bei vollständigem Match)
  - Jede Shift/Vergleich Phase untersucht also ein Zeichen höchstens zwei Mal
  - In jedem Schritt wird P mindestens um 1 Position verschoben
  - Also gibt es höchstens m Shift/Vergleich-Phasen
  - Also sind insgesamt höchstens 2m Zeichenvergleiche notwendig
  - qed.

Bis jetzt haben wir

- KMP ist korrekt
- KMP ist linear in der Suchphase O(m)
- Jetzt: Wie teuer ist das Preprocessing?
  - Berechnung der sp_i` Werte

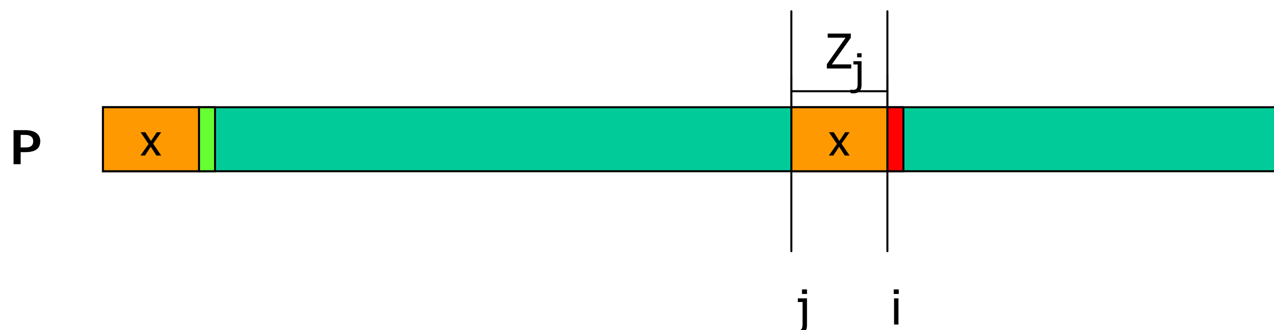
Preprocessing

- Wir führen das Preprocessing auf Z-Boxen zurück
- Erinnerung (Z-Box)
  - *Sei i>1. Dann ist Z_i die Länge des größten Substrings x in P mit*
    - *x=P[i..i+|x|-1]* *(x startet an Position i in P)*
    - *P[i..i+|x|-1] = P[1..|x|-1]* *(x ist auch Präfix von P)*
  - *x heißt Z-Box von P an Position i mit Länge $Z_i(P)$*



Verwendung der Z-Boxen

- Wir suchen echte Suffixe von P[1..i], die auch Präfixe von P sind
  - ... und sich nicht verlängern lassen (sp_i‘)
- So ein Suffix muss die Z-Box an Pos $j=i-sp_i`+1$ sein



Formal

- Theorem
  - *Für i>1 sei j>1 die am weitesten links stehende Position in P für die gilt:* $i=j+Z_j-1$
  - *Wenn j existiert, dann ist* $sp_i'=Z_j=i-j+1$
  - *Sonst* $sp_i'=0$
- Beweis
  - Wenn j existiert, ist Z_j ein längstes Suffix von P[1..i], das auch Präfix von P ist
    - Sonst wären Z-Boxen falsch berechnet
  - Wenn j nicht existiert, matched kein Suffix von P[1..i] mit einem Präfix von P
  - qed.

Berechnung

```
for i = 1 to n                  // Initialisierung
        sp_i` := 0;
end for;
for j = n downto 2              // Spätere (weiter links) Treffer
        i := j + Z_j - 1;       // überschreiben frühere
        sp_i` := Z_j;
end for;
```

- Damit
  - Berechnung Z-Boxen ist O(n)
  - Berechnung sp_i` ist O(n)
  - KMP Shift/Compare ist O(m)

➢ KMP ist O(m+n)

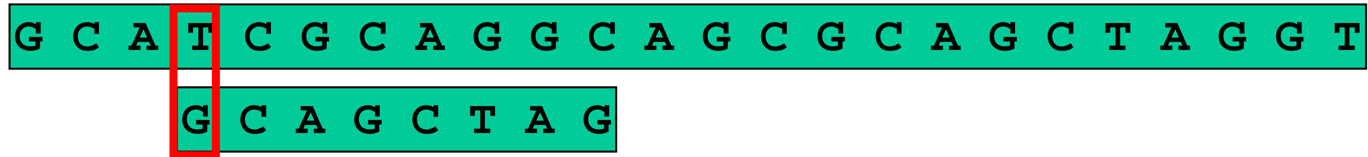
Shift-Regel:
Mismatch bei i+1
Schiebe um $i-sp_i'$
Vergleiche ab $sp_i'+1$

Beispiel

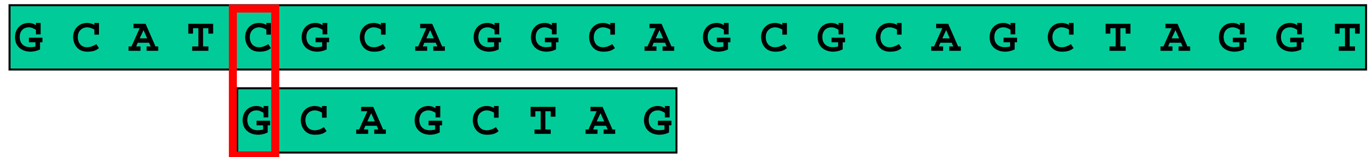
Pos:	1	2	3	4	5	6	7	8
P:	G	C	A	G	C	T	A	G
sp_i':	0	0	0	0	2	0	0	1

Schiebe um 3-0=3
Vergleiche ab 0+1

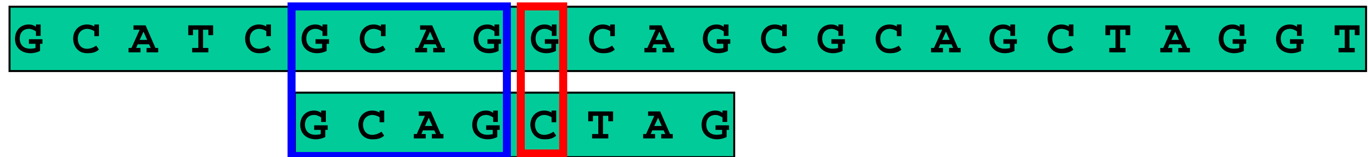
Schiebe um 1
Vergleiche ab 1



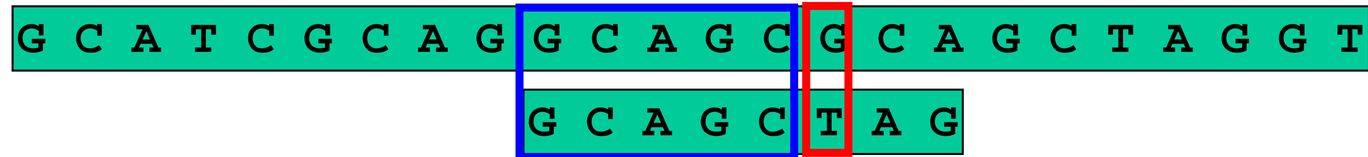
Schiebe um 1
Vergleiche ab 1



Schiebe um 4-0=4
Vergleiche ab 0+1



Schiebe um 5-2=3
Vergleiche ab 2+1

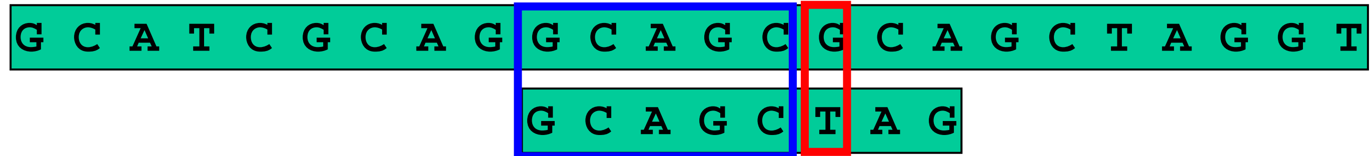


Shift-Regel:
Mismatch bei i+1
Schiebe um i-sp_i'
Vergleiche ab sp_i'+1

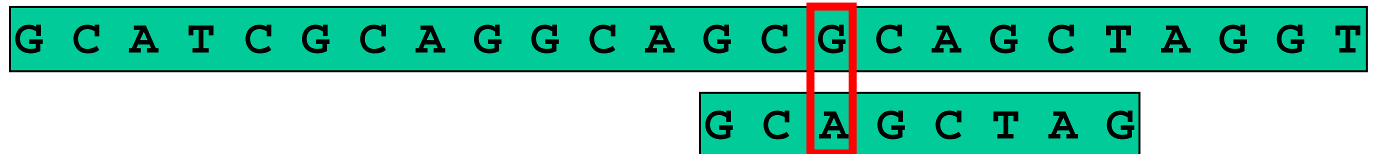
Beispiel 2

Pos:	1	2	3	4	5	6	7	8
P:	G	C	A	G	C	T	A	G
sp_i':	0	0	0	0	2	0	0	1

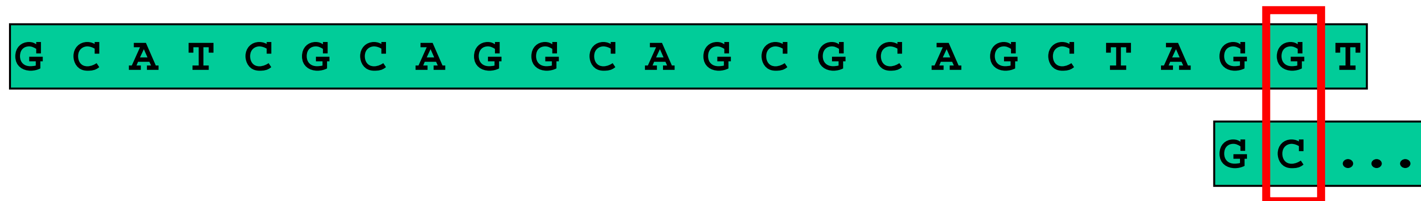
Schiebe um 5-2=3
Vergleiche ab 2+1



Schiebe um 2-0=2
Vergleiche ab 0+1



Schiebe um 8-1=7
Vergleiche ab 1+1



Kompletter KMP Algorithmus

```
compute sp_i`;
h := 1;                     // Next comparison in T
i := 1;                     // Next comparison in P
while h+(n-i)<=m do
       while P(i)=T(h) and i<=n do
              i++;
              h++;
       end while;
       if i=n+1 then               // Match
              print h-n;
       else                        // Mismatch at h / p
              if i=1 then
                     i++;   // First comp. fails - move 1 pos
                     h++;
       end if;
       // Comparison in T will continue at position h
       // Comparison in P will continue after shift
       i:=sp_{i-1}` +1;            // i is mismatch pos. in P
end while;
```

Vergleich

	Z-Box	Boyer-Moore (Apostolico-Giancarlo)	Knuth-Morris-Pratt
Komp. Preproc. **Komp. Suche** **Komp. Gesamt**	O(m+n) O(m) O(m+n)	O(n) O(m) O(m+n)	O(n) O(m) O(m+n)
Größe Alphabet	• Praktisch unabhängig von Alphabetgröße	• Je größer, desto besser – BCR führt zu großen Sprüngen • BCR greift nicht bei kleinen Alphabeten	• Praktisch unabhängig von Alphabetgröße
Bemerkung	• Avg und Worst Case Komplexität gleich - es wird jedes Zeichen mind. einmal verglichen	• WC der einfachen Variante bei Wiederholungen (z.B: a^n in a^m) •Best Case ist O(m/n) [Suche a^n in b^m]	• Erweiterbar auf mehrere Pattern