Bioinformatik

Knuth-Morris-Pratt Algorithmus

„Natürliche“ Erweiterung des naiven Matching

Ulf Leser

Wissensmanagement in der Bioinformatik

Boyer-Moore 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 X Zeichen nach rechts verschieben
   7. Gehe zu 2
8. Bei Match
   9. Weiter matchen nach links
   10. Gehe zu 2

- Das ist noch langweilig
- Wann kann man P um mehr als 1 Schritt nach rechts schieben?
  - Bad Character Rule
  - Good Suffix Rule

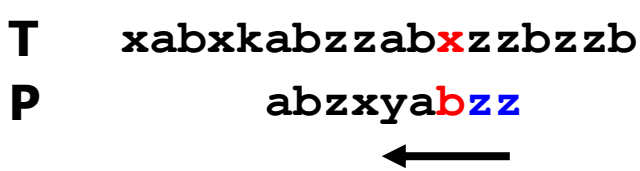
Bad Character Rule

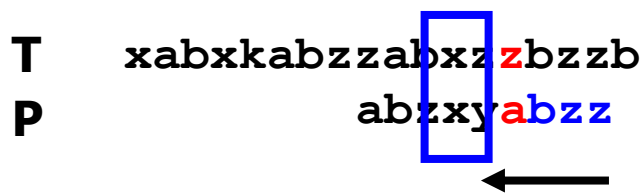
- Beobachtung
  - Wir befinden uns im Algorithmus und haben P(n) mit T(k) aligniert; k≥n
  - Sei der erste Mismatch an Position n-i von P
  - Sei x das Zeichen an Position k-i in T
  - Welches sind Kandidaten für einen Match mit x in P?
    - Fall1: x kommt in P nicht vor – verschiebe P um n-i Zeichen

Wie weit können wir jetzt schieben ?

Bad Character Rule 2

- Beobachtung
  - Wir befinden uns im Algorithmus und haben P(n) mit T(k) aligniert
  - Sei der erste Mismatch an Position n-i von P
  - Sei x das Zeichen an Position k-i in T
  - Sei l das letzte Vorkommen von x in P (also das am weitesten rechts liegende)
  - Welches sind Kandidaten für einen Match mit x in P?
    - Fall1: x kommt in P nicht vor – verschiebe P um n-i Zeichen
    - Fall2: l<i. Also kommt x in P nur vor i vor – verschiebe P um i-l Zeichen

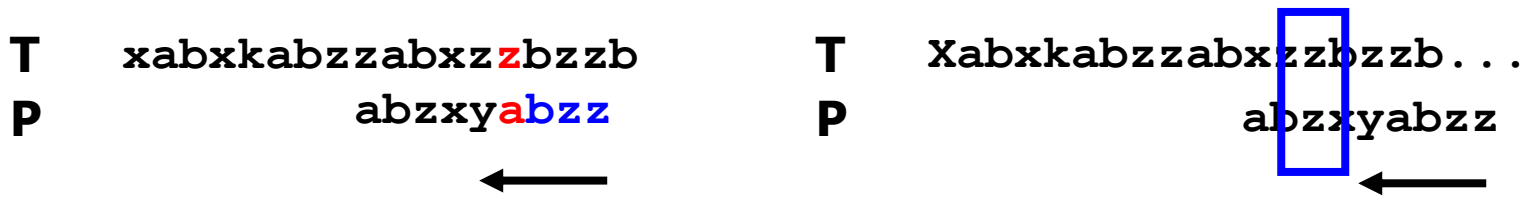




Wie weit können wir jetzt schieben ?

Extended Bad Character Rule

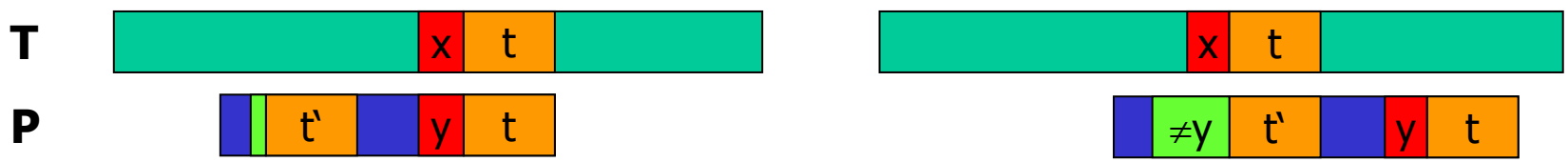
- Beobachtung
  - Die „x" rechts von i sind uninteressant – hier wurde schon geprüft
  - Also: Verschiebe zum nächsten, links von i liegenden x



- Benötigt Berechnung der relativen Positionen
  - Array [n,|Σ|] => linearer Lookup, aber Platzverbrauch O(n*|Σ|)
  - Listen für jedes Zeichen mit Positionen; linearer Platz O(n), aber nicht-linearer Lookup (Binsearch in Liste)
- Teurer als Berechnung bei einfacher Bad Character Rule
- BCR oder EBCR ist anwendungsabhängig

Good suffix rule: Fall 1

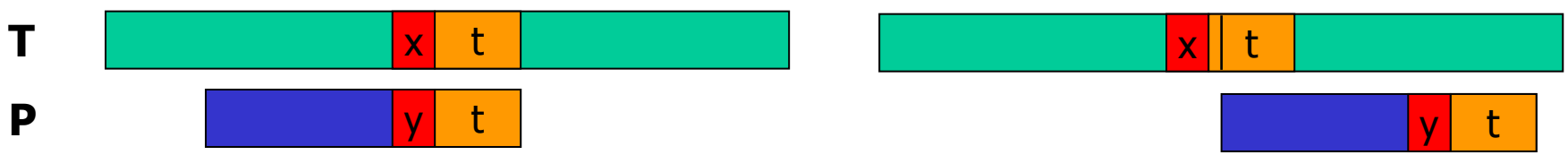
- Für die aktuelle Mismatchposition i (in P) und t=P[n-i+1,...], sei k das rechte Ende des weitesten rechts liegenden Vorkommen von t in P mit k<i und S(k-|t|) ≠ S(n-|t|) (‚y‘), sonst 0
- Dann
  - Wenn k≠0: Verschiebe P um n-k



- Warum fordern wir nicht S(k-|t|)=‚x‘ ?

Fall 2

- Für die aktuelle Mismatchposition i (in P) und t=P[n-i+1,...], sei k das rechte Ende des weitesten rechts liegenden Vorkommen von t in P mit k<i und S(k-|t|)≠S(n-|t|), 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 geringfügig weiter verschieben
  - ... soweit, bis Präfix von P mit Suffix von t macht
  - Später mehr

Preprocessing

- Gesucht: Zu jedem Suffix von P suchen wir den nächsten (von rechts nach links) identischen Substring von P
- Erinnerung Z-Boxen: ...zu jedem Präfix von P alle identischen Substring von P (von links nach rechts)

➢ Das Boxer-Moore Preprocessing ist also (fast) ein „invertiertes" Z-Box Preprocessing

Ableitung von L‘(i) aus N(j)

- N(j) sind die Z-Boxen auf dem umgedrehten String
- N(j) Werten geben die Länge von längsten Suffixen an, die links von j in P vorkommen
- L‘(i) sucht das am weitesten rechts liegende Auftreten von Suffixen der Länge n-i+1
  - i gibt die Länge des Suffixes vor
  - Suffix darf sich nicht verlängern lassen
- Damit ist L‘(i) der größte Wert j mit N(j)=n-i+1
  - Alle Positionen mit N(j)=n-i+1 stehen für (längste) Suffix der gewünschten Länge
  - Davon interessiert uns das am weitesten rechts liegende, also für das größte j

Zusammen: Preprocessing

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

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

- Komplexität: O(n)
  - Z-Box ist O(n)
  - L‘-Werte ist O(n)

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
```

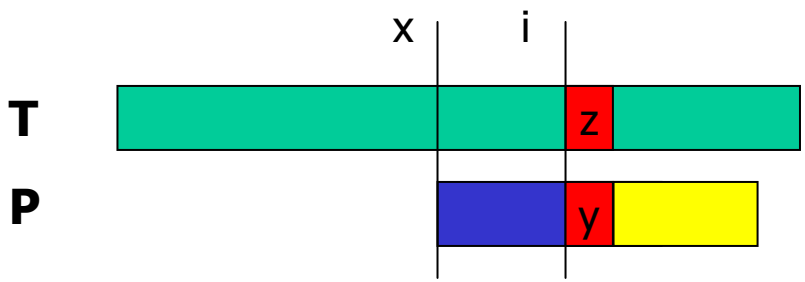
Wie weit kann man jetzt sicher springen ?
- T beginnt mit abcxabc
- Das zweite „a" in P kommt an Position 5

Also: 4 Zeichen (mind.) schieben

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

Preprocessing

- Was wollen wir wissen?
  - Bei einem Mismatch an Position i (in P) gilt:
    $$T[x \,..\, x+i-1] = P[1 \,..\, i-1]$$



  - Wir suchen nach einem möglichst großen Shift
  - Wir vergleichen das (schon gematchte) Präfix von P mit dem Rest
  - 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
spi:    00010123420
        abca
        abcaeab
        abcaeabc
        abcaeabca
```

```
P:      abcaeabcabd
spi:    00010123420
spi':   00010000420
        abcaeabc
        abcaeabcab
```

KMP im Überblick

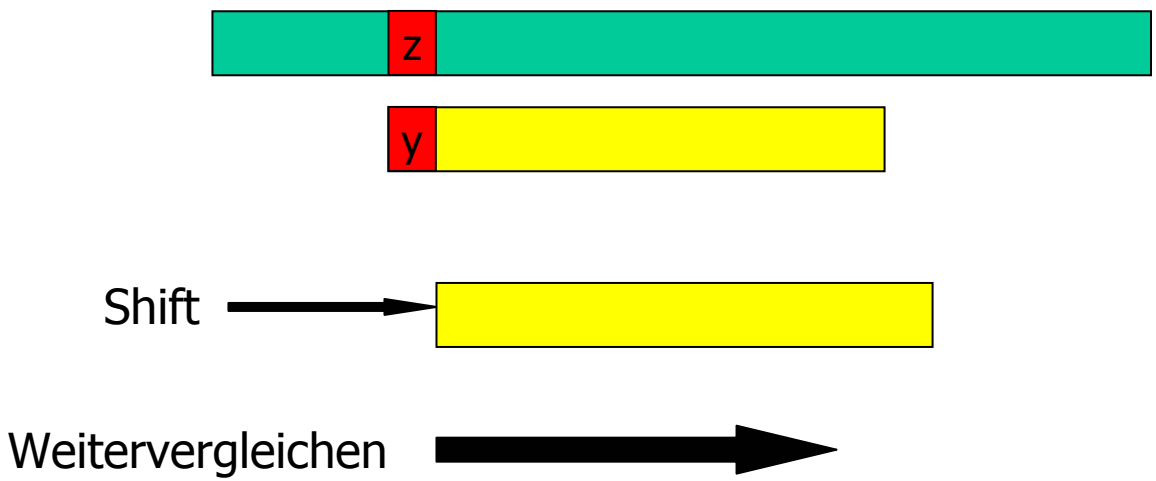
- Phasen
  - Preprocessing von P: Berechne sp_i und sp_i`
  - Matching von links nach rechts wie in naivem Alg.
  - Bei Match
    - Nächstes Zeichen matchen
  - Bei Mismatch (an Position i in P)
    - Schiebe P um ... (Länge durch sp_i` und i bedingt - später)
    - Weitervergleichen ab i+1 oder i (später)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

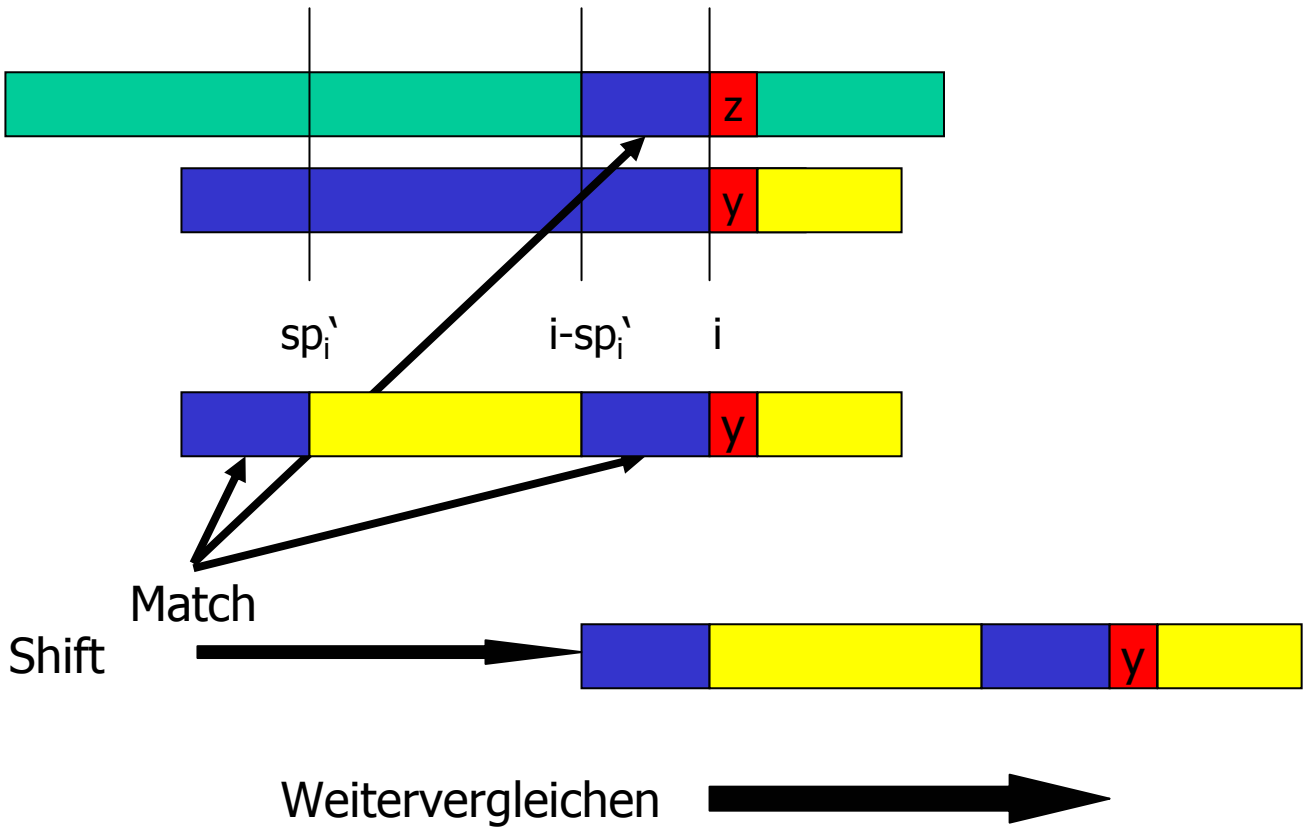
Shift Regel 1

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



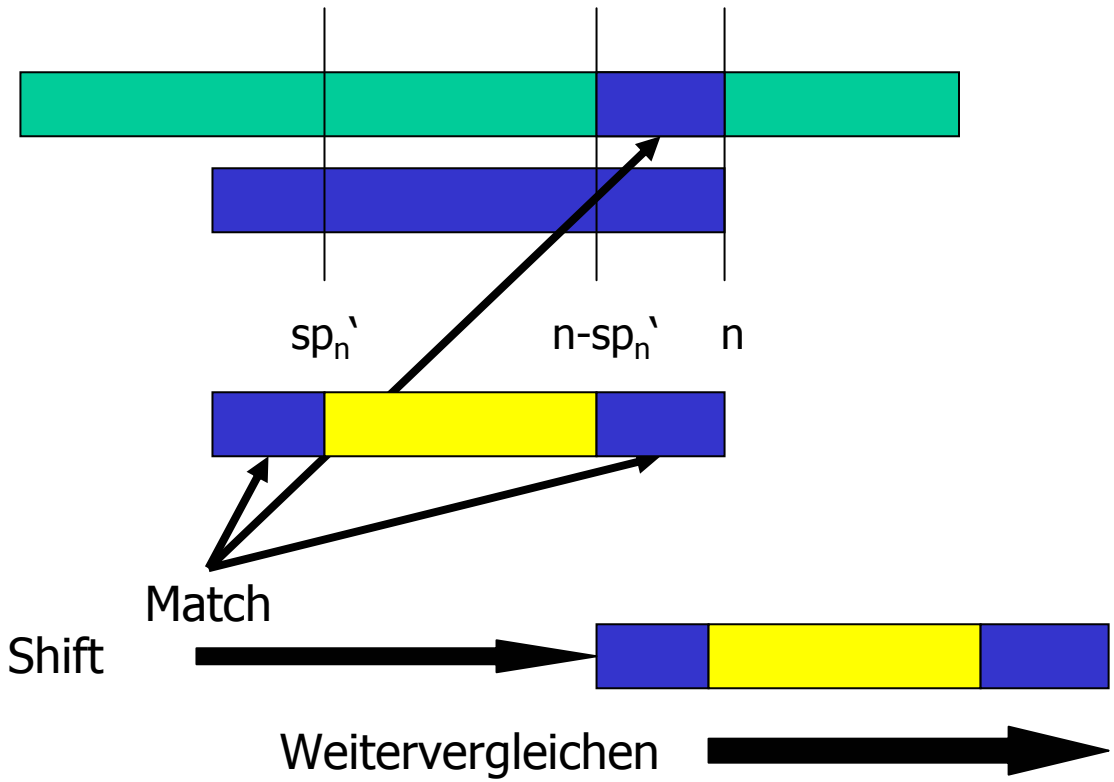
Shift Regel 2

- 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]

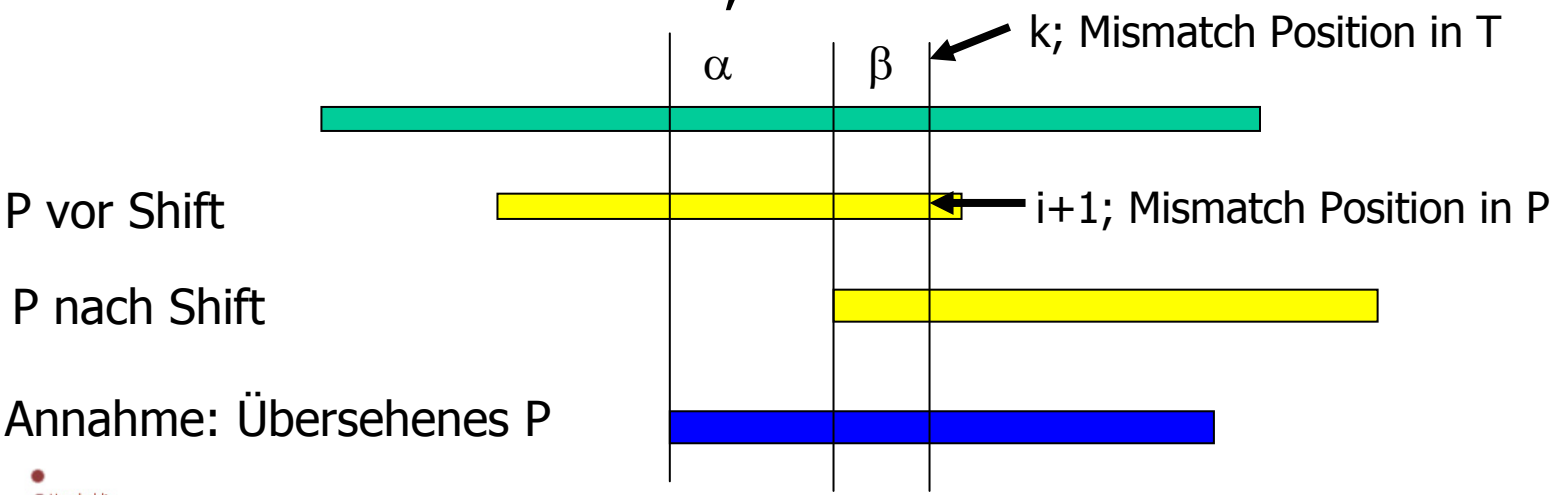


Warum sp_i‘ (und nicht sp_i)?

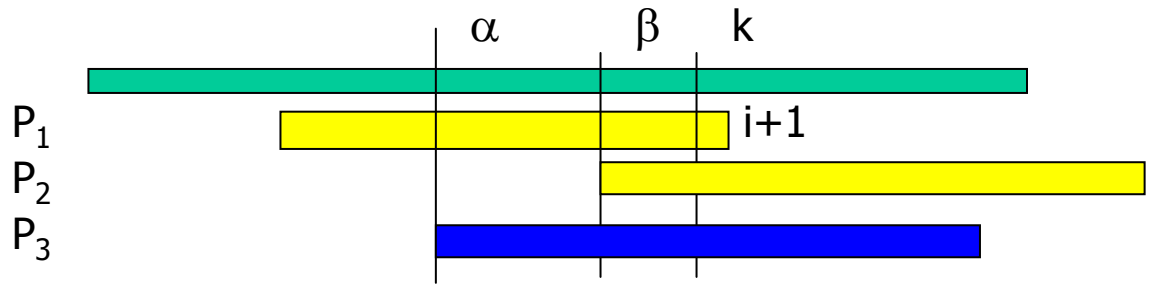
- Beide machen keinen Fehler (Beweis später)
- Beobachtung
  - Es gilt: sp_i‘ <= sp_i
    - Präfixe, die sp_i genügen, sind mindestens solange wie Präfixe für sp_i‘
  - Man schiebt um i - sp_i(‘)
  - Also: sp_i‘ erlaubt längere Verschiebungen als sp_i
- Was spart man?
  - Sei $sp_i \neq sp_i$‘ für ein i und bei i+1 tritt Mismatch auf
    - Sei P gerade mit Position x in T aligniert
  - Dann ist P(i+1) ≠ T(x+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(x+i+1)) ist also überflüssig
  - Außerdem: Oft ist sp_i‘=0≠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 dem so wäre, hätten wir:

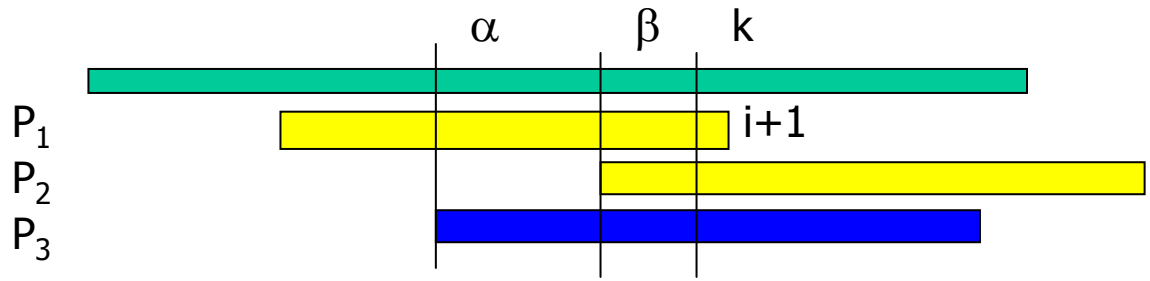


Beweis 2



- Wir zeigen, dass diese Situation im Widerspruch zur Definition von sp_i' steht
  - β ist Präfix von P; $|\beta| = sp_i'$ (Def. sp_i')
  - P_1 und P_3 matchen mit T bis k-1; also ist
    - $\alpha\beta$ Suffix von P[1..i] (in P_1)
    - $\alpha\beta$ Präfix von P (in P_3)
  - $\alpha\beta$ 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 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

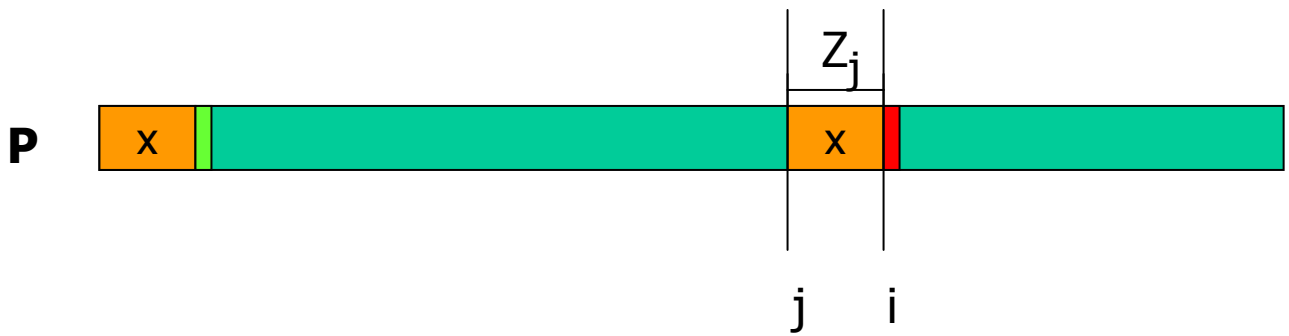
- KMP ist korrekt
- KMP ist linear in der Suchphase O(m)
- Jetzt: Wie teuer ist das Preprocessing?
  - Berechnung der sp_i' Werte

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|]* *(x startet an Position i in P)*
    - *P[i..i+|x|] = P[1..|x|]* *(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 von Position $j=i-sp_i'$ 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 maximal großes Suffix von P[1..i]; sonst wären Z-Boxen falsch berechnet
  - Wenn j nicht existiert, matched kein Suffix von P[1..i] ein 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;
```

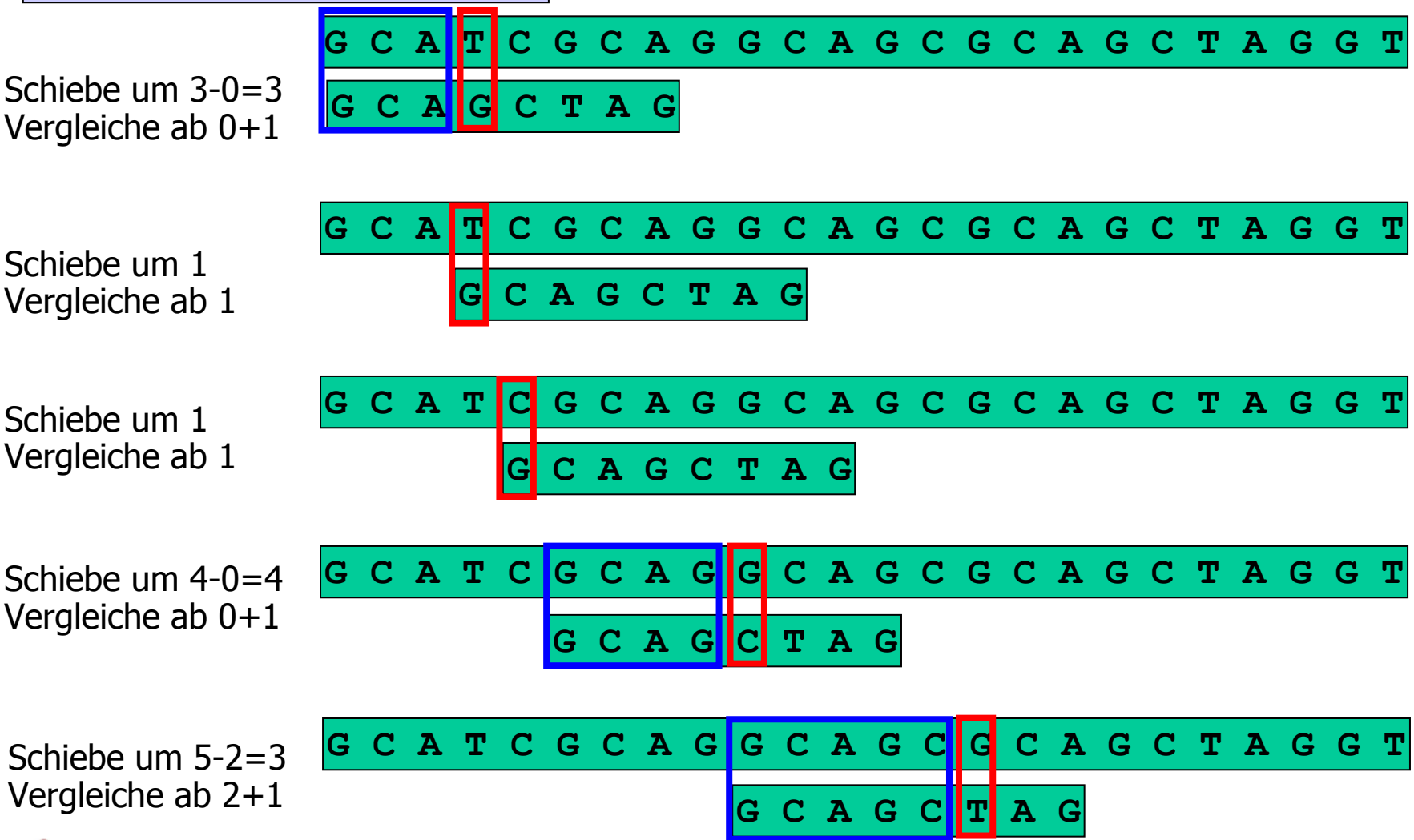
- Damit
  - Berechnung Z-Boxen ist O(n)
  - Berechnung sp_i' ist O(n)
  - KMP Shift/Compare ist O(m)

➢ 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

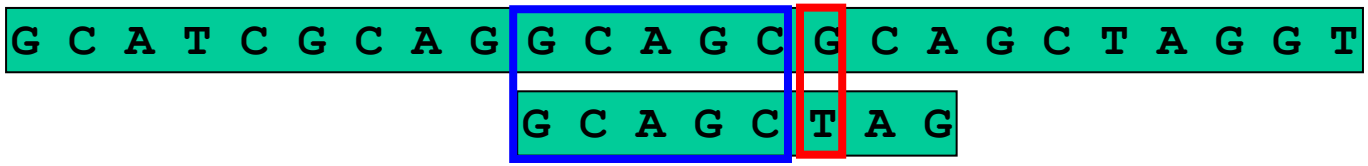


Shift-Regel:
Mismatch bei i+1
  Schiebe um i-sp_i`
  Vergleiche ab sp_i`+1

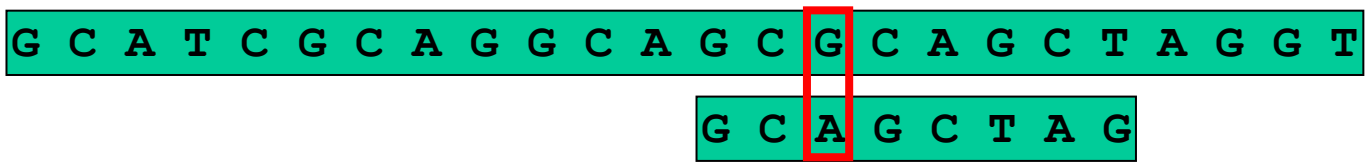
Beispiel 2

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

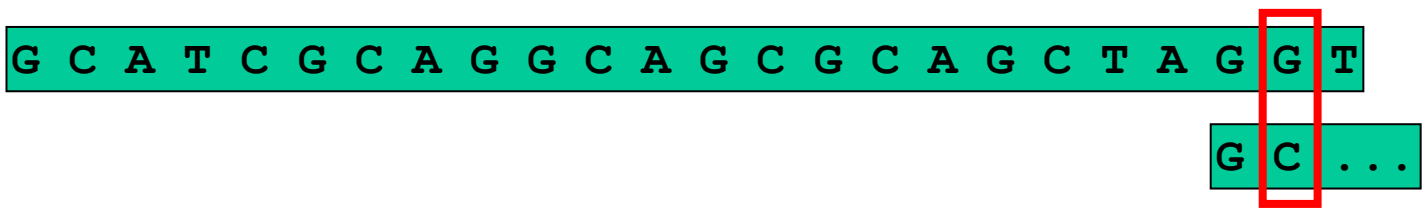
Schiebe um 5-2=3
Vergleiche ab 2+1



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



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



Kompletter KMP Algorithmus

```
compute sp_i';
c := 1;                  // Next comparison in T
p := 1;                  // Next comparison in P
while c+(n-p)<=m do
        while P(p)=T(c) and p<=n do
                p++;
                c++;
        end while;
        if p=n+1 then            // Match
                print c-n;
        else                     // Mismatch at c / p
                if p=1 then
                        c++;     // First comp. fails - move 1 pos
        end if;
        // Comparison in T will continue at position c
        // Comparison in P will continue after shift
        p:=sp_p-1' +1;           // p is mismatch pos. in P
end while;
```

Vergleich

	Z-Box	Boyer-Moore	Knuth-Morris-Pratt
Komp. Preproc. **Komp. Suche** **Komp. Gesamt**	O(m+n) O(m) O(m+n)	O(n) O(n*m) O(n*m)	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	• Je größer, desto schlechter- viele frühe Mismatches, kurze Shifts
Bemerkung	• Avg und Worst Case Komplexität gleich - es wird jedes Zeichen mind. einmal verglichen	• Worst-Case bei Wiederholungen (z.B: a^n in a^m) • Modifikationen zu linearem WC bekannt • Best Case ist O(m/n) [Suche $a^{n-1}b$ in b^m]	• Erweiterbar auf mehrere Pattern