

Algorithmische Bioinformatik

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



Ulf Leser
Wissensmanagement in der
Bioinformatik



Gerüst des Algorithmus

- Anordnung der Strings P und T
 - Erstes Zeichen von P „unter“ dem ersten von T
- Matche P und sein Gegenüber in T von rechts nach links
 - Also T[n] mit P[n], dann T[n-1] mit P[n-1], ...
 - Bei Mismatch oder Match für ganz P
 - Verschiebe P **um k Zeichen** nach rechts
 - Wieder von rechts nach links matchen
- 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 xabx**f**abzzabxzzbzzb
P abwx**y**abzz



T xabx**f**abzzab**w**zzbzzb
P abwx**y**ab**b**zz



Wie weit können wir
jetzt schieben ?

Bad Character Rule 2

- Beobachtung

- Es sei gerade $P[n]$ mit $T[j]$ aligniert; $j \geq n$
- Sei der erste Mismatch an Position i von P
- Sei x das Zeichen an Position $j-n+i$ in T
- Sei l das am weitesten rechts liegende Vorkommen von x in P
- Welches sind Kandidaten für einen Match mit x in P ?
 - Fall 1: x kommt in P nicht vor: Schiebe P um i Zeichen nach rechts
 - Fall 2: $l < i$. Also kommt x in P nur vor l vor – verschiebe P um $i-l$ Zeichen

T xabxkabzzabwzzbzzb
P abzwyabzz

↑ ↑
l i

T xabxkabzzabwzzbzzb
P abzwyabzz

←

Zusammengenommen

- Definition
 - Gegeben Pattern P. Dann sei $R(x)$ für alle $x \in \Sigma$ definiert als*
 - $R(x) = 0$, wenn $x \notin P$
 - *Sonst: $R(x) =$ „Position des am weitesten rechts liegenden Auftretens von x in P “*
- Berechnung leicht in $O(n)$ möglich
 - Wie?
- Damit
 - 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

Extended Bad Character Rule

- Beobachtung

- Die x rechts von i sind uninteressant – hier wurde schon geprüft
- Verbesserung: Verschiebe zum rechtesten x in P, das links von i liegt

T xabxkabzzabwz**z**bzzb
P abzwy**ab**zz



T Xabxkabzzabwz**z**pbzzb...
P abzwy**ab**zz



- Benötigt Berechnung der **relativen Positionen**

- Für jede Position i und jedes Zeichen x: Merke Position des am weitesten rechts aber links von i liegenden Vorkommen von x
- Array $[n, |\Sigma|]$ => **konstanter Lookup**, aber Platzverbrauch $O(n * |\Sigma|)$
- Listen für jedes Zeichen mit Positionen; **linearer Platz** $O(n)$, aber was kostet der Lookup?

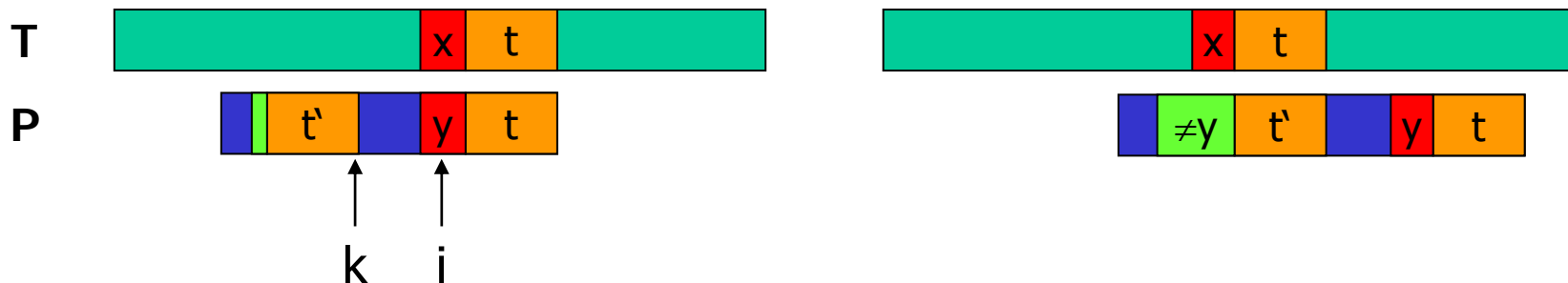
Good-Suffix Rule 2



- 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 das linke Ende von t in T

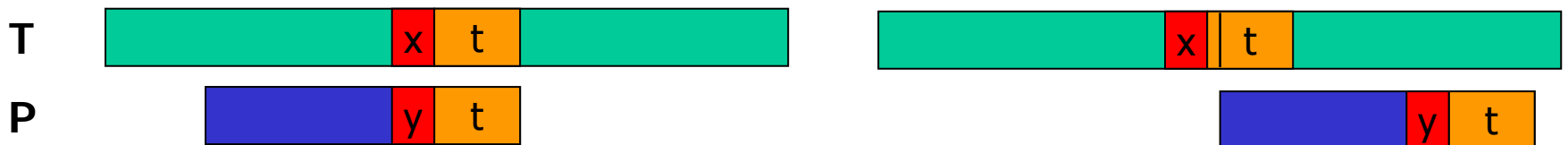
Fall 1

- Für die Mismatchposition i in P und $t=P[n-i+1,\dots]$, sei k das rechte Ende des am weitesten rechts liegenden Vorkommens von t in P mit $k < n$ und $P(k-|t|) \neq P(n-|t|)$ ($\neq y'$)
- Existiert kein solches Vorkommen von t in P , dann sei $k=0$
- Dann
 - Wenn $k \neq 0$: Verschiebe P um $n-k$



Fall 2

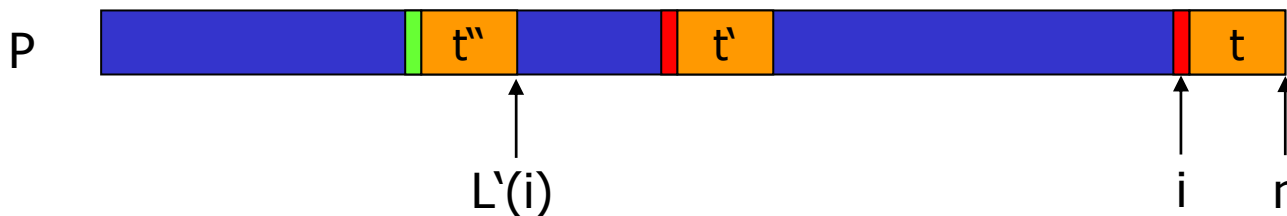
- Für die Mismatchposition i in P und $t=P[n-i+1,\dots]$, sei k das rechte Ende des am weitesten rechts liegenden Vorkommen von t in P mit $k < n$ und $P(k-|t|) \neq P(n-|t|)$ ($\neq y$), sonst 0
- Dann
 - Wenn $k \neq 0$: Verschiebe P um $n-k$
 - Wenn $k=0$ und $P \neq t$: Verschiebe P um $n-|t|+1$



- Man kann unter Umständen weiter verschieben
 - Später mehr

Preprocessing

- Woher wissen wir, wo und ob t in P noch vorkommt?
- Gesucht: Zu jedem Suffix ($t=P[i..n]$) von P den am weitesten rechts liegenden Endpunkt ($L'(i)$) eines identischen Teilstrings in P
- Definition:
 $L'(i)$ von P ist der größte Wert zur Position i für den gilt:
 - *Bedingung 1:* $P[L'(i)-|t|+1 .. L'(i)] = P[i..n]$
 - *Bedingung 2:* $L'(i) < n$
 - *Bedingung 3:* $P[L'(i)-|t|] \neq P[i-1]$ (*Strong good suffix*)
 - $L'(i) = 0$, falls kein solcher Teilstring existiert



Zwischenschritt

- Definition

Sei $N(j)$ 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 eines t

- Berechnung der $N(j)$ Werte

- $N(j)$ **symmetrisch** zu Z_i Werten des Z-Box Algorithmus
- Berechnung durch Z-Box Algorithmus auf **umgedrehtem P ($=P^r$)**

Ableitung von $L'(i)$ aus $N(j)$

- $N(j)$ Werte 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, nach dem wir suchen
 - Suffix darf sich nicht verlängern lassen, sonst hätten wir bei Schieben von P wieder denselben Mismatch
- Damit ist **$L'(i)$ der größte Wert j für den gilt: $N(j)=n-i+1$**
 - Alle Positionen mit $N(j)=n-i+1$ stehen für ein (nicht verlängerbares) Suffix der gewünschten Länge
 - Davon interessiert uns das am weitesten rechts liegende
 - Und das entspricht dem größten j

Boyer-Moore

```
compute L'(i);
compute R(x) for each x∈Σ;           // Simple BCR
k := n;                               // Runs thru T
while (k≤m) do
    align P with T with right end k;
    match P and T from right to left until
        mismatch:    Compute shift s1 using BCR and R(x);
                    Compute shift s2 using GSR and L'(i);
                    k := k + max(s1, s2);
        P matched:   print k;
                    k := k + 1;           // Could be impr.
end while;
```

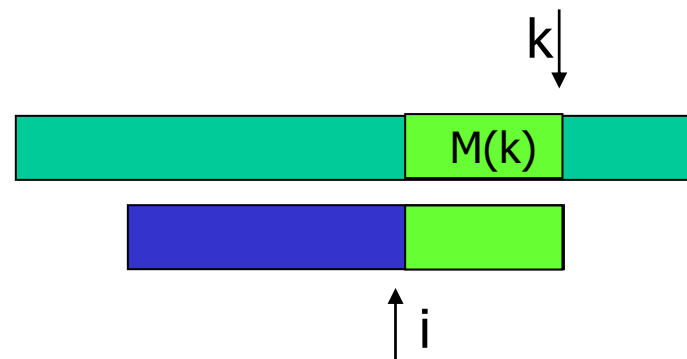
- Algorithmus hat Worst-Case $O(m \cdot n)$
 - Warum? Beispiel?
- Erweiterungen zu $O(m)$ kommt gleich
- Praxis: Sublinear

BM mit linearem Worst-Case

- Variante von Apostolico-Giancarlo (1986)
- Wir stellen sicher, dass **jedes Zeichen in T maximal einmal einen Match** erzeugt
- Damit wären wir fertig [Analogie zu Z-Box]
 - Nach jedem Mismatch schieben wir um mindestens 1 – also haben wir insgesamt maximal m Mismatches
 - Außerdem maximal m Matches (siehe oben)
 - Also ist der Algorithmus $O(m)$

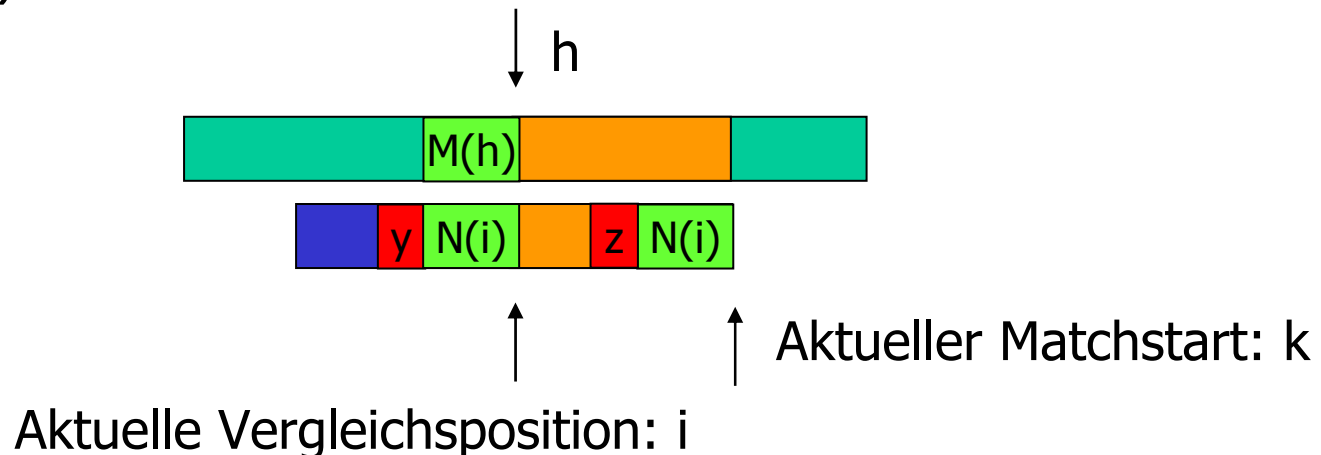
M: Suffixe matchen Suffixe

- 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 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 in $M(k)$ (Details später)



M: Suffixe matchen Suffixe

- M wird wie folgt benutzt
 - Später vergleichen wir $M(h)$ und $N(i)$
 - $N(i)$ sind Suffixe von P in P
 - $M(h)$ 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 $i < k$



Zusammenfassung

- Exakt-String-Matching Algorithmus mit
 - Average Case sublinear
 - Worst-Case linear
- Praxistauglichster Algorithmus für viele Arten von Text
 - Eignung hängt von Größe des Alphabets und Häufigkeiten der einzelnen Zeichen ab
 - Wenn Zeichen sehr ungleich häufig vorkommen, kann man das auch ausnutzen ([wie?](#))
 - Für sehr kurze Pattern gibt es schnellere Verfahren (SHIFT-AND)
- Als nächstes: Knuth-Morris-Pratt

Inhalt dieser Vorlesung

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

Noch ein exakter Stringmatching Algorithmus?

- Klassiker
- Intuitive Erweiterung des naiven Algorithmus
 - Läuft auch von links nach rechts
- Schöner Beweis der Korrektheit
- Erweiterung zum linearen Matching mit mehrerer Pattern – Aho-Corasick

Grundidee

- Erinnerung an den naiven Algorithmus

ctgagatcgcgta

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

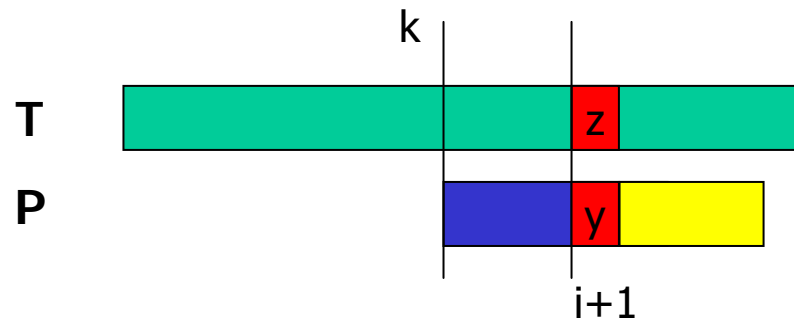
Also: 4 Zeichen (mind.) schieben

- 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+1 (in P) gilt: $T[k..k+i-1] = P[1..i]$



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

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' : 00010**000**420
 abcaeabc
 abcaeabcab

KMP im Überblick

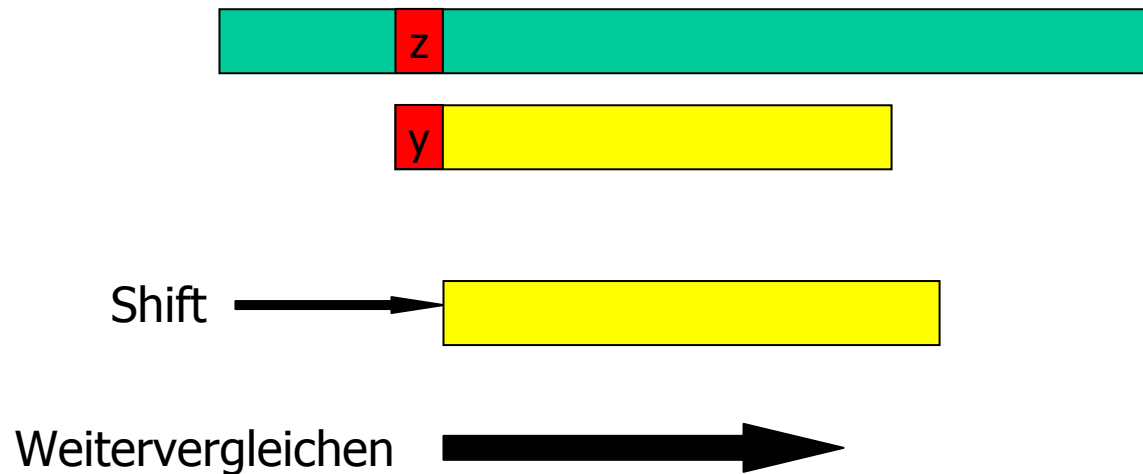
- Phasen
 - Preprocessing von P: Berechne sp_i und sp_i'
 - Matching von links nach rechts wie in naivem Algorithmus
 - Bei Match – weiter nach rechts matchen
 - Bei Mismatch (an Position $i+1$ in P)
 - Schiebe P um ... (Länge durch sp_i' und i bedingt - später)
 - Weitervergleichen ab ... (später)
 - 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 $sp_i'+1$** (und nicht bei 1)

Detailübersicht

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

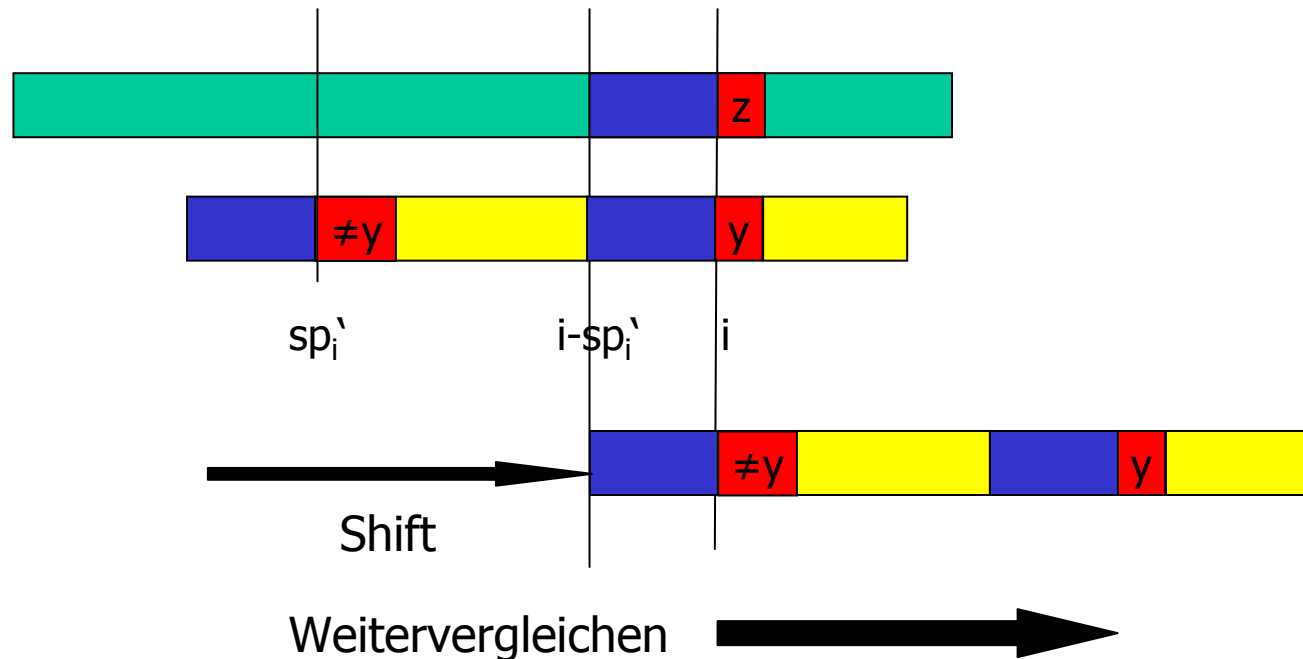
Shift Regel 1

- Wenn $P[1]$ und $T[k]$ **sofort mismatchen**
 - Schiebe P um ein Zeichen nach rechts
 - Vergleiche weiter ab $P[1]$



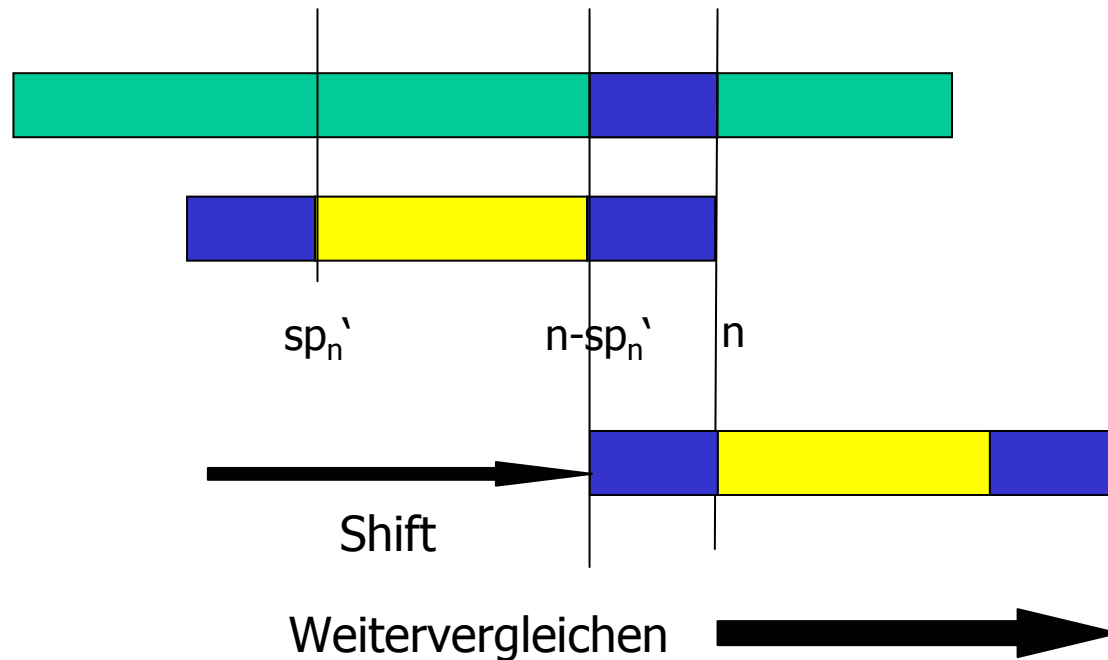
Shift Regel 2

- 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^{\wedge} (und nicht sp_i)?

- Sind beide korrekt?
 - Sei $sp_i \neq sp_i^{\wedge}$ für ein i und bei $i+1$ tritt Mismatch auf
 - Dann ist $P[i+1] \neq T[k+i+1]$
 - Wegen $sp_i \neq sp_i^{\wedge}$ gilt: $P[i+1] = P[sp_i+1]$
 - Der nächste Vergleich $P[sp_i+1]$ mit $T[k+i+1]$ ist überflüssig
- Beobachtung
 - Es gilt: $sp_i^{\wedge} \leq sp_i$
 - Präfixe, die sp_i genügen, sind mindestens solange wie Präfixe für sp_i^{\wedge}
 - Man schiebt um $i - sp_i^{\wedge}$
 - Also: sp_i^{\wedge} erlaubt **längere Verschiebungen** als sp_i

Korrektheit der Shift-Regel

- Was passiert bei Vorkommen des Präfix zwischen dem Präfix und dem Substring vor dem Mismatch?

- Versuchen wir mal

...BZZZBABC EBBZZEFFGAA
BZZZBCBDD

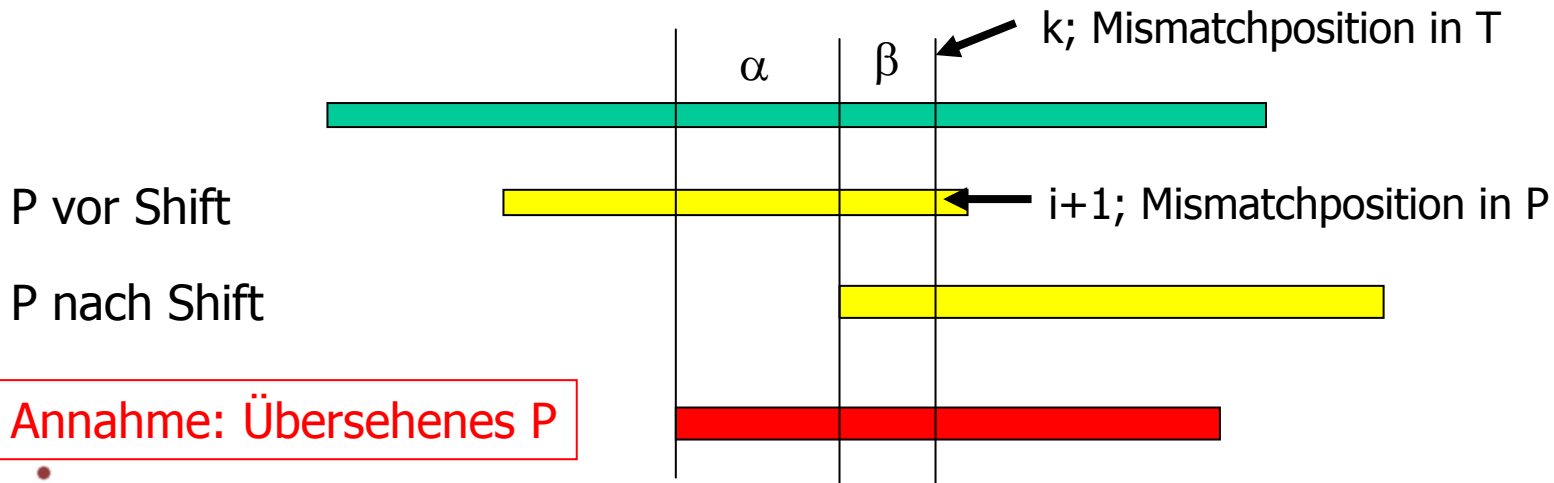
- Wir haben drei Substrings (B), die Präfix sind und mit unterschiedlichen Zeichen fortgesetzt werden
- Alignment geht schief

...BBBBBABC EBBZZEFFGAA
BBBBBCBDD

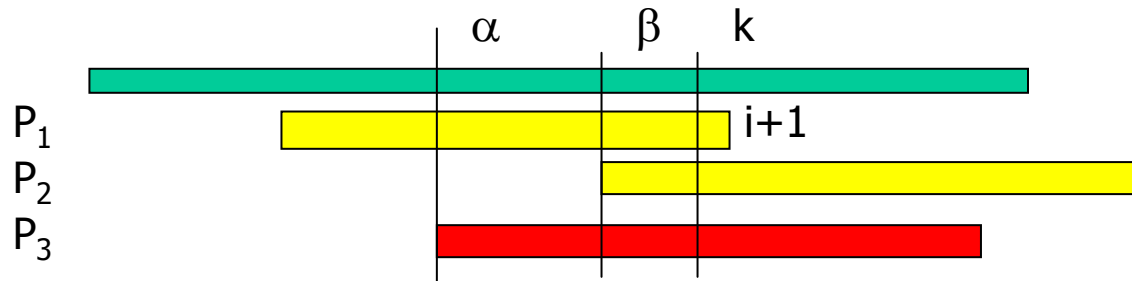
...BBBBBABC EBBZZEFFGAA
BBBABCDD

Korrektheit der Shift-Regel

- Theorem
 - Die Shift-Regel verschiebt nie soweit, dass ein Vorkommen von P in T übersehen wird
- Beweis
 - Wenn das anders wäre, hätten wir:

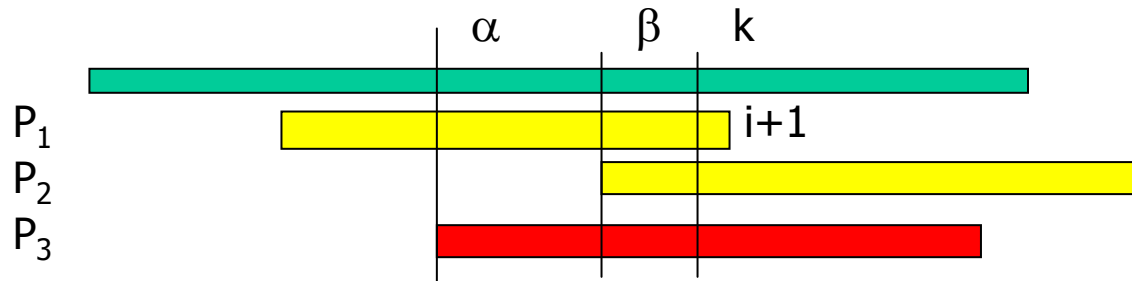


Beweis 2



- Wir zeigen, dass diese Situation im **Widerspruch zur Definition** von sp_i' steht
 - β ist Präfix von P ; $|\beta| = sp_i'$ (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)
 - Das Zeichen nach $\alpha\beta$ ist ein Mismatch
 - $P[i+1] \neq P[|\alpha\beta|+1]$ (sonst kein Mismatch in Position k)

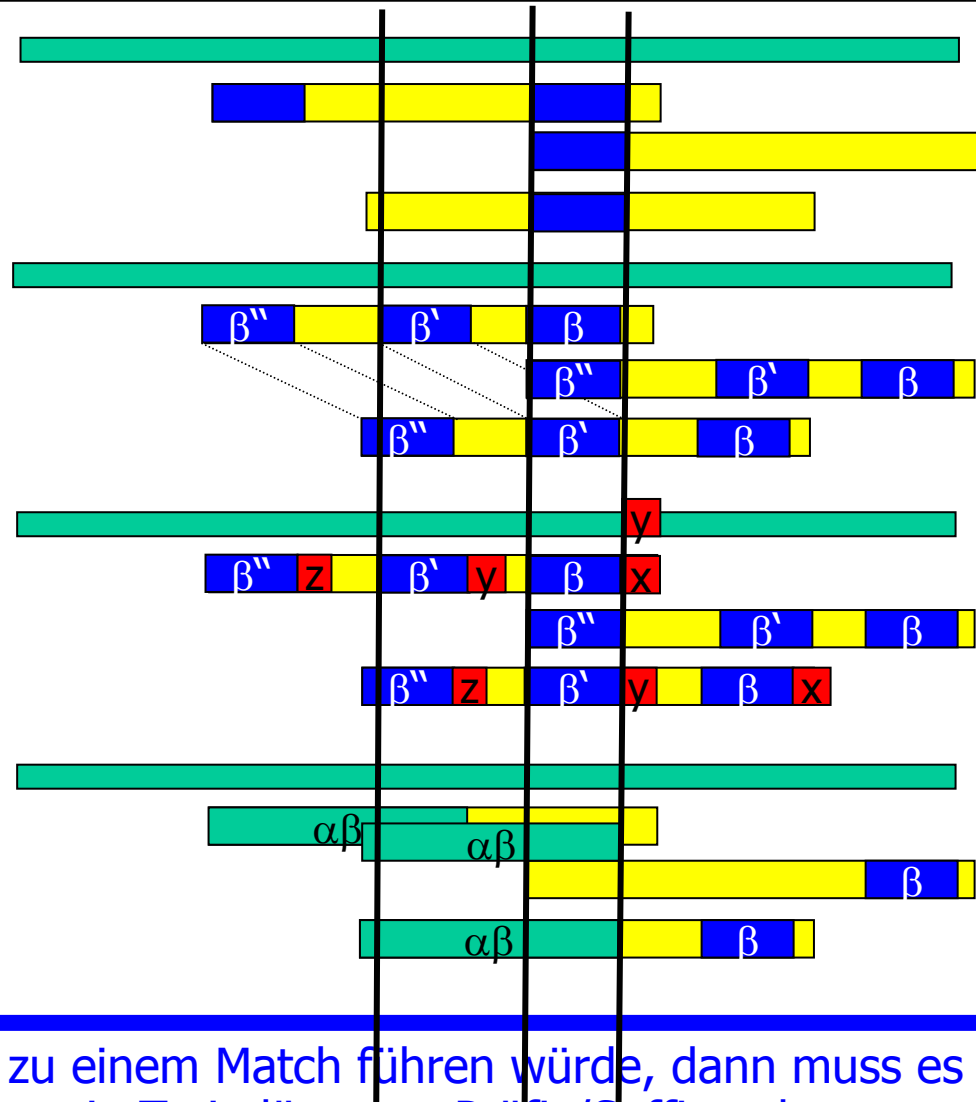
Beweis 3



- 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'**

Mehrere β

- Ausgangssituation
- Sei β' das dritte Vorkommen von β zwischen β und β''
- Weil P3 matcht, müssen nach β und β' unterschiedliche Zeichen kommen (es ist aber möglich, dass $z=y$)
- P1 und P3 matchen tw. denselben Teilstring in T; $\alpha\beta$ ist also Präfix von P und Suffix von $P[1\dots i]$



Wenn es also ein β' gibt, das zu einem Match führen würde, dann muss es wegen der Überlappung in T ein längeres Präfix/Suffix geben

Komplexität von KMP

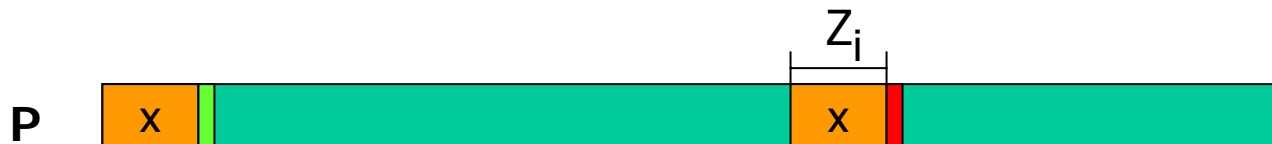
- Theorem
 - KMP benötigt *höchstens* $2m$ Zeichenvergleiche
- Beweis (wie immer)
 - Denken wir uns wieder compare/shift Phasen
 - Mismatches
 - Pro Phase gibt es maximal einen Mismatch
 - Es gibt höchstens m Shift/Vergleich-Phasen
 - es wird immer um mindestens 1 verschoben
 - Matches
 - Jeder Vergleich beginnt in T entweder
 - Am letzten Zeichen des letzten Vergleichs (bei Mismatch)
 - Am Zeichen rechts vom letzten Zeichen des letzten Vergleichs (bei vollständigem Match)
 - Damit wird ein Zeichen in T niemals mehr als einmal mit positivem Ausgang verglichen
 - 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

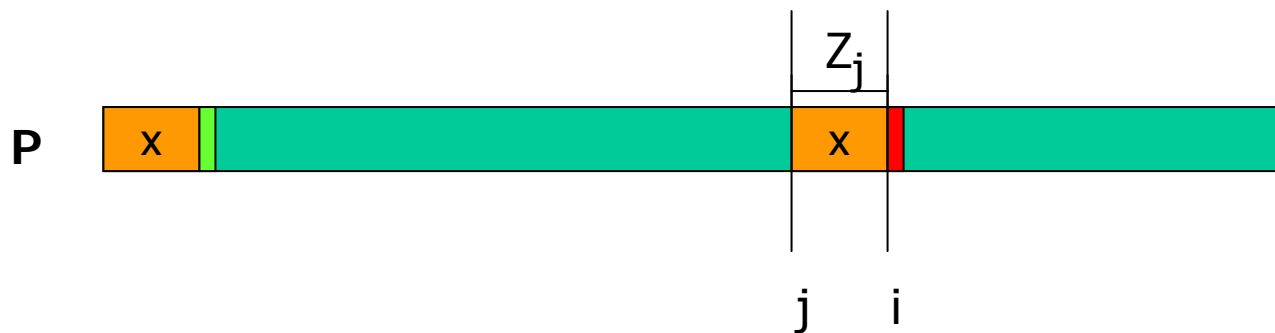
Preprocessing

- 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

- Für alle i suchen wir 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
    spi' := 0;
end for;
for j = n downto 2      // Spätere (weiter links) Treffer
    i := j + Zj - 1;    // überschreiben frühere
    spi' := Zj;
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

G C A T C G C A G G C A G C G C A G C T A G G T

G C A G C T A G

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

G C A T C G C A G G C A G C G C A G C T A G G T

G C A G C T A G

Schiebe um 1
Vergleiche ab 1

G C A T C G C A G G C A G C G C A G C T A G G T

G C A G C T A G

Schiebe um 1
Vergleiche ab 1

G C A T C G C A G G C A G C G C A G C T A G G T

G C A G C T A G

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

G C A T C G C A G G C A G C G C A G C T A G G T

G C A G C T A G

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

Shift-Regel:

Mismatch bei $i+1$

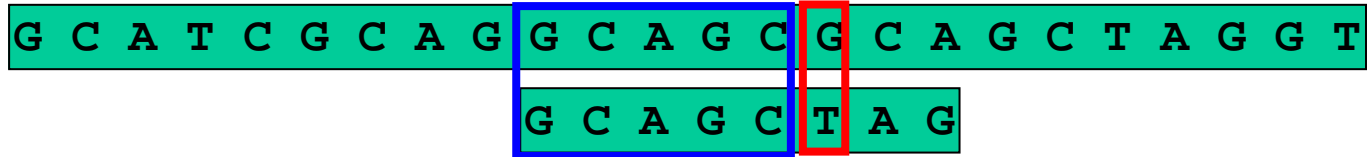
Schiebe um $i - sp_i'$

Vergleiche ab $sp_i' + 1$

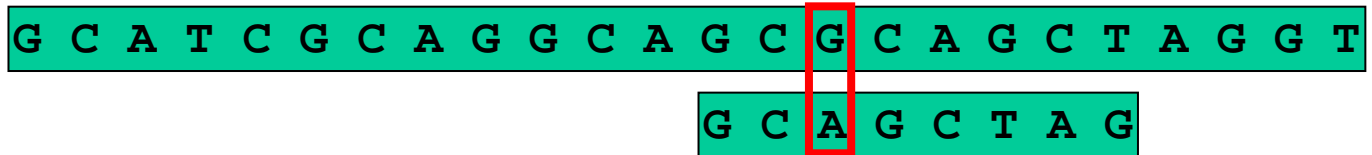
Beispiel 2

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

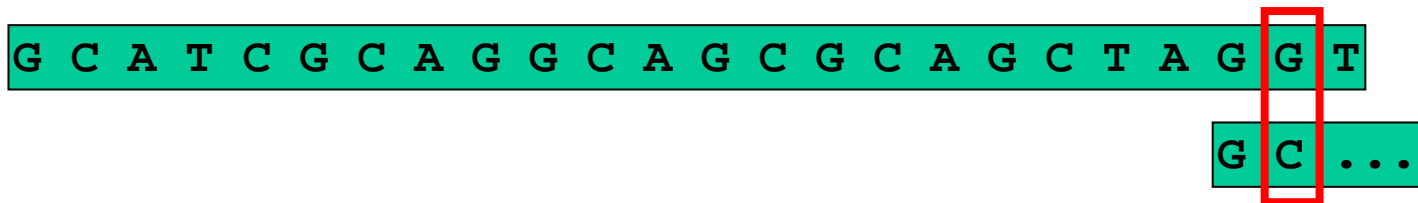
Schiebe um $5-2=3$
Vergleiche ab $2+1$



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



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



Kompletter KMP Algorithmus

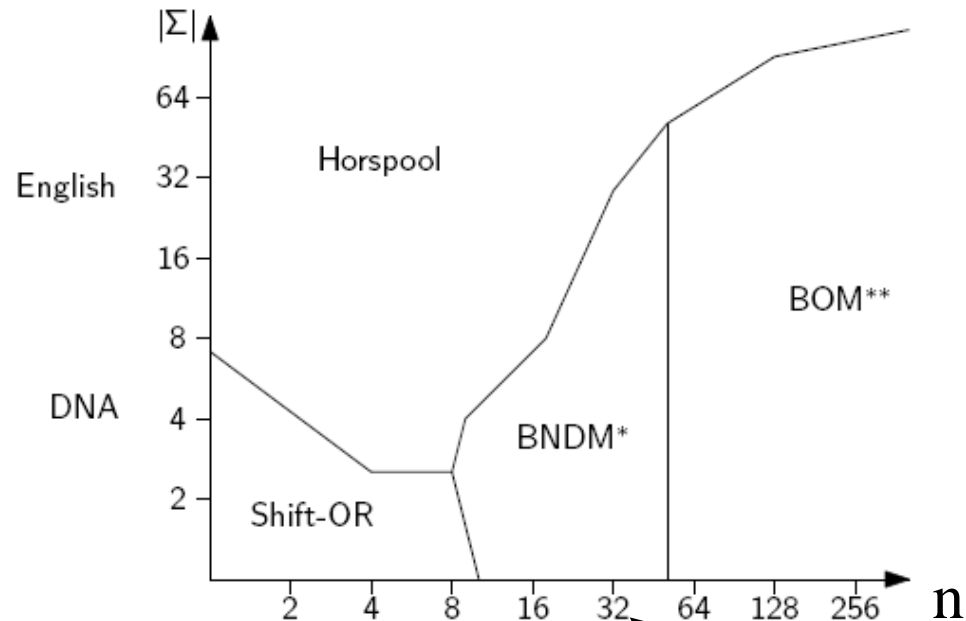
```
compute spi`;  
h := 1;           // Next comparison in T  
i := 1;           // Next comparison in P  
while h+(n-i)<=m do  
    while P[i]=T[h] and i<=n do  
        i++;  
        h++;  
    end while;  
    if i=n+1 then      // Match  
        print h-n;  
    else                // Mismatch at h/i  
        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:=spi-1` +1;      // i is mismatch pos. in P  
    end while;
```

Vergleich

	Z-Box	Boyer-Moore (Apostolico-Giancarlo)	Knuth-Morris-Pratt
Preprocessing	$O(m+n)$	$O(n)$	$O(n)$
Suche	$O(m)$	$O(m)$	$O(m)$
Gesamt	$O(m+n)$	$O(m+n)$	$O(m+n)$
Größe Alphabet	<ul style="list-style-type: none"> • Praktisch unabhängig von Alphabetgröße 	<ul style="list-style-type: none"> • Je größer, desto besser – BCR führt zu großen Sprüngen • BCR greift selten bei kleinen Alphabeten 	<ul style="list-style-type: none"> • Praktisch unabhängig von Alphabetgröße
Bemerkung	<ul style="list-style-type: none"> • Avg und Worst Case Komplexität gleich 	<ul style="list-style-type: none"> • 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] 	<ul style="list-style-type: none"> • Avg und Worst Case Komplexität gleich • Erweiterbar auf mehrere Pattern

Experimenteller Vergleich

(Gonzalo Navarro & Mathieu Raffinot, 2002)



- Horspool: Variante des Boyer-Moore
- Shift-OR: Sehr schnelle Bit-Operationen, Maschinenwortlänge
- BNDM: Backward nondeterministic Dawg Matching
- BOM: Backward Oracle Matching