

Bioinformatik

Boyer-Moore Algorithmus

Ulf Leser

Wissensmanagement in der
Bioinformatik



Z-Algorithmus: Preprocessing

- Definition

- Sei $i > 1$. Dann ist $Z_i(S)$ die Länge des *längsten Substrings* x von S mit

- $x = S[i..i+|x|-1]$ (x startet an Position i in S)
- $S[i..i+|x|-1] = S[1..|x|-1]$ (x ist auch Präfix von S)

- x ist die *Z-Box* von S an Position i mit Länge $Z_i(S)$



Lineare Berechnung der Z_i Werte

- Trick
 - Verwenden von bereits bekannten Z_i zur Berechnung von Z_k ($k > i$)
- Grundaufbau
 - Lineares Durchlaufen des Strings (Laufvariable k)
 - Kontinuierliches Vorhalten der Werte $l=l_k$ und $r=r_k$
- **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 gibt, die k enthält
 - Wir wissen nichts über den Bereich in S ab k
 - Dann gehen wir primitiv vor
 - Berechne Z_k durch Zeichen-für-Zeichen Matching
 - Wenn $Z_k > 0$, setze $r = r_k$ und $l = l_k$

Beispiel

k
CTCGAGTTGCAG
0
1
0
?

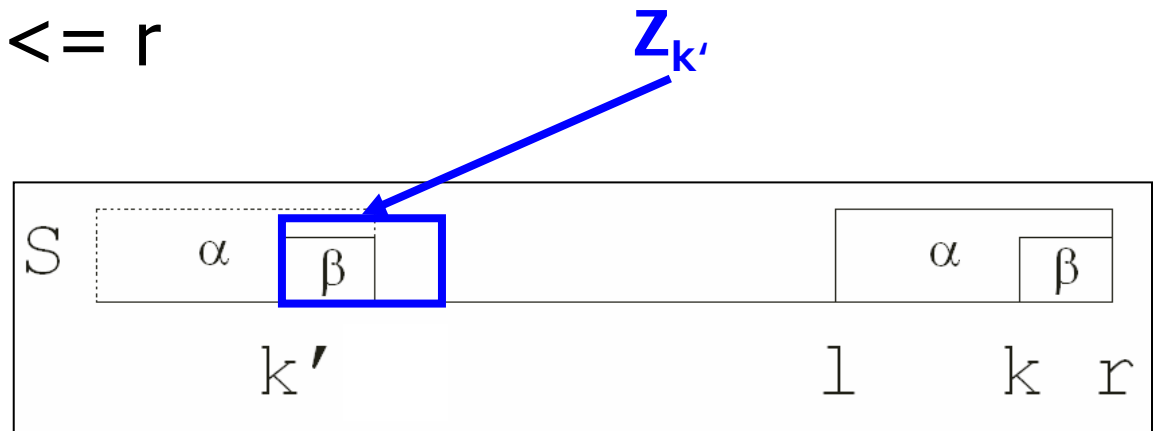
Gegenbeispiel

lk r
CTACTACTTTGCAG
0
0
5
?

Z-Algorithmus, Fall 2

- Möglichkeit 2: $k \leq r$

- Die Situation:



- Also

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

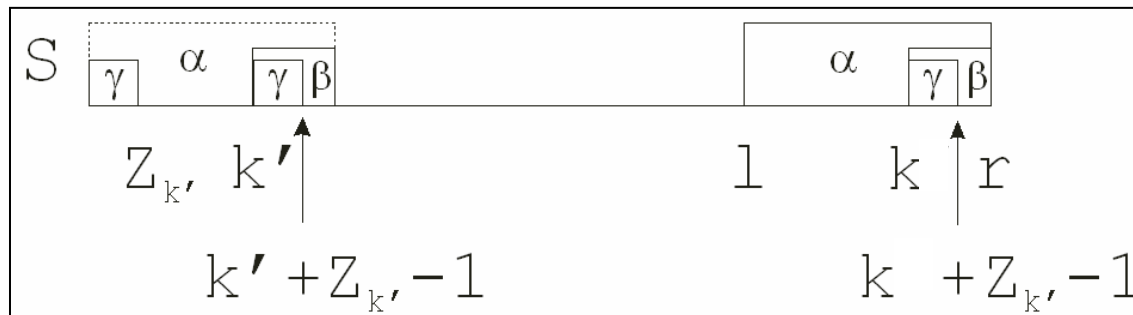
Z-Algorithmus, Fall 2.1

- Fallunterscheidung

- $Z_{k'} < |\beta| = r - k + 1$

Dann ist das Zeichen an $k' + Z_{k'}$ ein Mismatch bei der Präfixverlängerung. Dann ist das Zeichen $S[k + Z_k]$ der gleiche Mismatch. Also:

$Z_k = Z_{k'}$; r und l unverändert



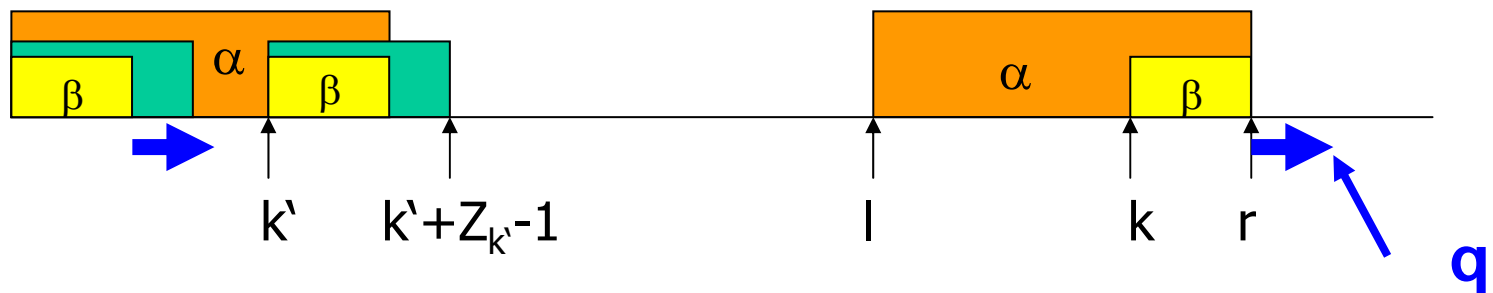
Z-Algorithmus, Fall 2.2

- $Z_{k'} \geq |\beta|$: Dann ist β ein Präfix von S
 - ... dass sich vielleicht noch verlängern lässt
 - Wenn $Z_{k'} > |\beta|$, dann wissen wir: $S[|\beta|+1]=S[k'+|\beta|]$
 - Wir wissen aber nichts über $S[r+1]$ – dieses Zeichen wurde noch nie betrachtet

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 $q \neq r + 1$: $l = k$



Komplexität

- Theorem

Der Z-Box Algorithmus berechnet alle Z-Werte in $O(|S|)$

- Beweis

- Der Algorithmus ist mindestens $O(|S|)$ durch die FOR-Schleife
- Wir zählen m = „Anz. Matches“ und m' = „Anz. Mismatches“
 - Erst m' . 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: $m' \leq |S|$

Komplexität 2

- Beweisfortsetzung

- Jetzt m. Wann führt der Algorithmus Matches aus?

- Induktionsanfang – Egal für Komplexität (max. $|S|$)

- Fall 1

- » Nehmen wir x Matches an, dann Mismatch; r wird nach rechts verschoben, Fälle 2.1 und 2.2 werden eintreten

- Fall 2.1

- » Es werden keine Zeichen verglichen

- Fall 2.2

- » Verglichen wird nur rechts von r , und damit rechts vom letzten Match

- Also kann ein Zeichen höchstens einmal einen Match erzeugen

- Also gilt: $m \leq |S|$

- Qed.

Linearer Stringmatching Algorithmus

- Verwendung der Z-Boxen für String Matching

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

- Komplexität

- Schleife wird |S|-mal durchlaufen => $O(m)$

Inhalt dieser Vorlesung

- Boyer-Moore Algorithmus
- Bad Character und Good Suffix Rule
- Preprocessing
- Beispiel
- Erweiterung zum linearen Worst-Case

Boyer-Moore Algorithmus

- R.S. Boyer /J.S. Moore. „A Fast String Searching Algorithm“, Communications of the ACM, 1977
 - Darstellung hier nach Gusfield, Kap. 2 bzw. 3.1
- Grundidee
 - Alignierung der Strings wie in naivem Algorithmus
 - „Äußere Schleife“
 - P springt beim Schieben weiter als 1, wenn möglich
 - Matching von **rechts nach links**
 - „Innere Schleife“

Boyer-Moore Algorithmus

- Vorweg
 - Zunächst die Grundidee mit Worst-Case $O(n*m)$
 - Verbesserungen zu $O(m)$ am Ende
- Average-Case sublinear durch Sprünge
 - „Normaler“ Text erlaubt große Sprünge
 - Großes Alphabet, z.B. Englisch, Proteinsequenzen, Chinesisch
 - Vergleich Z-Box Algorithmus: Ist nie sublinear, da jedes Zeichen von T angefasst werden muss
 - Boyer-Moore ist Methode der Wahl für normalen Text

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 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 i vor – verschiebe P um $i-l$ Zeichen

T xabxkabzzabwzzbzzb
P abzwyabzz

↑ ↑
l i

T xabxkabzzabwzzbzzb
P abzwyabzz

←

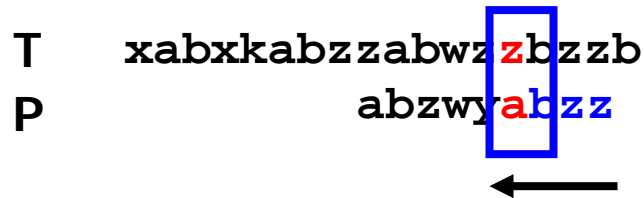
Wie weit können wir
jetzt schieben ?

Bad Character Rule 3

- 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: $l < i$. x kommt x in P nur vor i vor – verschiebe P um $i-l$ Zeichen
 - Fall 3: $l > i$. Das nützt uns (erst mal) nichts

T xabxkabzzabwz**zb**zzb
P abzw**ab**zz



„z“ gibt es rechts von i
kann man nix machen (oder?)

Bad Character Rule 4

- 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: $l < i$. x kommt x in P nur vor i vor – verschiebe P um $i-l$ Zeichen
 - Fall 3: $l > i$. Das nützt uns (erst mal) nichts
 - Fall 4: $l = i$: ?

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

- Zweite Regel zum Verschieben
- Unabhängig von Bad Character Rule
- Grundidee
 - Bad Character Rule sucht nach dem ersten Mismatch
 - Bis dahin haben wir aber schon einen (evt. recht langen) **Match** gefunden
 - Wo ist dieser Match noch in P enthalten?
 - Preprocessing von P erforderlich

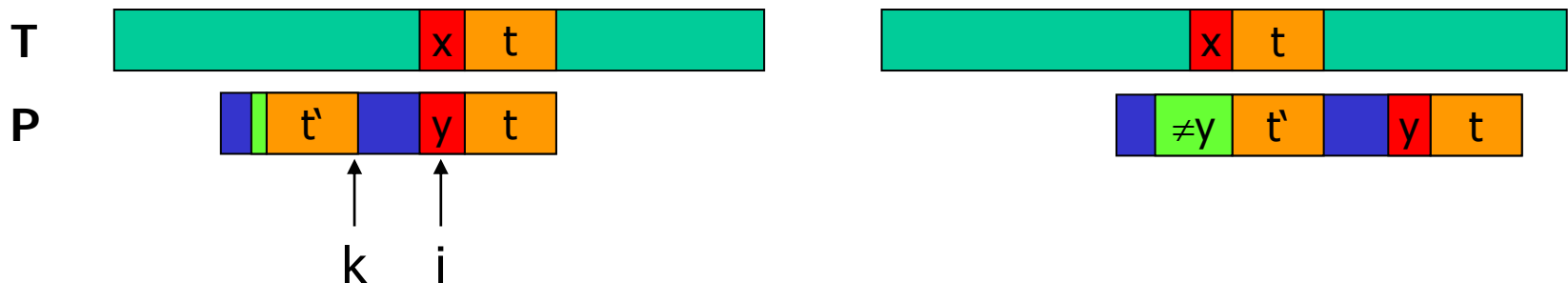
Good-Suffix Rule 2



- 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

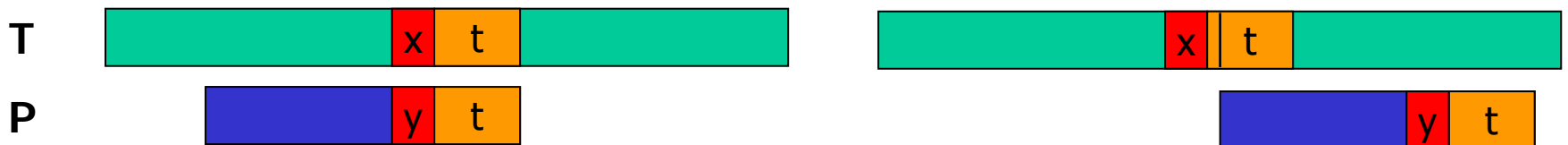
- Für die Mismatchposition i in P und $t=P[n-i+1,..]$, sei k das rechte Ende des am weitesten rechts liegenden Vorkommen von t in P mit $k < n$ und $P(k-|t|) \neq P(n-|t|)$ ($,y'$)
- Existiert kein solches Vorkommen von t in P , dann sei $k=0$
- Dann
 - Wenn $k \neq 0$: Verschiebe P um $n-k$



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

Fall 2

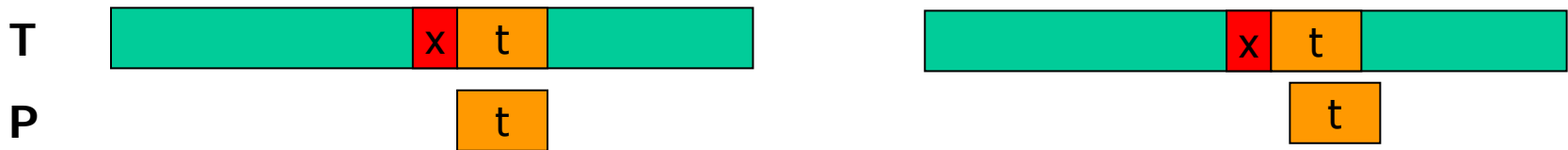
- Für die Mismatchposition i in P und $t=P[n-i+1,\dots]$, sei k das rechte Ende des am weitesten rechts liegenden Vorkommen von t in P mit $k < n$ und $P(k-|t|) \neq P(n-|t|)$ (y), sonst 0
- Dann
 - Wenn $k \neq 0$: Verschiebe P um $n-k$
 - Wenn $k=0$ und $P \neq t$: Verschiebe P um $n-|t|+1$



- Man kann unter Umständen weiter verschieben
 - Später mehr

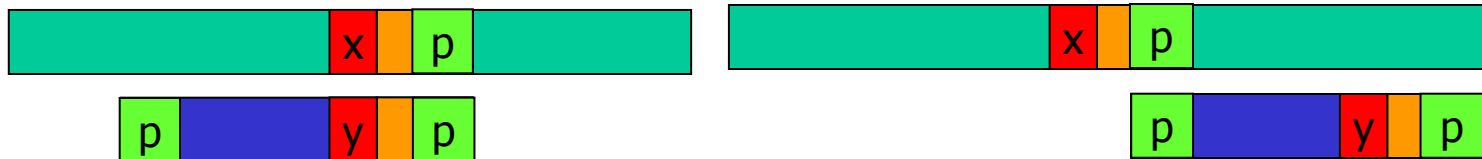
Fall 3

- 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 $P(k-|t|) \neq P(n-|t|)$ (,y'), sonst 0
- Dann
 - Wenn $k \neq 0$: Verschiebe P um $n-k$
 - Wenn $k=0$ und $P \neq t$: Verschiebe P um $n-|t|+1$
 - Wenn $k=0$ und $P=t$: Verschiebe P um 1



Hinweis

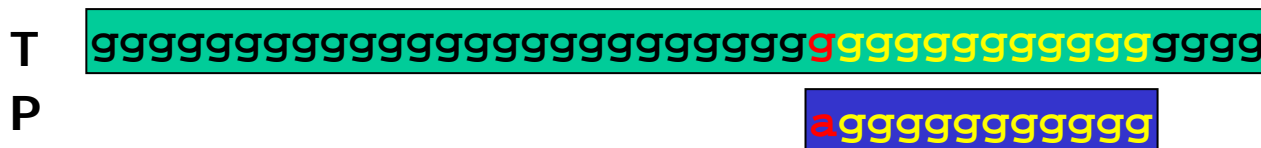
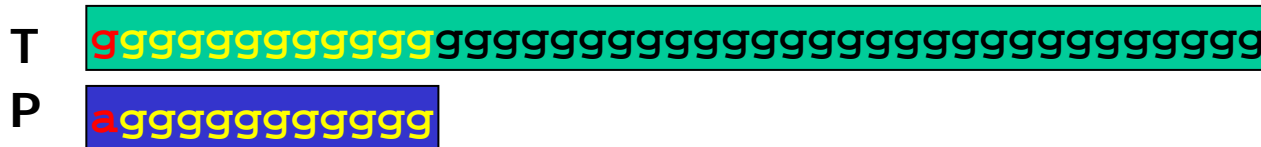
- Was heißt „... unter Umständen weiter...“?
- Bisher
 - Wenn t in P nicht mehr vorkommt, verschiebe P um $n - |t| + 1$
- Man kann weiter verschieben
 - Betrachte das längste Präfix von P , das auch Suffix von P ist
 - Ist im **voraus berechenbar**



- Details: Siehe Gusfield, $l(i)$ -Werte

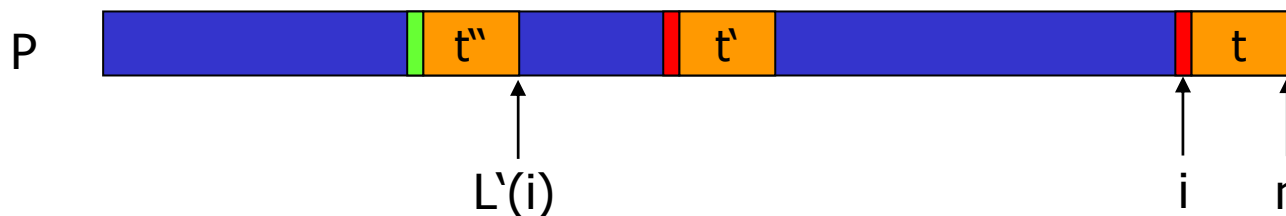
Beispiel

- Das Worst-case Beispiel der BCR läuft jetzt (mit dem „weiter“ Trick) in linearer Zeit



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



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), die man nicht weiter verlängern kann ...“
- Das Boxer-Moore Preprocessing ist also (fast) eine „invertierte“ Z-Box Berechnung
 - Unterschied: Es kann viele Präfix-Substring Paare geben, wir suchen aber nur eines davon

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

Beispiel

1234567890123
dcabcabdabdab
$N(j) = 0002002005000$

- Suchen wir $L'(12)$
 - Also Suffixe „ab“
 - Zeichen vor diesem „ab“ darf nicht „d“ sein
 - Das sind alle Positionen j mit $N(j)=2$
 - Denn dort liegt ein längstes Suffix der Länge 2
 - Davon das „rechtteste“ ist unser Treffer
 - Also: $L'(12) = 7$

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;
```

- **Komplexität: $O(n)$**

- Z-Box ist $O(n)$
- L' -Werte ist $O(n)$

$$N(j) = n - i + 1$$

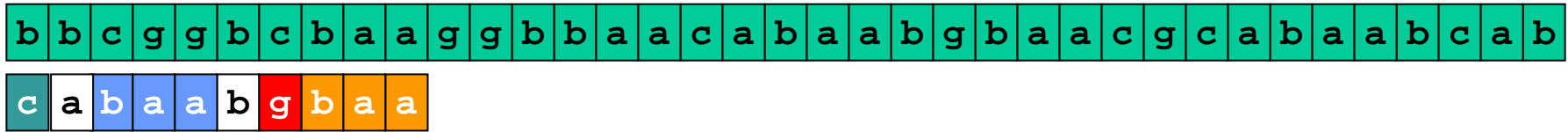
- Wir machen hier nur Preprocessing des Pattern; T bleibt unberührt

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

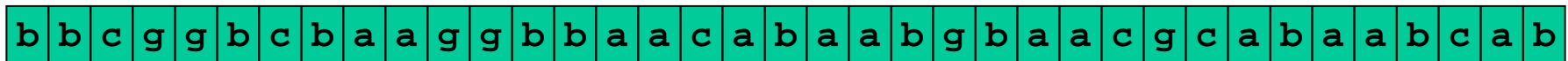
Beispiel mit EBCR



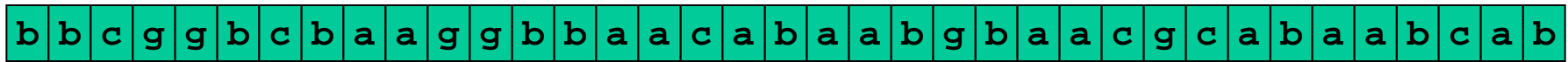
EBCR wins



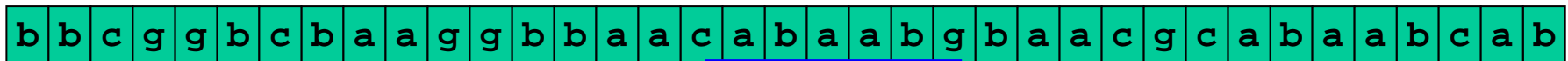
Mit BCR Schieben um 1 (geht immer)



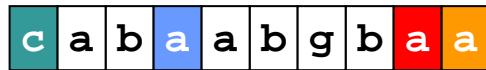
GSR wins



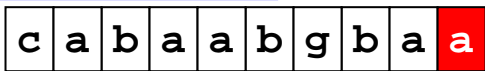
GSR wins



Mit /ohne l'



- Match
- Good suffix
- Mismatch
- Ext. Bad character



BM mit linearem Worst-Case

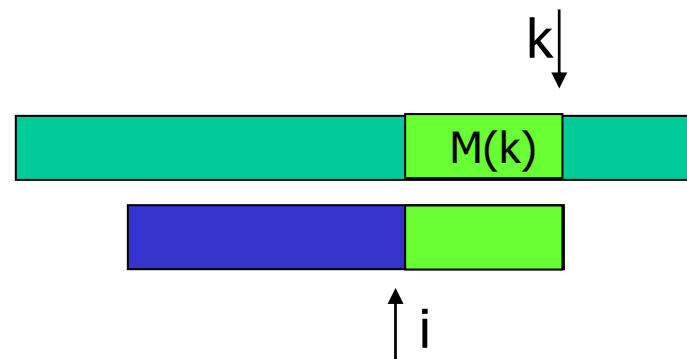
- Variante von Apostolico-Giancarlo (1986)
- Wir stellen sicher, dass **jedes Zeichen in T maximal einmal einen Match** erzeugt
- Damit wären wir fertig [Analogie zu Z-Box]
 - Nach jedem Mismatch schieben wir um mindestens 1 – also haben wir insgesamt maximal m Mismatches
 - Außerdem maximal m Matches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

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
 - Sei M ein Array $\text{int}[m]$, initialisiert mit -1

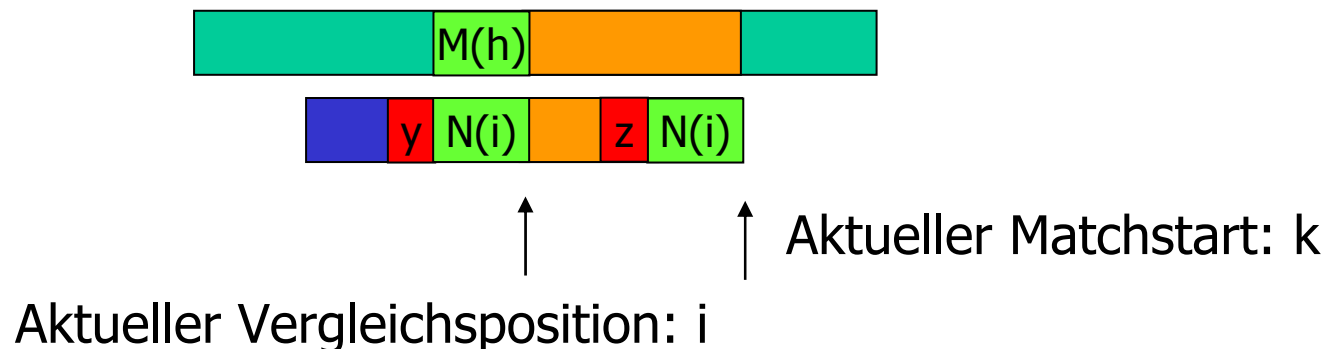
M: Suffixe matchen Suffixe

- 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 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)



M: Suffixe matchen Suffixe

- M wird wie folgt benutzt
 - 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 $i < k$

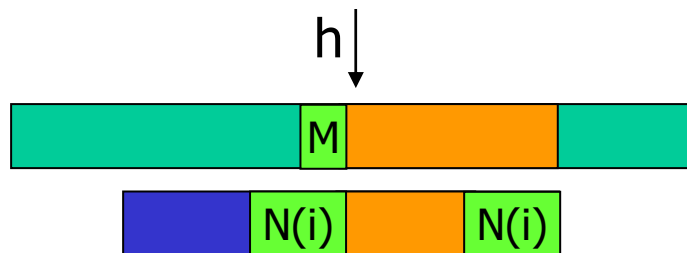


Details – 1

- Wir beginnen eine Phase an Position k in T
 - Wir schieben immer wie beim „normalen“ Boyer-Moore
 - Wir **sparen nur Vergleiche** im Shift-Schritt einer 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 ($h--$; $i--$)
 - $T[h]\neq P[i]$: Kein Match; Setze $M(k)=k-h$; beende Phase

Details – 2

- Fall 2: Wenn $M(h) < N(i)$
 - Also matched P mit T ab $h-M(h)+1$ bis h
 - 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 – 3

- Fall 3: Wenn $M(h) \geq N(i)$ und $N(i) = i > 0$
 - In T liegt ein String, der bei h endet,
 - ... der länger ist als $N(i)$
 - ... und dessen Suffix mit $N(i)$ matched
 - ... und $N(i)$ ist so lang, dass der Teil in P vor i und der Teil nach i (haben wir schon gesehen) zusammen ein kompletter Match sind
 - Vorkommen von P melden
 - Setze $M(k) := n$
 - Gusfield setzt auf $k-h$ (kürzer ist nie schlimmer) wg Beweis
 - Phase beenden (mit Match)



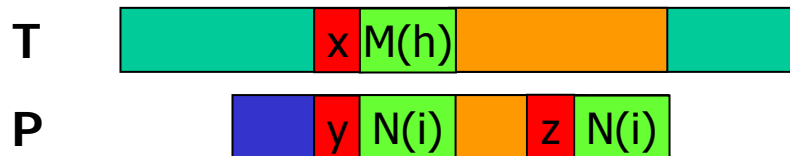
Details – 4

- 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 schlimmer) wg Beweis
 - Phase beenden (mit Mismatch)



Details – 5

- 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 k endet
 - Was danach kommt, wissen wir nicht
 - Setze $h := h-M(h)$; $i := i-M(h)$;
 - Dann weiter nach links matchen
 - Phase läuft weiter



Beweis

- Sparen wir uns
- Intuitiv
 - Matches
 - Zeichen, die wir matchen, führen zur Erhöhung von $M(k)$
 - Wenn $M(h) > 0$, dann haben wir einen Substring in T schon gesehen und gematched
 - In all diesen Fällen überspringen wir die nächsten $M(h)$ Zeichen oder beenden die Phase
 - Mismatches
 - Nach jedem Mismatch beenden wir die Phase

Zusammenfassung

- Nach etwas Arbeit
- Wir haben einen 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
 - Für sehr kurze Pattern gibt es schnellere Verfahren (SHIFT-AND)
- Als nächstes: Knuth-Morris-Pratt
 - Lineares String Matching
 - Erweiterbar auf mehrere Pattern