FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Theoretische Grundlagen der Informatik, Algorithmen und Datenstrukturen

2. Semester Angewandte Mathematik und Informatik

© Prof. Dr. rer. nat. Jörg Striegnitz
striegnitz@fh-aachen.de

Algorithmen und Datenstrukturen

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Der Algorithmus von Boyer und Moore

Suche in Texten

© Prof. Dr. rer. nat. Jörg Striegnitz
striegnitz@fh-aachen.de

Die Grundidee des Boyer-Moore Verfahrens

- Von Robert S. Boyer und J[1] Strother Moore 1977 veröffentlicht.
- Muster wird von **rechts**/**hinten** **nach links**/**vorne** verglichen
  nicht von links nach rechts, wie bei den bisher vorgestellten Verfahren!
- Bei Unglechheit (*Mismatch*) verschiebe Muster nach rechts
- Versatz wird durch zwei **Heuristiken** (Strategien) "maximiert":
  1. die ***Bad Character*-Heuristik** und als Ergänzung ggf.
  2. die ***Good Suffix*-Heuristik**.
- Wir skizzieren hier nur die Idee des Boyer-Moore Verfahrens
  Implementierung wird evtl. Teil der Hausaufgaben

[1] „J" ist der volle Vorname

Implementierung - 1

```
1   // Eine Heurstic berechnet den Index des nächsten Zeichens im Text
2   // für den Fall, dass t[i] und p[j] ungleich sind.
3   public interface Heuristic {
4     int Next(String t, String p, int i, int j);
5   }
6
7   public class BoyerMoore {
8     // cmp führt den Vergleich aus, wenn das Ende des Musters unter t[i] liegt.
9     // Im Erfolgsfall wird der Index i den Ergebnissen in res hinzugefügt.
10    // cmp nutzt die übergebenen Heuristiken um den maximal möglichen Versatz
11    // für das Muster zu ermitteln und gibt diesen an den Aufrufer zurück.
12    static protected int cmp(String t,String p,int i,ArrayList<Integer> res,
13                             Heuristic ...heuristics) {
14      int nextI = i + 1, j = p.length() - 1;
15      // Vergleich von rechts (hinten) nach links (vorne) ...
16      while (t.charAt(i) == p.charAt(j) && j > 0) {
17        i = i - 1;
18        j = j - 1;
19      }
20      if ((j == 0) && (t.charAt(i) == p.charAt(0)) ) {
21        res.add(i);
22      }
23      // Alle Heuristiken durchgehen ...
24      for (Heuristic h : heuristics) {
25        int ih = h.Next(t, p, i, j);
26        if (ih > nextI) { // besserer Versatz?
27          nextI = ih;     // ja: übernehmen!
28        }
29      }
30
31      return nextI;
32    }
...
```

Implementierung - 2

```
32    // Sucht im Text t nach allen Vorkommen des Musters p. Zur Bestimmung der
33    // Versätze werden die übergebenen Heuristiken verwendet. Wird keine
34    // Heuristik übergeben, verhält sich der Algorithmus wie die einfache
35    // Textsuche!
36    public static ArrayList<Integer> apply(String t,String p, Heuristic ... h) {
37      ArrayList<Integer> res = new ArrayList<Integer>();
38      int i = p.length() - 1; // Position an der Musterende ausgerichtet wird
39
40      while (i < t.length()) {
41        i = cmp(t,p,i,res,h);
42      }
43
44      return result;
45    }
46  } // class BoyerMoore
```

Anmerkung

- In der Praxis Heuristiken besser direkt implementieren (ohne `interface`)

Die *Bad Charachter*-Heuristik - 1. Fall

- Anwendbar, wenn **erstes** verglichenes Zeichen **nicht** übereinstimmt

  Unterschied beim letzten Zeichen des Suchmusters

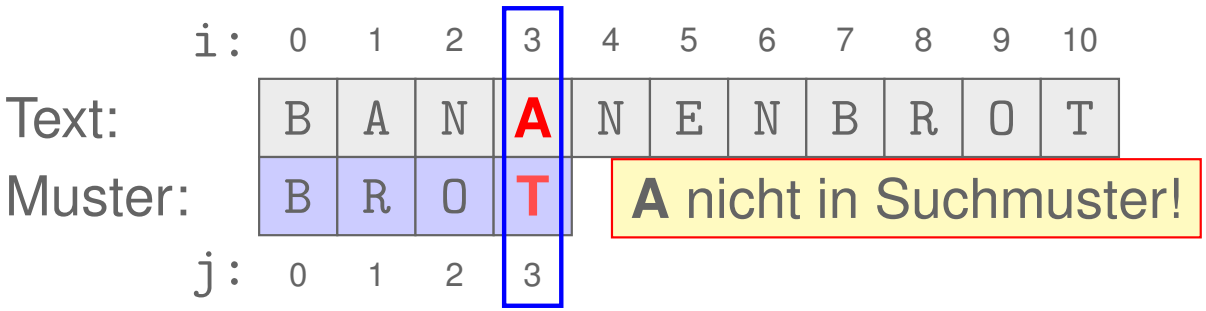
Die *Bad Charachter*-Heuristik - 1. Fall

- Anwendbar, wenn **erstes** verglichenes Zeichen **nicht** übereinstimmt
  Unterschied beim letzten Zeichen des Suchmusters
- Zwei Fälle sind denkbar:

Die *Bad Charachter*-Heuristik - 1. Fall

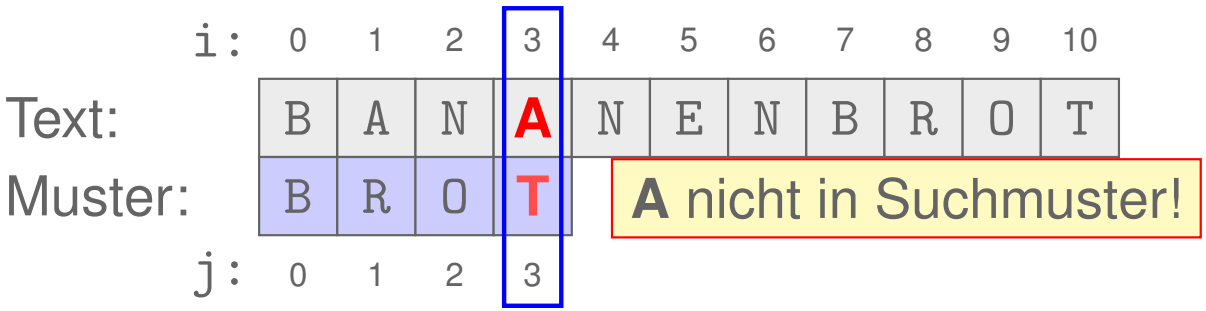
- Anwendbar, wenn **erstes** verglichenes Zeichen **nicht** übereinstimmt

  Unterschied beim letzten Zeichen des Suchmusters
- Zwei Fälle sind denkbar:
  1. Zeichen aus Text kommt **nicht in Suchmuster** vor:

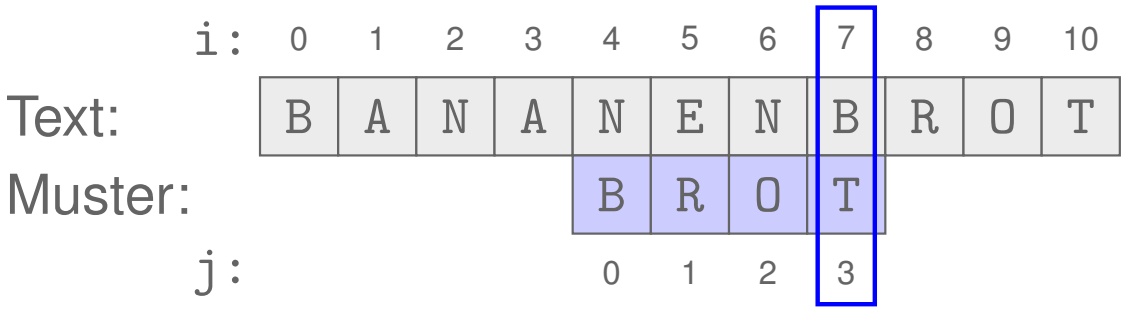


Die *Bad Charachter*-Heuristik - 1. Fall

- Anwendbar, wenn **erstes** verglichenes Zeichen **nicht** übereinstimmt
  Unterschied beim letzten Zeichen des Suchmusters
- Zwei Fälle sind denkbar:
  1. Zeichen aus Text kommt **nicht in Suchmuster** vor:



⇒ Verschiebe Muster um Musterlänge nach rechts:

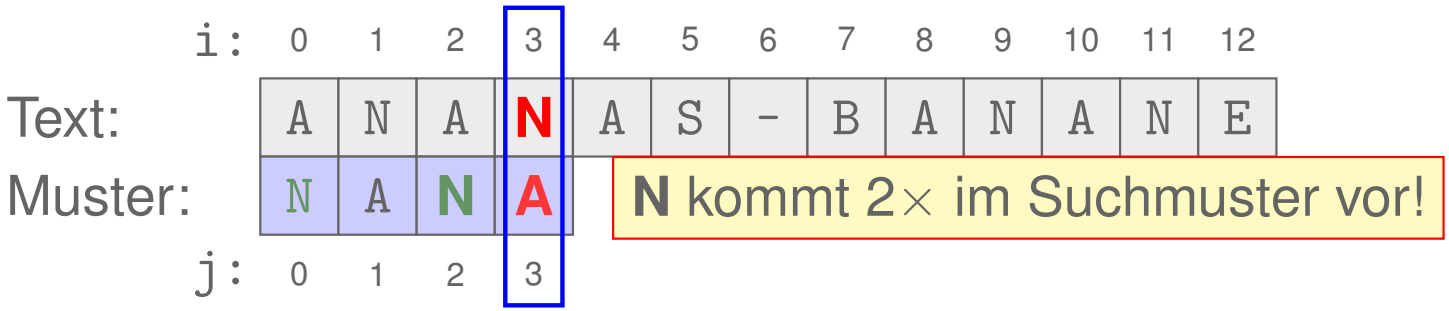


Die *Bad Charachter*-Heuristik - 2. Fall

- Anwendbar, wenn **erstes** verglichenes Zeichen **nicht** übereinstimmt
  Unterschied beim letzten Zeichen des Suchmusters
- Zwei Fälle sind denkbar:
  1. Zeichen aus Text kommt nicht in Suchmuster vor
  2. Zeichen aus Text (hier "N") ist **Teil des Suchmusters**:

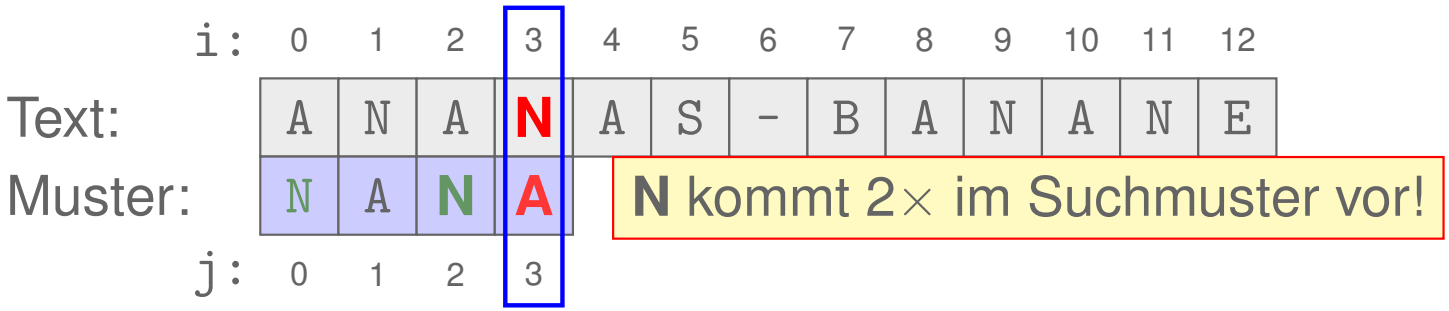
Die *Bad Charachter*-Heuristik - 2. Fall

- Anwendbar, wenn **erstes** verglichenes Zeichen **nicht** übereinstimmt
  Unterschied beim letzten Zeichen des Suchmusters
- Zwei Fälle sind denkbar:
  1. Zeichen aus Text kommt nicht in Suchmuster vor
  2. Zeichen aus Text (hier "N") ist **Teil des Suchmusters**:

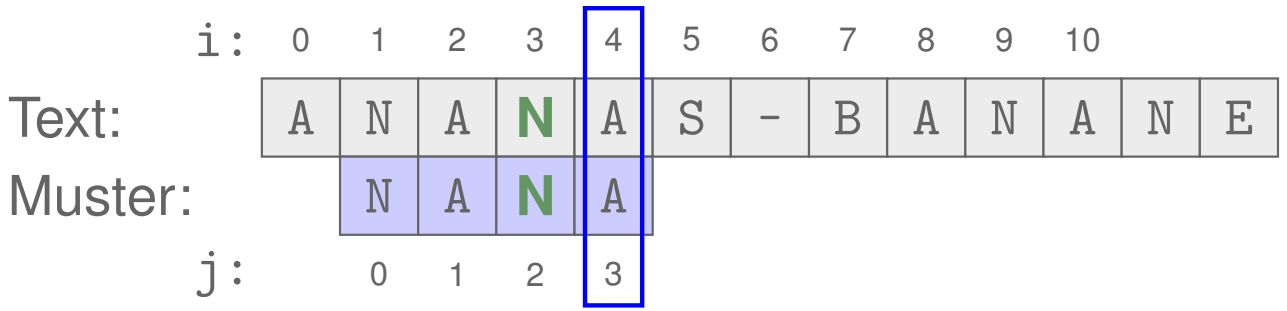


Die *Bad Charachter*-Heuristik - 2. Fall

- Anwendbar, wenn **erstes** verglichenes Zeichen **nicht** übereinstimmt
  Unterschied beim letzten Zeichen des Suchmusters
- Zwei Fälle sind denkbar:
  1. Zeichen aus Text kommt nicht in Suchmuster vor
  2. Zeichen aus Text (hier "N") ist **Teil des Suchmusters**:



⇒ Gehe im Muster nach links und suche die Position des ersten Vorkommens dieses Zeichens ("N"); daraus ergibt sich Versatz:



Die *Bad Charachter*-Heuristik

- Anwendbar, wenn **erstes** verglichenes Zeichen **nicht** übereinstimmt

  Unterschied beim letzten Zeichen des Suchmusters

- Zwei Fälle sind denkbar:
  1. Zeichen aus Text kommt nicht in Suchmuster vor

     $\Rightarrow$ Verschiebe Suchmuster um Musterlänge nach rechts:
  2. Zeichen aus Text ist Teil des Suchmusters

     $\Rightarrow$ Das letzte Vorkommen bestimmt, wie weit verschoben wird

- Implementierung: `skip`-Tabelle

  Stellen, um die Muster nach rechts verschoben werden muss oder Musterlänge, falls Zeichen nicht in Muster.

- Erweiterbar für Unterschiede bei anderem als dem letzten Zeichen

  $\Rightarrow$ Hausaufgabe (für 1. Fall leicht; 2. Fall mit Mehraufwand)

Die *Bad Character*-Heuristik - Beispiel

Beispiel 1.1 (`last` und `skip`).

Wir betrachten ein Alphabet Σ und das Muster `TOOTH`:

Muster:	T	O	O	T	H
j:	0	1	2	3	4

`last(TOOTH)`: $\Sigma \to \mathbb{N}_0$ liefert den letzten Index eines Buchstabens im Muster plus 1; für alle anderen Zeichen 0:

last			
t	o	h	$*$
4	3	5	0

`skip`$(p)(x) = \texttt{len}(p) - \texttt{last}(x)$:

skip			
t	o	h	$*$
1	2	0	5

Die *Bad Character*-Heuristik
Implementierung

```java
public class BadCharacterHeuristic implements Heuristic {
  // Um Platz zu sparen verwenden wir eine HashMap; ansonsten müssten wir nämlich ein
  // Feld für alle möglichen Zeichen anlegen ... der "*" in der Tabelle auf den Folien
  // spiegelt dann den Fall wieder, dass für ein Zeichen kein Eintrag in der skipTable
  // ist (get liefert dann null).
  private HashMap<Character, Integer> skipTable;
  private int m;
  BadCharacterHeuristic(String p) {
    skipTable = new HashMap<Character, Integer>();
    m = p.length();
    for (int i=0 ; i<m ; i++) { // i+1 = last
      skipTable.put(p.charAt(i), m-(i+1));
    }
  }
  // Berechnet skip mithilfe der skipTable
  protected int skip(char c) {
    Integer sk = skipTable.get(c);
    if (sk == null) { // Zeichen nicht in Muster?
      return m; // -> skip um Musterlänge
    }
    return sk.intValue();
  }
  public int Next(String t, String p, int i, int j) {
    if (j == m-1) { // letztes Zeichen im Muster?
      // Ja: Versatz ergibt sich aus skip
      return i + skip(t.charAt(i));
    }
    // Nein: die Heuristik greift nicht -> ein Zeichen weiter
    return i+1;
  }
}
```

Die *Bad Character*-Heuristik - Beispiel

Beispiel 1.2 (Suche mit der *Bad Character*-Heuristik).

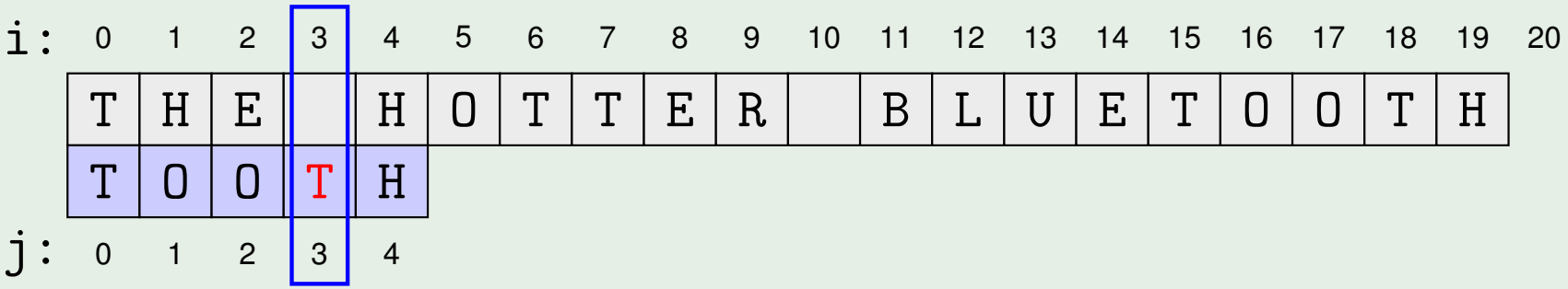
skip			
t	o	h	∗
1	2	0	5

i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	T	H	E		H	O	T	T	E	R		B	L	U	E	T	O	O	T	H	

Die *Bad Character*-Heuristik - Beispiel

Beispiel 1.2 (Suche mit der *Bad Character*-Heuristik).

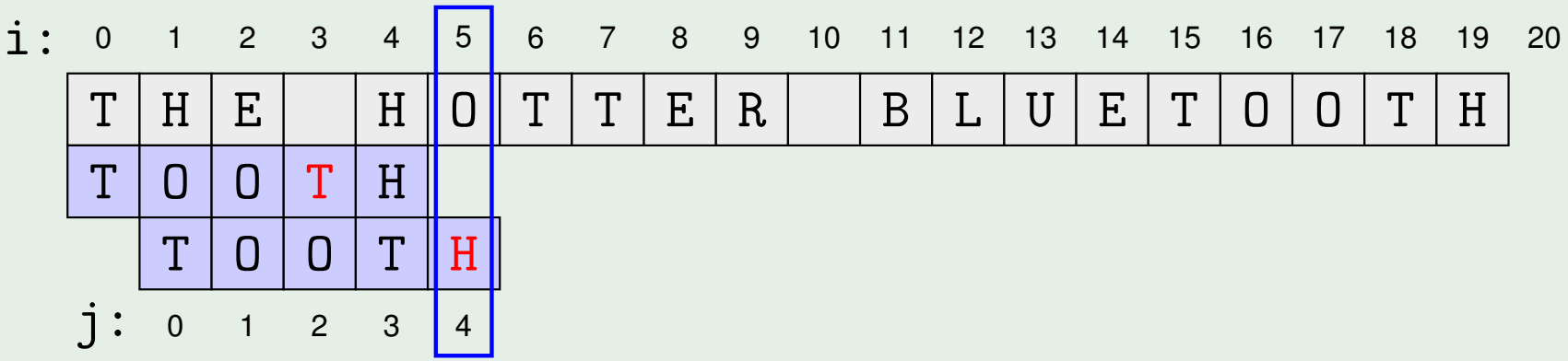
skip			
t	o	h	∗
1	2	0	5



Die *Bad Character*-Heuristik - Beispiel

Beispiel 1.2 (Suche mit der *Bad Character*-Heuristik).

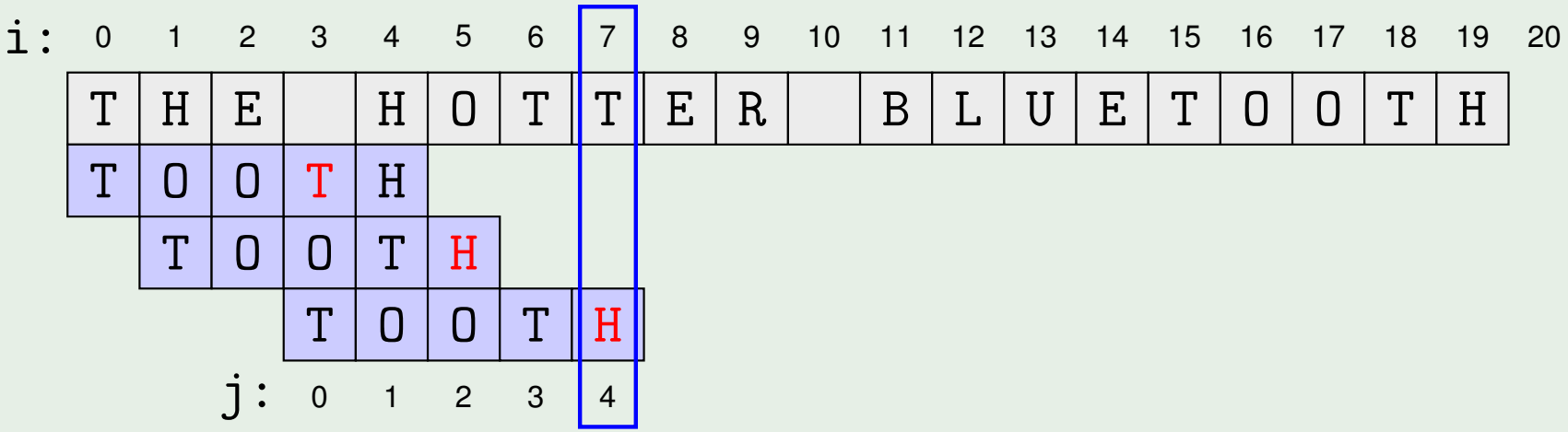
skip			
t	o	h	∗
1	2	0	5



Die *Bad Character*-Heuristik - Beispiel

Beispiel 1.2 (Suche mit der *Bad Character*-Heuristik).

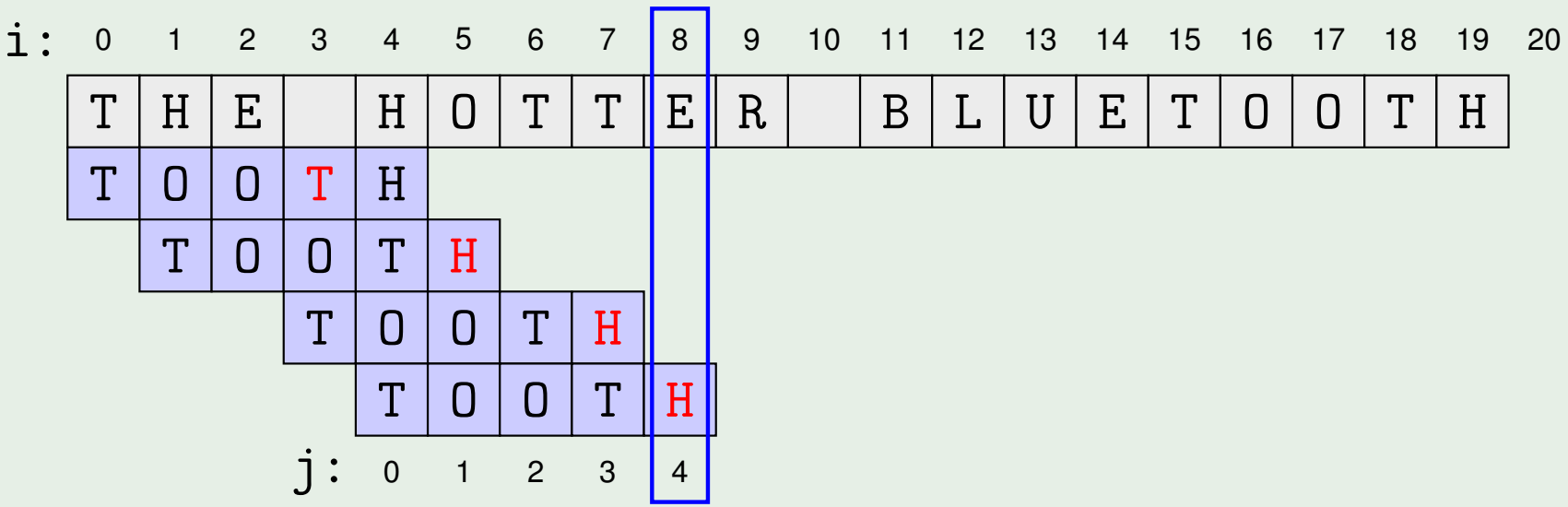
skip			
t	o	h	∗
1	2	0	5



Die *Bad Character*-Heuristik - Beispiel

Beispiel 1.2 (Suche mit der *Bad Character*-Heuristik).

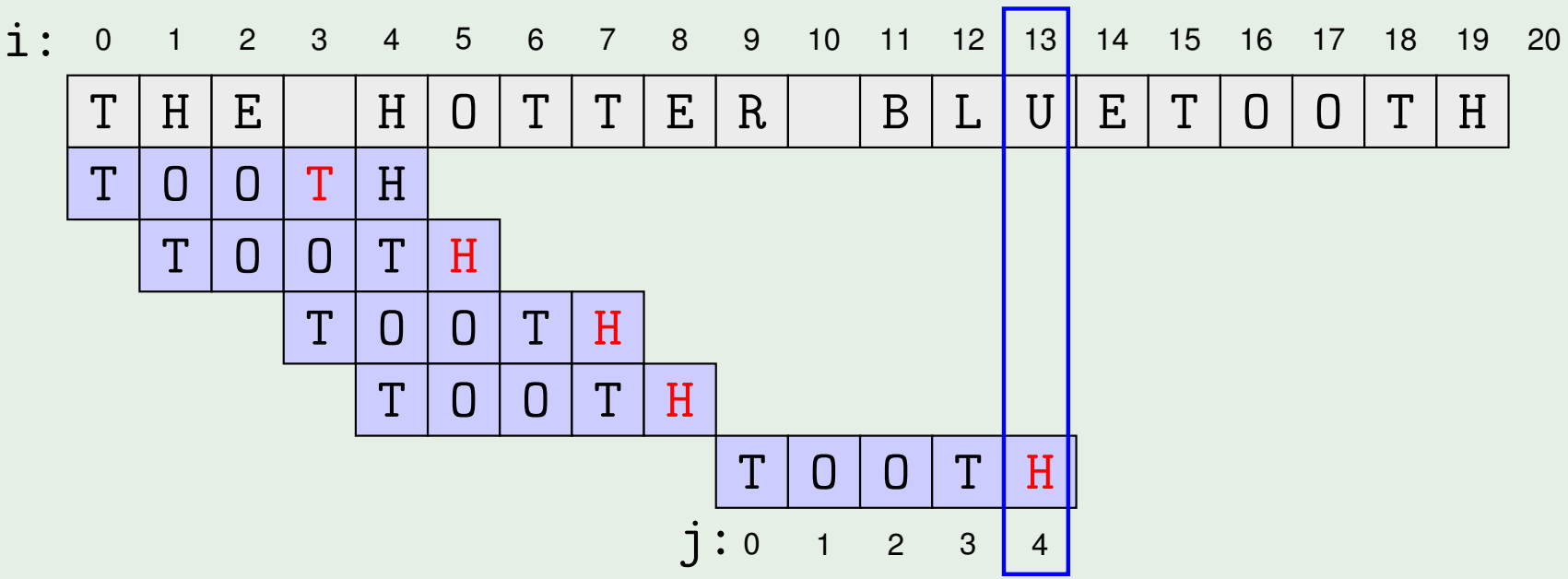
skip			
t	o	h	∗
1	2	0	5



Die *Bad Character*-Heuristik - Beispiel

Beispiel 1.2 (Suche mit der *Bad Character*-Heuristik).

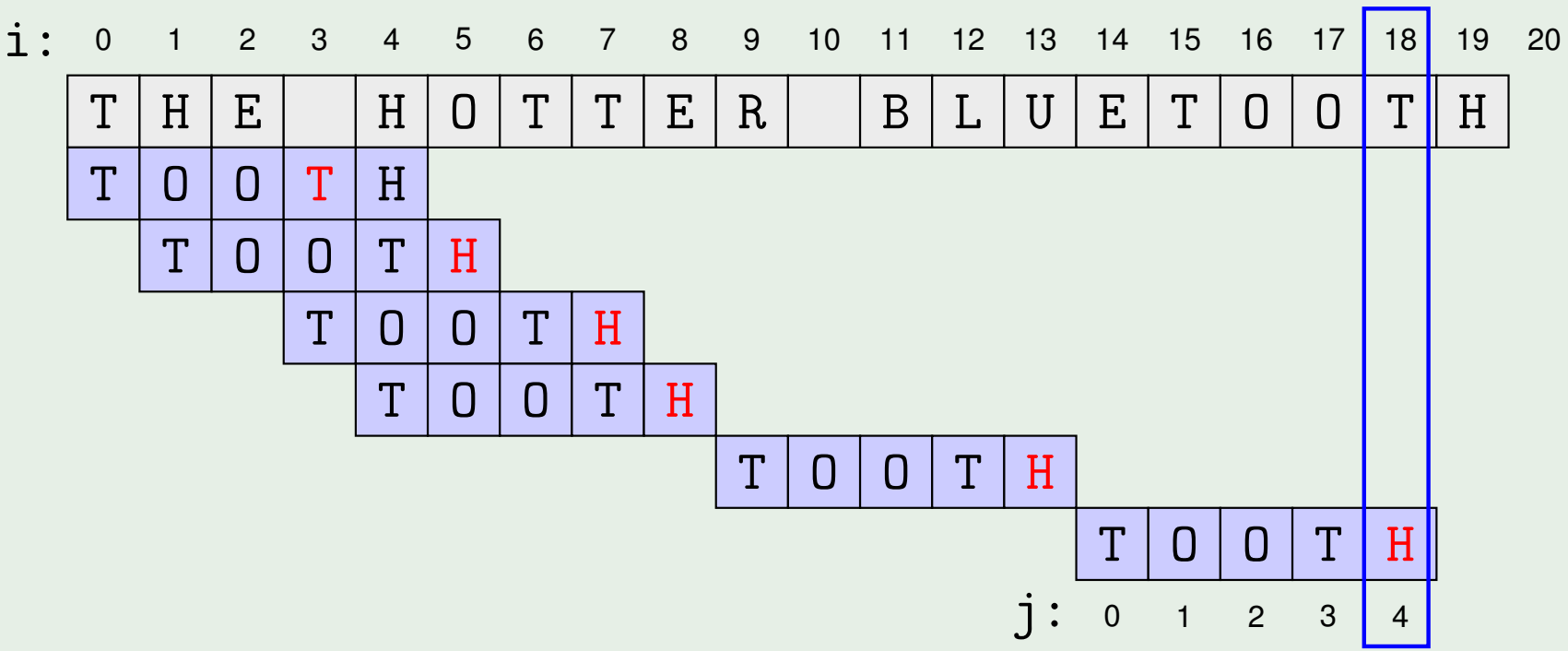
skip			
t	o	h	∗
1	2	0	5



Die *Bad Character*-Heuristik - Beispiel

Beispiel 1.2 (Suche mit der *Bad Character*-Heuristik).

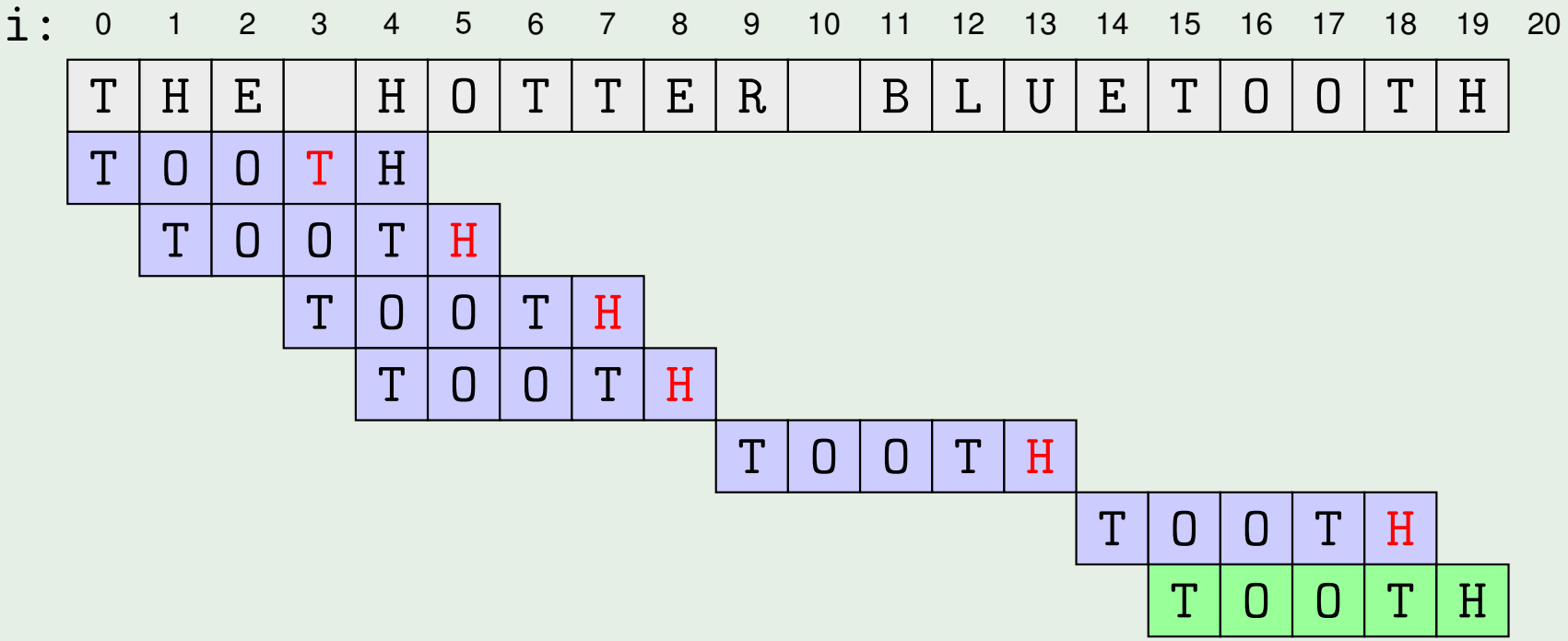
skip			
t	o	h	∗
1	2	0	5



Die *Bad Character*-Heuristik - Beispiel

Beispiel 1.2 (Suche mit der *Bad Character*-Heuristik).

skip			
t	o	h	∗
1	2	0	5



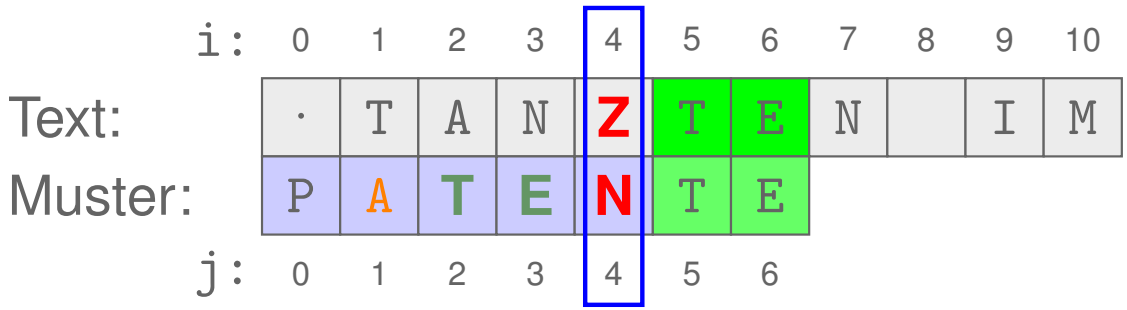
Die *Good Suffix*-Heuristik

- Behandelt den Fall, dass ein **echtes Suffix** des Suchmusters **erfolgreich abgeglichen** wurde, bevor es zu einer Ungleichheit kam.

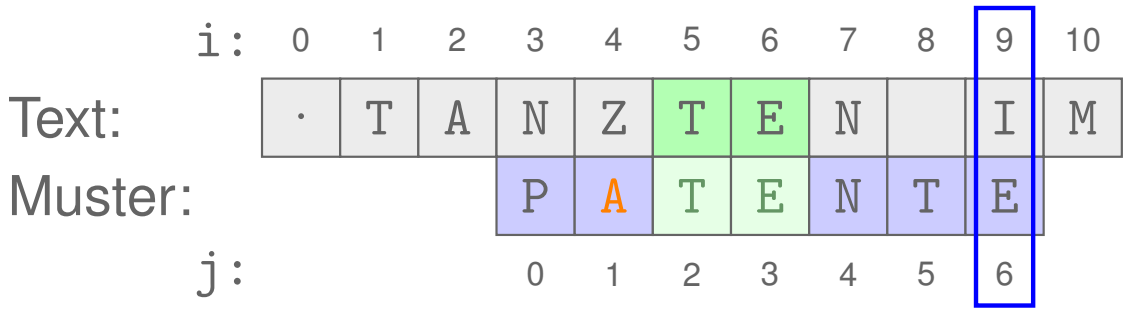
- Drei Fälle können unterschieden werden (s.n.F.)

Die *Good Suffix*-Heuristik - 1. Fall

- Im Groben[2] können drei Fälle unterschieden werden:
  1. Das übereinstimmende Suffix kommt mehrfach im Suchmuster vor:



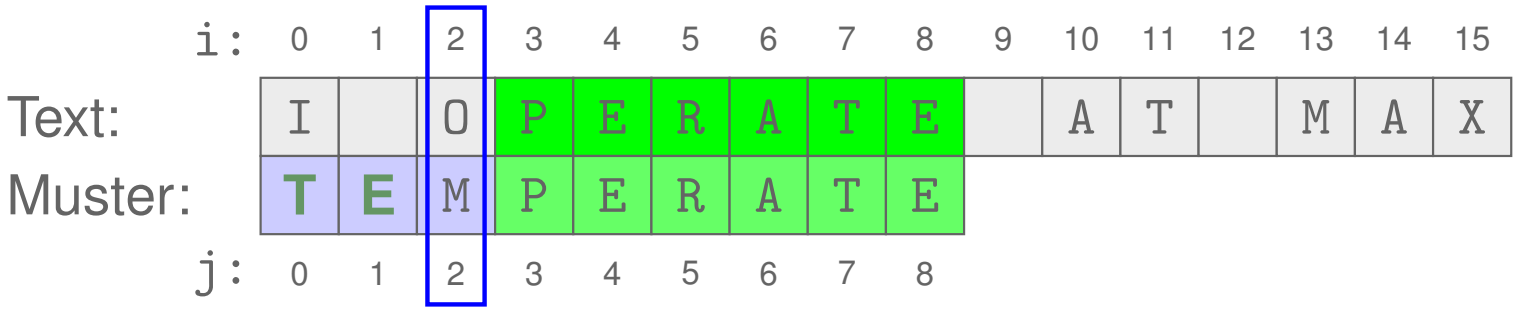
Ausgehend von der Position der Nichtübereinstimmung: gehe im Muster nach links und suche erstes Vorkommen des Suffixes, vor dem das letzte verglichene Zeichen des Musters (hier "N") **nicht** steht.



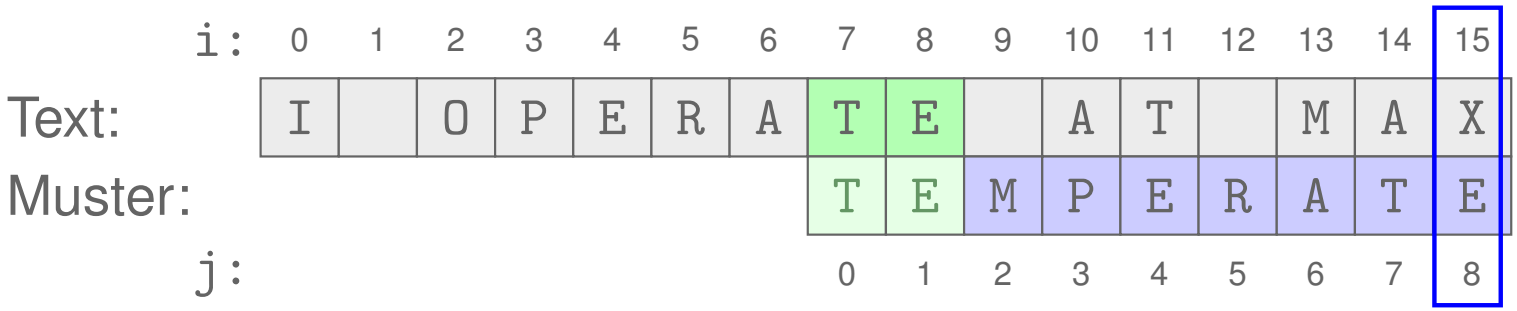
[2] es geht zunächst um die Idee!

Die *Good Suffix*-Heuristik - 2. Fall

- Im Groben[3] können drei Fälle unterschieden werden:
  1. Das übereinstimmende Suffix kommt mehrfach im Suchmuster vor.
  2. Ein Präfix des Musters ist Suffix (hier "TE") des übereinstimmenden Teils:



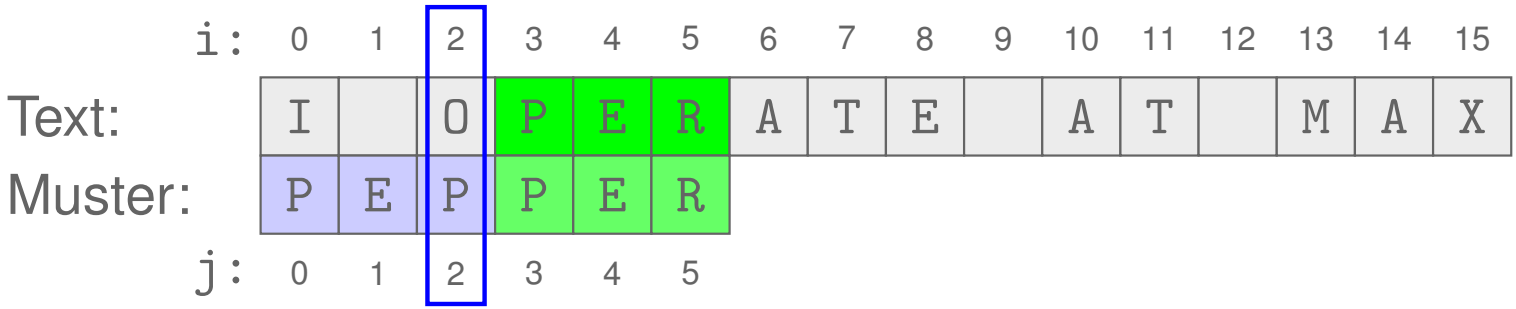
Die Länge des **längsten Präfixes** bestimmt den Versatz:



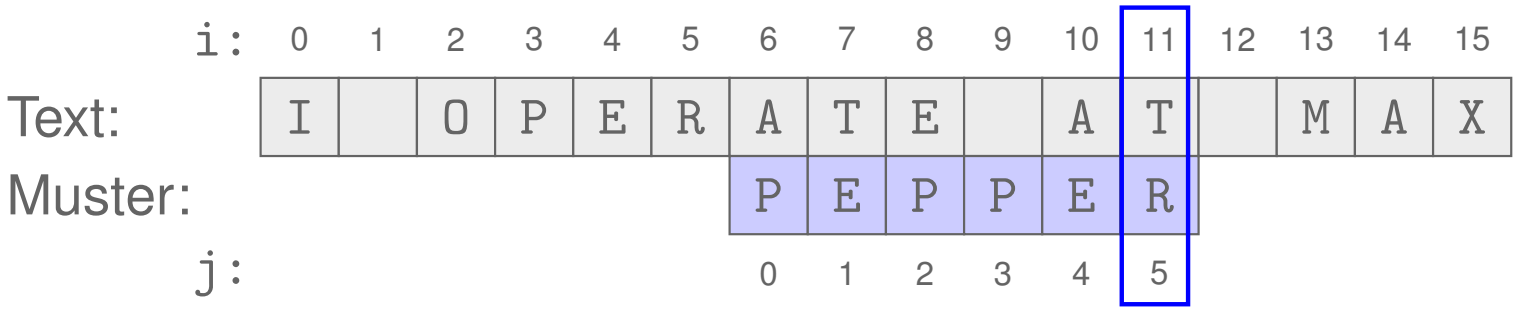
[3] es geht zunächst um die Idee!

Die *Good Suffix*-Heuristik - 3. Fall

- Im Groben[4] können drei Fälle unterschieden werden:
  1. Das übereinstimmende Suffix kommt mehrfach im Suchmuster vor.
  2. Ein Präfix des Musters ist Suffix des übereinstimmenden Teils.
  3. Kein Suffix des übereinstimmenden Teils kommt wiederholt im Muster vor:



Das Muster wird um eine **Musterlänge** nach rechts geschoben:



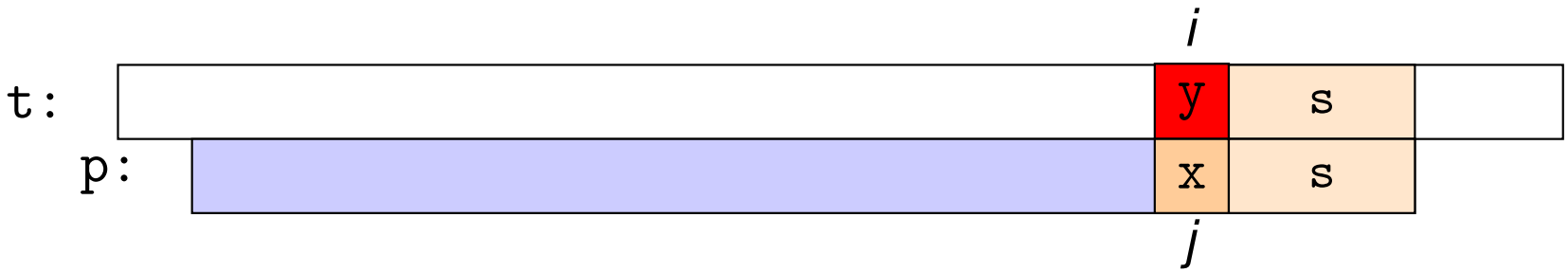
[4] es geht zunächst um die Idee!

Feld `shift` wird in 2 Phasen berechnet:

```
1  public class GoodSuffixHeuristic implements Heuristic {
2    protected int shift[];
3    protected int bStart[]; // Start-Index der Ränder (Suffix)
4    protected int m;        // Länge des Musters
5
6    protected void initPhase1(String p) {
7      // behandelt 1. Fall
8    }
9
10   protected void initPhase2() {
11     // behandelt 2. und 3. Fall
12   }
13
14   GoodSuffixHeuristic(String p) {
15     m = p.length();
16     bStart = new int[m+1];
17     shift = new int[m+1];
18     initPhase1(p);
19     initPhase2();
20     bStart = null; // <- wird nicht mehr gebraucht
21   }
22
23   public int Next(String t,String p,int i,int j) {
24     return shift[j+1];
25   }
26 }
```

Good Suffix-Heuristik: Implementierung 1

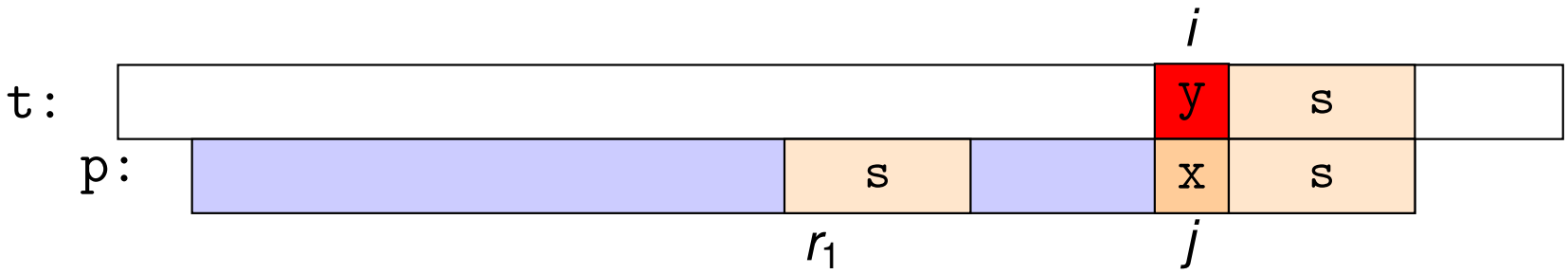
1. Fall: Die Situation stellt sich so dar:



- **Mismatch**: $\mathtt{y} = \mathtt{t}[i] \neq \mathtt{p}[j] = \mathtt{x}$
- Suche im Muster links von $j + 1$ nach Wiederholung von s

Good Suffix-Heuristik: Implementierung 1

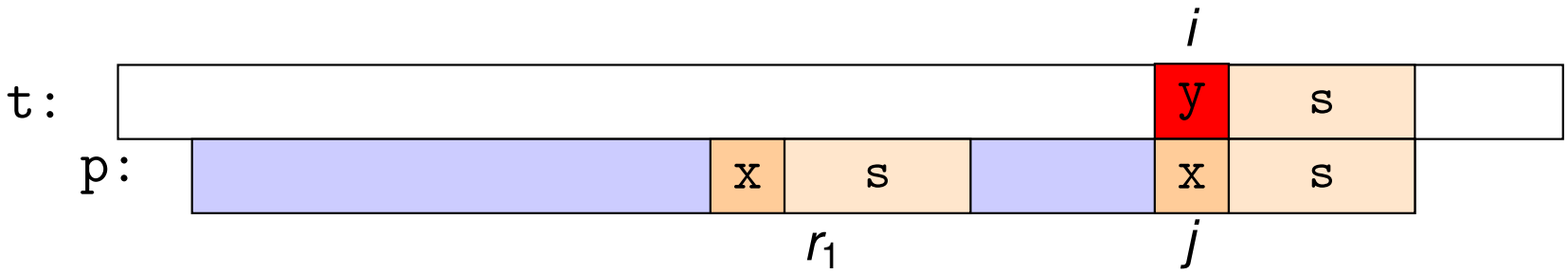
1. Fall: Die Situation stellt sich so dar:



- **Mismatch**: $\mathtt{y} = \mathtt{t}[i] \neq \mathtt{p}[j] = \mathtt{x}$
- Suche im Muster links von $j + 1$ nach Wiederholung von s
  - finde r_1

Good Suffix-Heuristik: Implementierung 1

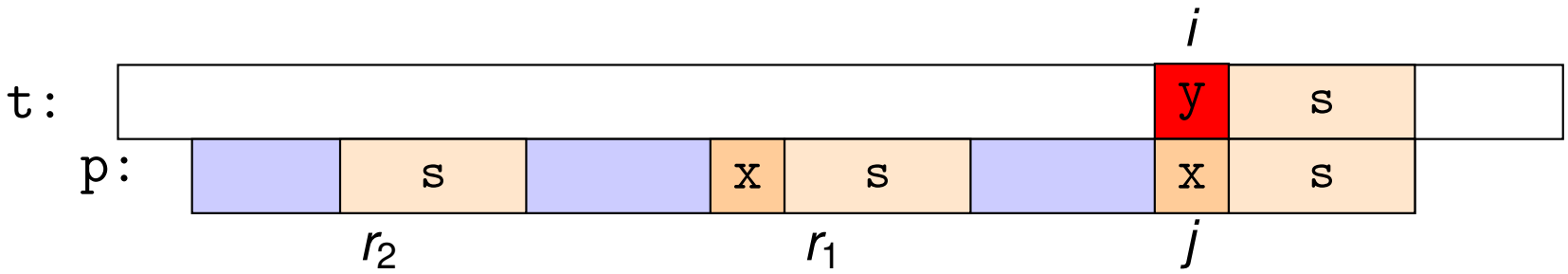
1. Fall: Die Situation stellt sich so dar:



- **Mismatch**: $\mathtt{y} = \mathtt{t}[i] \neq \mathtt{p}[j] = \mathtt{x}$
- Suche im Muster links von $j + 1$ nach Wiederholung von s
  - finde r_1; betrachte Zeichen davor (hier x). Verschieben würde zum gleichen Mismatch führen $\Rightarrow$ weitersuchen ...
    Anmerkung: Ränder von s wurden in früheren Iterationen behandelt.

Good Suffix-Heuristik: Implementierung 1

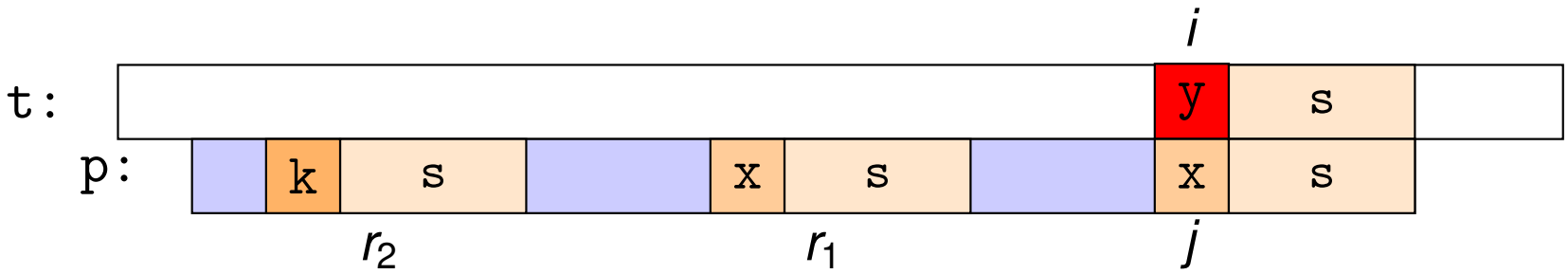
1. Fall: Die Situation stellt sich so dar:



- **Mismatch**: $\mathtt{y} = \mathtt{t}[i] \neq \mathtt{p}[j] = \mathtt{x}$
- Suche im Muster links von $j + 1$ nach Wiederholung von s
  - finde r_1; betrachte Zeichen davor (hier x). Verschieben würde zum gleichen Mismatch führen $\Rightarrow$ weitersuchen ...
    Anmerkung: Ränder von s wurden in früheren Iterationen behandelt.
  - finde r_2

Good Suffix-Heuristik: Implementierung 1

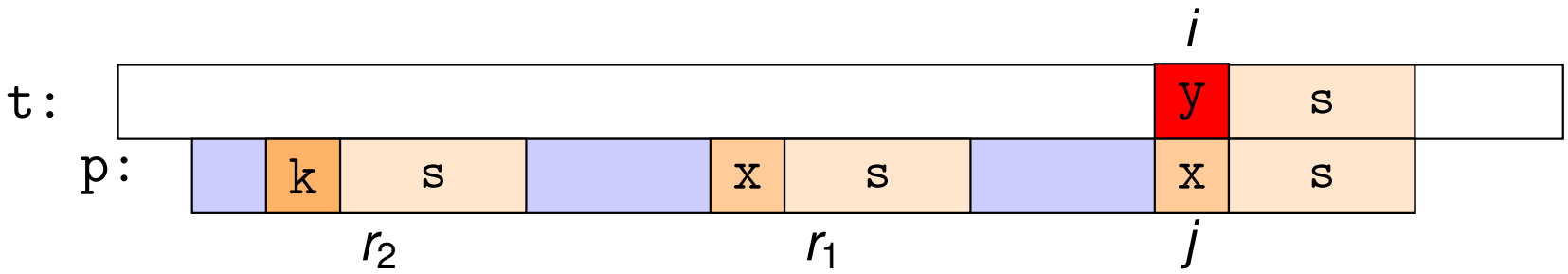
1. Fall: Die Situation stellt sich so dar:



- **Mismatch**: $\mathtt{y} = \mathtt{t}[i] \neq \mathtt{p}[j] = \mathtt{x}$
- Suche im Muster links von $j + 1$ nach Wiederholung von s
  - finde r_1; betrachte Zeichen davor (hier x). Verschieben würde zum gleichen Mismatch führen ⇒ weitersuchen ...
    Anmerkung: Ränder von s wurden in früheren Iterationen behandelt.
  - finde r_2; Zeichen davor könnte zum Text passen, daher wählen wir den entsprechenden Versatz!

Good Suffix-Heuristik: Implementierung 1

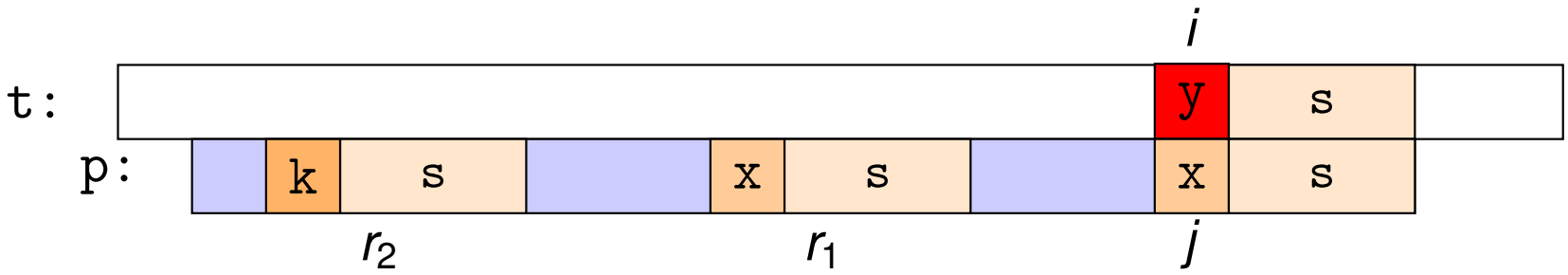
1. Fall: Die Situation stellt sich so dar:



- **Mismatch**: $\mathtt{y} = \mathtt{t}[i] \neq \mathtt{p}[j] = \mathtt{x}$
- Suche im Muster links von $j + 1$ nach Wiederholung von s
  - finde r_1; betrachte Zeichen davor (hier x). Verschieben würde zum gleichen Mismatch führen $\Rightarrow$ weitersuchen ...
    Anmerkung: Ränder von s wurden in früheren Iterationen behandelt.
  - finde r_2; Zeichen davor könnte zum Text passen, daher wählen wir den entsprechenden Versatz!
- **Beachte:** s ist **Rand** von
  - $\mathtt{p}[r_1]...\mathtt{p}[m-1]$ und
  - $\mathtt{p}[r_2]...\mathtt{p}[m-1]$

Good Suffix-Heuristik: Implementierung 1

1. Fall: Die Situation stellt sich so dar:

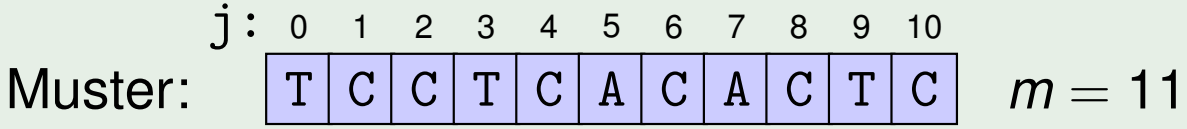


- **Mismatch**: $\mathtt{y} = \mathtt{t}[i] \neq \mathtt{p}[j] = \mathtt{x}$
- Suche im Muster links von $j + 1$ nach Wiederholung von s
  - finde r_1; betrachte Zeichen davor (hier x). Verschieben würde zum gleichen Mismatch führen $\Rightarrow$ weitersuchen ...
    Anmerkung: Ränder von s wurden in früheren Iterationen behandelt.
  - finde r_2; Zeichen davor könnte zum Text passen, daher wählen wir den entsprechenden Versatz!
- **Beachte:** s ist **Rand** von
  - $\mathtt{p}[r_1]...\mathtt{p}[m-1]$ und
  - $\mathtt{p}[r_2]...\mathtt{p}[m-1]$

Phase 1: Berechne für alle Suffixe von p den maximalen Rand (analog zu KMP) ; falls ein Rand "passt", merke Versatz in `shift`$[j]$, sofern `shift`$[j]$ noch nicht initialisiert.

Berechnung von `shift` - 1

Beispiel 1.3 (Berechnung von `shift` - Phase 1).



j	Suffix	Randlänge	shift	Anmerkung
11	ε	0	1	Mismatch 1. Zeichen v.r.

Berechnung von `shift` - 1

Beispiel 1.3 (Berechnung von `shift` - Phase 1).

j:	0	1	2	3	4	5	6	7	8	9	10
Muster:	T	C	C	T	C	A	C	A	C	T	C

$m = 11$

j	Suffix	Randlänge	shift	Anmerkung
11	ε	0	1	Mismatch 1. Zeichen v.r.
10	C	0	-	

Berechnung von `shift` - 1

Beispiel 1.3 (Berechnung von `shift` - Phase 1).

j:	0	1	2	3	4	5	6	7	8	9	10
Muster:	T	C	C	T	C	A	C	A	C	T	C

$m = 11$

j	Suffix	Randlänge	shift	Anmerkung
11	ε	0	1	Mismatch 1. Zeichen v.r.
10	C	0	-	
9	TC	0	-	

Berechnung von `shift` - 1

$\texttt{shift}[m - \text{Randlänge}] = m - j + \text{Randlänge}$

Beispiel 1.3 (Berechnung von `shift` - Phase 1).

j:	0	1	2	3	4	5	6	7	8	9	10
Muster:	T	C	C	T	C	A	C	A	C	T	C

$m = 11$

j	Suffix	Randlänge	shift	Anmerkung
11	ε	0	1	Mismatch 1. Zeichen v.r.
10	C	0	2	
9	TC	0	-	
8	ACTC	1	-	setze `shift[10]`

Berechnung von `shift` - 1

$\texttt{shift}[m - \text{Randlänge}] = m - j + \text{Randlänge}$

Beispiel 1.3 (Berechnung von `shift` - Phase 1).

j:	0	1	2	3	4	5	6	7	8	9	10
Muster:	T	C	C	T	C	A	C	A	C	T	C

$m = 11$

j	Suffix	Randlänge	shift	Anmerkung
11	ε	0	1	Mismatch 1. Zeichen v.r.
10	C	0	2	
9	TC	0	-	
8	A C T C	1	-	setze `shift[10]`
7	ACTC	0	-	

Berechnung von `shift` - 1

$\texttt{shift}[m - \text{Randlänge}] = m - j + \text{Randlänge}$

Beispiel 1.3 (Berechnung von `shift` - Phase 1).

j:	0	1	2	3	4	5	6	7	8	9	10
Muster:	T	C	C	T	C	A	C	A	C	T	C

$m = 11$

j	Suffix	Randlänge	shift	Anmerkung
11	ε	0	1	Mismatch 1. Zeichen v.r.
10	C	0	2	
9	TC	0	-	
8	A C T C	1	-	setze `shift[10]`
7	ACTC	0	-	
6	A C ACT C	1	-	`shift[10]` belegt!

Berechnung von `shift` - 1

$\texttt{shift}[m - \text{Randlänge}] = m - j + \text{Randlänge}$

Beispiel 1.3 (Berechnung von `shift` - Phase 1).

j:	0	1	2	3	4	5	6	7	8	9	10
Muster:	T	C	C	T	C	A	C	A	C	T	C

$m = 11$

j	Suffix	Randlänge	shift	Anmerkung
11	ε	0	1	Mismatch 1. Zeichen v.r.
10	C	0	2	
9	TC	0	-	
8	A C T C	1	-	setze `shift[10]`
7	ACTC	0	-	
6	A C ACT C	1	-	`shift[10]` belegt!
5	ACACTC	0	-	

Berechnung von `shift` - 1

$\texttt{shift}[m - \text{Randlänge}] = m - j + \text{Randlänge}$

Beispiel 1.3 (Berechnung von `shift` - Phase 1).

j:	0	1	2	3	4	5	6	7	8	9	10
Muster:	T	C	C	T	C	A	C	A	C	T	C

$m = 11$

j	Suffix	Randlänge	shift	Anmerkung
11	ε	0	1	Mismatch 1. Zeichen v.r.
10	C	0	2	
9	TC	0	-	
8	A C T C	1	-	setze `shift[10]`
7	ACTC	0	-	
6	A C ACT C	1	-	`shift[10]` belegt!
5	ACACTC	0	-	
4	T C ACACT C	1	-	ungeeignet

Berechnung von `shift` - 1

$\texttt{shift}[m - \text{Randlänge}] = m - j + \text{Randlänge}$

Beispiel 1.3 (Berechnung von `shift` - Phase 1).

j:	0	1	2	3	4	5	6	7	8	9	10
Muster:	T	C	C	T	C	A	C	A	C	T	C

$m = 11$

j	Suffix	Randlänge	shift	Anmerkung
11	ε	0	1	Mismatch 1. Zeichen v.r.
10	C	0	2	
9	TC	0	-	
8	A C T C	1	-	setze `shift[10]`
7	ACTC	0	-	
6	A C ACT C	1	-	`shift[10]` belegt!
5	ACACTC	0	-	
4	T C ACACT C	1	-	ungeeignet
3	C TC ACAC TC	2	-	ungeeignet

Berechnung von `shift` - 1

$\texttt{shift}[m - \text{Randlänge}] = m - j + \text{Randlänge}$

Beispiel 1.3 (Berechnung von `shift` - Phase 1).

j:	0	1	2	3	4	5	6	7	8	9	10
Muster:	T	C	C	T	C	A	C	A	C	T	C

$m = 11$

j	Suffix	Randlänge	shift	Anmerkung
11	ε	0	1	Mismatch 1. Zeichen v.r.
10	C	0	2	
9	TC	0	-	
8	A C T C	1	6	setze `shift[10]`
7	ACTC	0	-	
6	A C ACT C	1	-	`shift[10]` belegt!
5	ACACTC	0	-	
4	T C ACACT C	1	-	ungeeignet
3	C TC ACAC TC	2	-	ungeeignet
2	C CTC ACA CTC	3	-	setze `shift[8]`

Berechnung von `shift` - 1

$\texttt{shift}[m - \text{Randlänge}] = m - j + \text{Randlänge}$

Beispiel 1.3 (Berechnung von `shift` - Phase 1).

j:	0	1	2	3	4	5	6	7	8	9	10
Muster:	T	C	C	T	C	A	C	A	C	T	C

$m = 11$

j	Suffix	Randlänge	shift	Anmerkung
11	ε	0	1	Mismatch 1. Zeichen v.r.
10	C	0	2	
9	TC	0	-	
8	A C T C	1	6	setze `shift[10]`
7	ACTC	0	-	
6	A C ACT C	1	-	`shift[10]` belegt!
5	ACACTC	0	-	
4	T C ACACT C	1	-	ungeeignet
3	C TC ACAC TC	2	-	ungeeignet
2	C CTC ACA CTC	3	-	setze `shift[8]`
1	T C CTCACACT C	1	-	ungeeignet

Berechnung von `shift` - 1

$\texttt{shift}[m - \text{Randlänge}] = m - j + \text{Randlänge}$

Beispiel 1.3 (Berechnung von `shift` - Phase 1).

j:	0	1	2	3	4	5	6	7	8	9	10
Muster:	T	C	C	T	C	A	C	A	C	T	C

$m = 11$

j	Suffix	Randlänge	shift	Anmerkung
11	ε	0	1	Mismatch 1. Zeichen v.r.
10	C	0	2	
9	TC	0	-	
8	A C T C	1	6	setze `shift[10]`
7	ACTC	0	-	
6	A C ACT C	1	-	`shift[10]` belegt!
5	ACACTC	0	-	
4	T C ACACT C	1	-	ungeeignet
3	C TC ACAC TC	2	-	ungeeignet
2	C CTC ACA CTC	3	-	setze `shift[8]`
1	T C CTCACACT C	1	-	ungeeignet
0	TC CTCACAC TC	2	-	*nur Randber.* $\rightarrow$ *Phase 2*

Initialisierung von `shift` - Phase 1

Implementierung - Teil 2

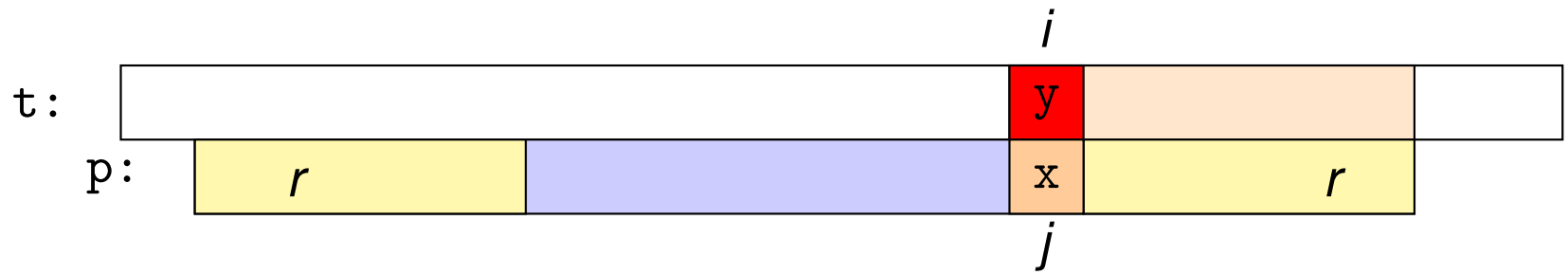
```
1  public class GoodSuffixHeuristic implements Heuristic {
2    protected int shift[];
3    protected int bStart[]; // Start-Index der Ränder (Suffix)
4    protected int m;        // Länge des Musters
5
6    protected void initPhase1(String p) {
7      int i=m, j=m+1; // m+1 -> virtuelles Epsilon am Ende von p
8      bStart[i] = j;
9      while (i>0) {
10       // Ab zweitem Zeichen von rechts (j<=m): suche links nach erstem Zeichen,
11       // welches den Rand *nicht* verlängert
12       while (j<=m && p.charAt(i-1)!=p.charAt(j-1)) {
13         // Es handelt sich um einen gültigen shift; wir übernehmen
14         // ihn aber nur, wenn er der am weitesten rechts stehende
15         // (shift[j]=0) ist!
16         if (shift[j]==0)
17           shift[j]=j-i;
18         j=bStart[j]; // Index des Zeichens vor nächst kleinerem Rand (Suffix)
19       }
20       // Hier gilt p[i-1] == p[j-1] -> Rand konnte verlängert werden
21       i--; j--;
22       bStart[i]=j;
23     }
24   }
25   ...
26 }
```

Beachte: Die Randlänge wird hier hier nicht gespeichert. In `bStart` wird der Start-Index des hinteren Teils eines Randes abgelegt; z.B. für `ab` `x` `ab` der Index 3, was die Implementierung erleichtert.

Good Suffix-Heuristik: Implementierung 2

2. Fall:

a) *Mismatch* **unmittelbar vor** Musterrand von r:

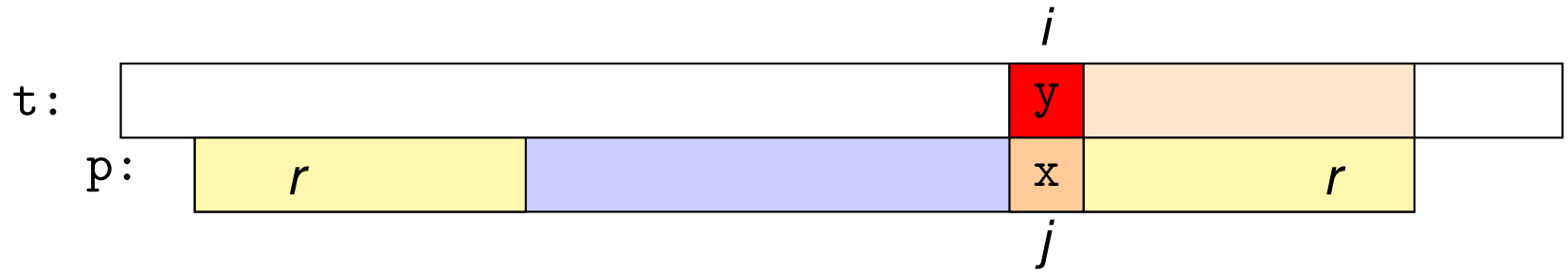


$\Rightarrow$ `shift` ergibt sich aus Länge Randes r (in Phase 1 berechnet).

Good Suffix-Heuristik: Implementierung 2

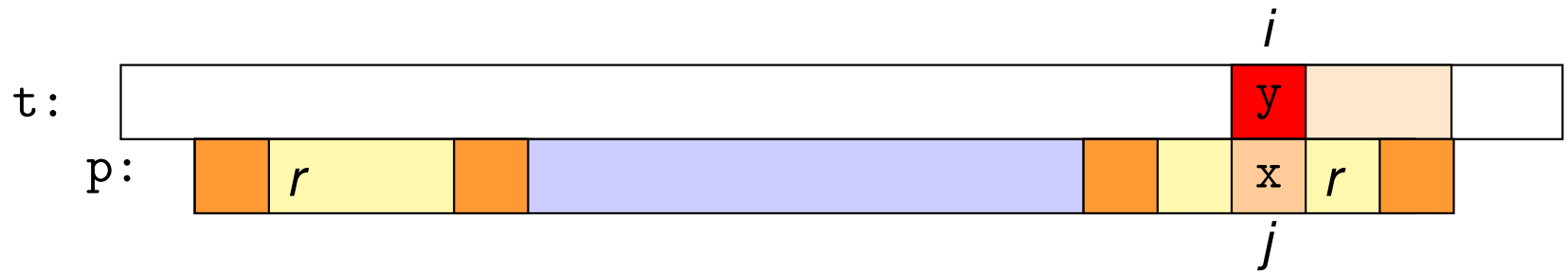
2. Fall:

a) *Mismatch* **unmittelbar vor** Musterrand von r:



$\Rightarrow$ `shift` ergibt sich aus Länge Randes r (in Phase 1 berechnet).

b) *Mismatch* **innerhalb** des maximalen Randes r:

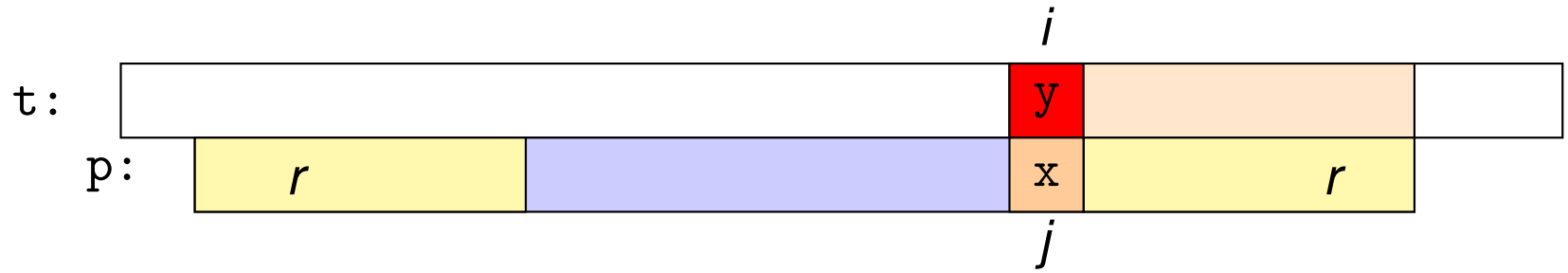


$\Rightarrow$ `shift` ergibt sich aus längstem Rand des Musterrandes r, der rechts von j liegt

Good Suffix-Heuristik: Implementierung 2

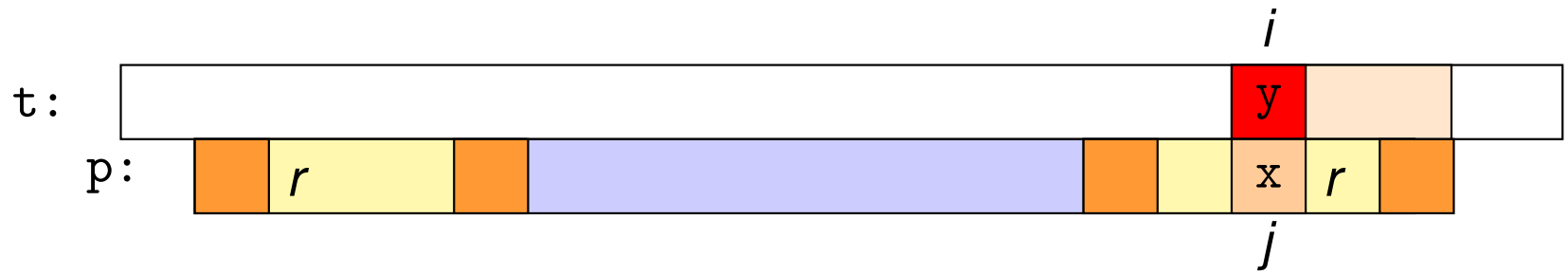
2. Fall:

a) *Mismatch* **unmittelbar vor** Musterrand von r:



$\Rightarrow$ `shift` ergibt sich aus Länge Randes r (in Phase 1 berechnet).

b) *Mismatch* **innerhalb** des maximalen Randes r:



$\Rightarrow$ `shift` ergibt sich aus längstem Rand des Musterrandes r, der rechts von j liegt

Stellen, an denen `shift` noch nicht initialisiert bilden wir auf 2a) ab.

Initialisierung von `shift` - Phase 2

Implementierung - Teil 3

```
1   public class GoodSuffixHeuristic implements Heuristic {
2     protected int shift[];
3     protected int bStart[]; // Start-Index der Ränder (Suffix)
4     protected int m;        // Länge des Musters
5
6     ...
7
8     protected void initPhase2() {
9       int i, j;
10      j=bStart[0]; // Grösster Rand des gesamten Musters
11      for (i=0; i<=m; i++) {
12        if (shift[i]==0) // shift noch nicht initialisiert?
13          shift[i]=j;    // Benutze aktuellen Rand!
14        if (i==j)        // Randgrenze erreicht?
15          j=bStart[j];   // Dann nächst kleineren Rand nehmen
16      }
17    }
18
19    ...
20  }
```

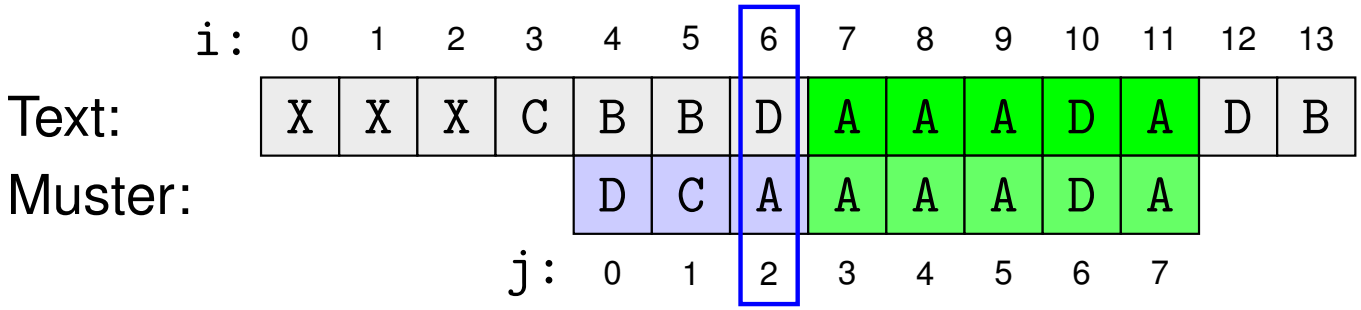
Warum *Bad Character* nur für letztes Zeichen?

Im Original-Papier von Boyer und Moore heißt es:

> Therefore unless *char* matchtes the last character of *pat* we can move past $delta_1$ characters of *string* without looking at the characters skipped;

δ_1 entspricht dabei unserer Funktion `skip`.

Die Ausweitung der *Bad Character*-Heuristik auf andere Positionen könnte ohne Anpassungen zu Fehlern führem:



Im Sinne der *Bad Character*-Heuristik darf das Muster hier zwei Stellen nach rechts verschoben weden; aber es ist $\texttt{last}(\texttt{D}) = 7$ und $m = 8$, woraus $\texttt{skip}(D) = 1$ folgt! Somit ergobt sich $\texttt{i} = 7$ und das Muster würde effektiv **nach links** bewegt wird!

Analyse:

- Im *worst case* benötigt der Algorithmus $\Theta(n \cdot m)$ Vergleiche.
- Ist das Alphabet groß und das Muster recht kurz, sorgt die *Bad Character*-Heuristig für eine Komplexität im *average case* von $\Theta(\frac{n}{m})$.
- Die *Good Suffix*-Heuristik bringt Vorteile bei kleinem Alphabet (z.B. Bio-Informatik: A,C,G,T).

Ausblick:

- *Bad Character*-Heuristik erweiterbar!
- R.N. Horspool und D.M. Sunday schlagen Varianten vor, bei denen auf *Good Suffix* verzichtet wird.