Lecture # 11 – Pattern Matching

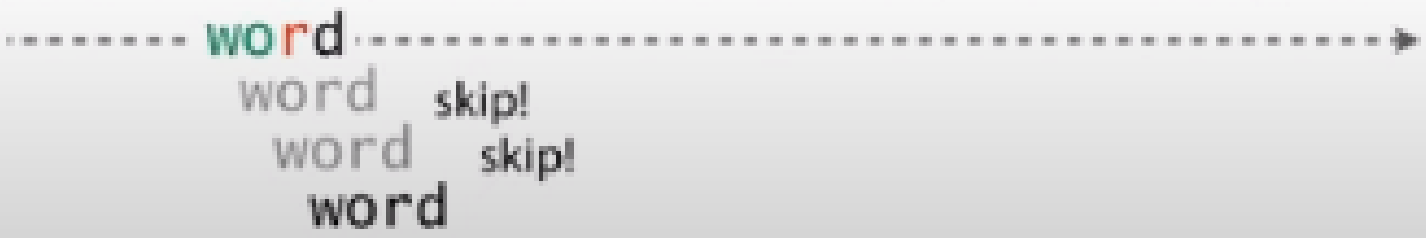
Boyer-Moore & KMP

Exact matching: better naïve algorithm

P: word
T: There would have been a time for such a word

u doesn't occur in *P*, so we can skip next two alignments

P: word
T: There would have been a time for such a word



Pattern Matching Algorithms

- If you want to search a piece of text for a target substring instead of a pattern, there are faster approaches.

- Brute-force Approach
  - The most obvious strategy is to loop over all the characters in the text and see if the target is at each position.
  - The pseudo-code in next slide shows this brute-force approach:
  - In this algorithm, variable i loops over the length of the text.
  - For each value of i, the variable j loops over the length of the target.
  - If the text has length N and the target has length M, the total run time is O(N × M).
  - This is simpler than using an NFA, but it's still not very efficient.

Pattern Matching Algorithms

- Brute-force Approach (contd.)

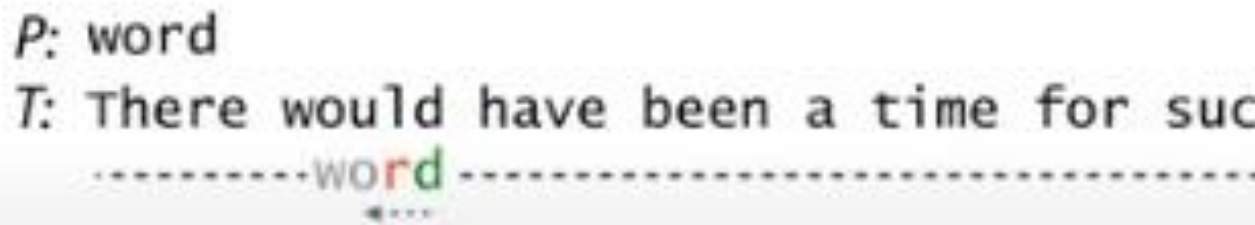
```
// Return the position of the target in the text.
Integer: FindTarget(String: text, String: target)
   For i = 0 To <last index of string>
      // See if the target begins at position i.
      Boolean: found_it = True
      For j = 0 To <last index of target>
         If (text[i + j] != target[j]) Then found_it = False
      Next j

      // See if we found the target.
      If (found_it) Then Return i
   Next i
   // If we got here, the target isn't present.
   Return -1
End FindTarget
```

Boyer-Moore

Learn from character comparisons to skip pointless alignments

Try alignments in left-to-right order, and try character comparisons in right-to-left order



Boyer-Moore: Bad character rule

Upon mismatch, skip alignments until (a) mismatch becomes a match, or (b) *P* moves past mismatched character

Step 1:

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

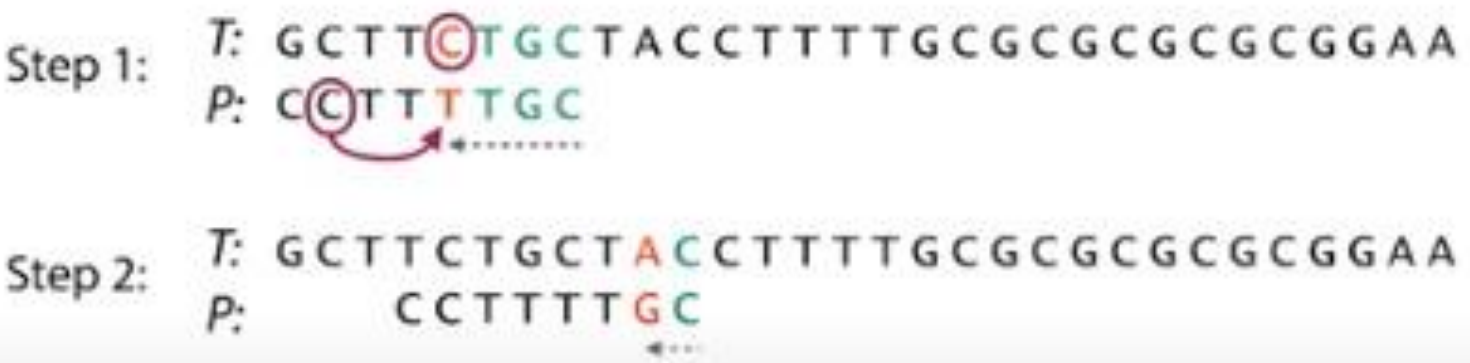
P: C C T T T T G C

Boyer-Moore: Bad character rule

Upon mismatch, skip alignments until (a) mismatch becomes a match, or (b) *P* moves past mismatched character

Boyer-Moore: Bad character rule

Upon mismatch, skip alignments until (a) mismatch becomes a match, or (b) *P* moves past mismatched character



Boyer-Moore: Bad character rule

Upon mismatch, skip alignments until (a) mismatch becomes a match, or (b) *P* moves past mismatched character

Boyer-Moore: Bad character rule

Upon mismatch, skip alignments until (a) mismatch becomes a match, or (b) *P* moves past mismatched character

Boyer-Moore: Good suffix rule

Let *t* = substring matched by inner loop; skip until (a) there are no mismatches between *P* and *t or* (b) *P* moves past *t*

Step 1:

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

P: C T T A C T T A C

Boyer-Moore: Good suffix rule

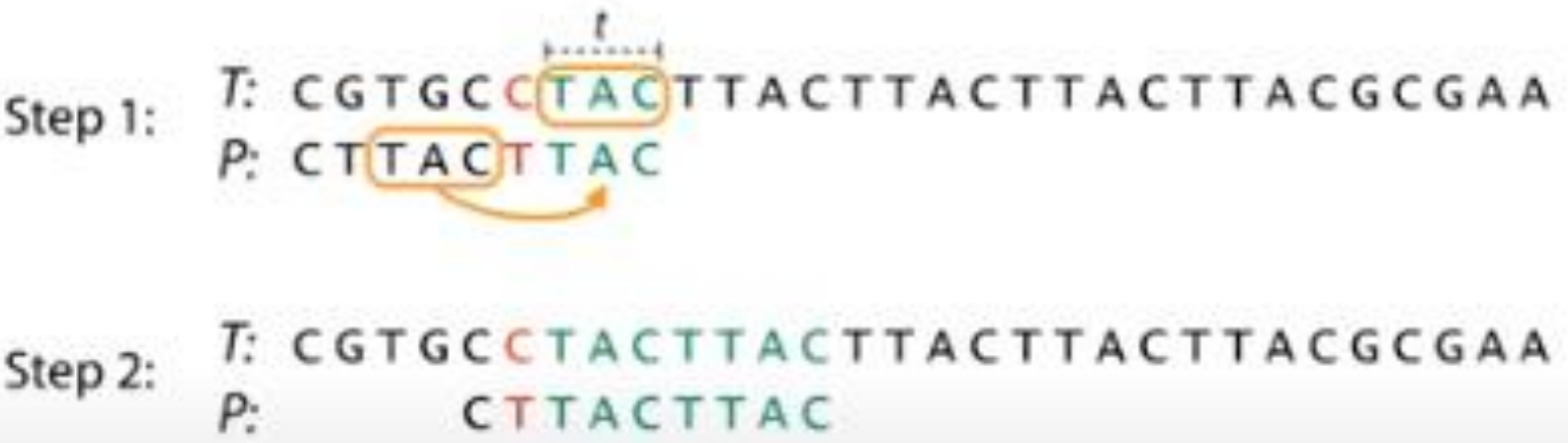
Let t = substring matched by inner loop; skip until (a) there are no mismatches between P and t *or* (b) P moves past t

Boyer-Moore: Good suffix rule

Let t = substring matched by inner loop; skip until (a) there are no mismatches between P and t *or* (b) P moves past t

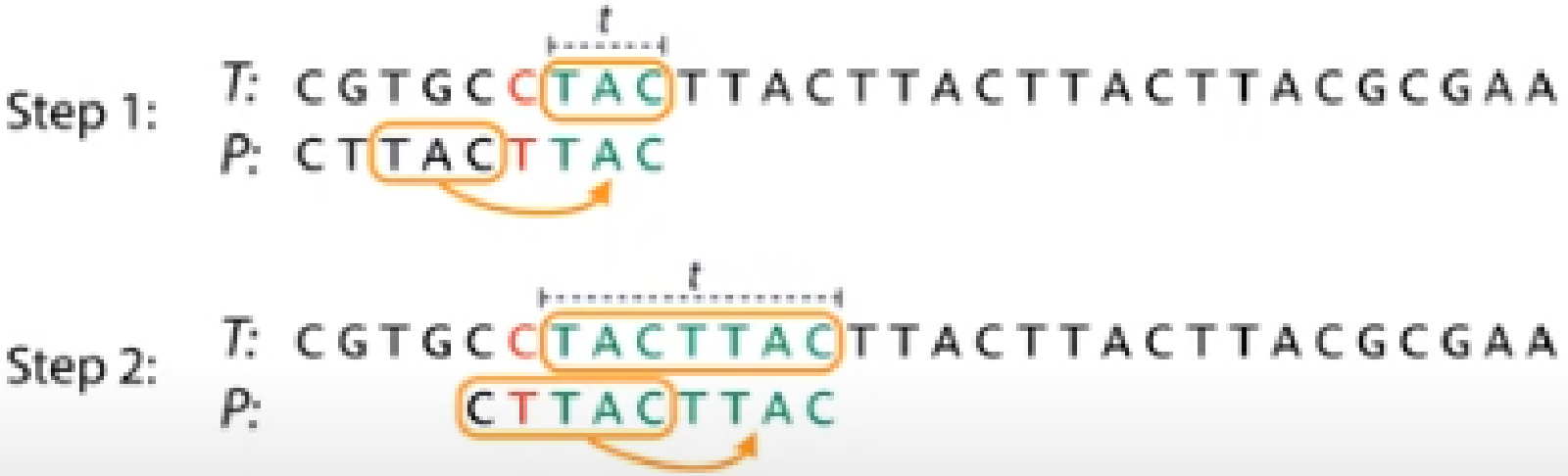
Boyer-Moore: Good suffix rule

Let t = substring matched by inner loop; skip until (a) there are no mismatches between P and t *or* (b) P moves past t



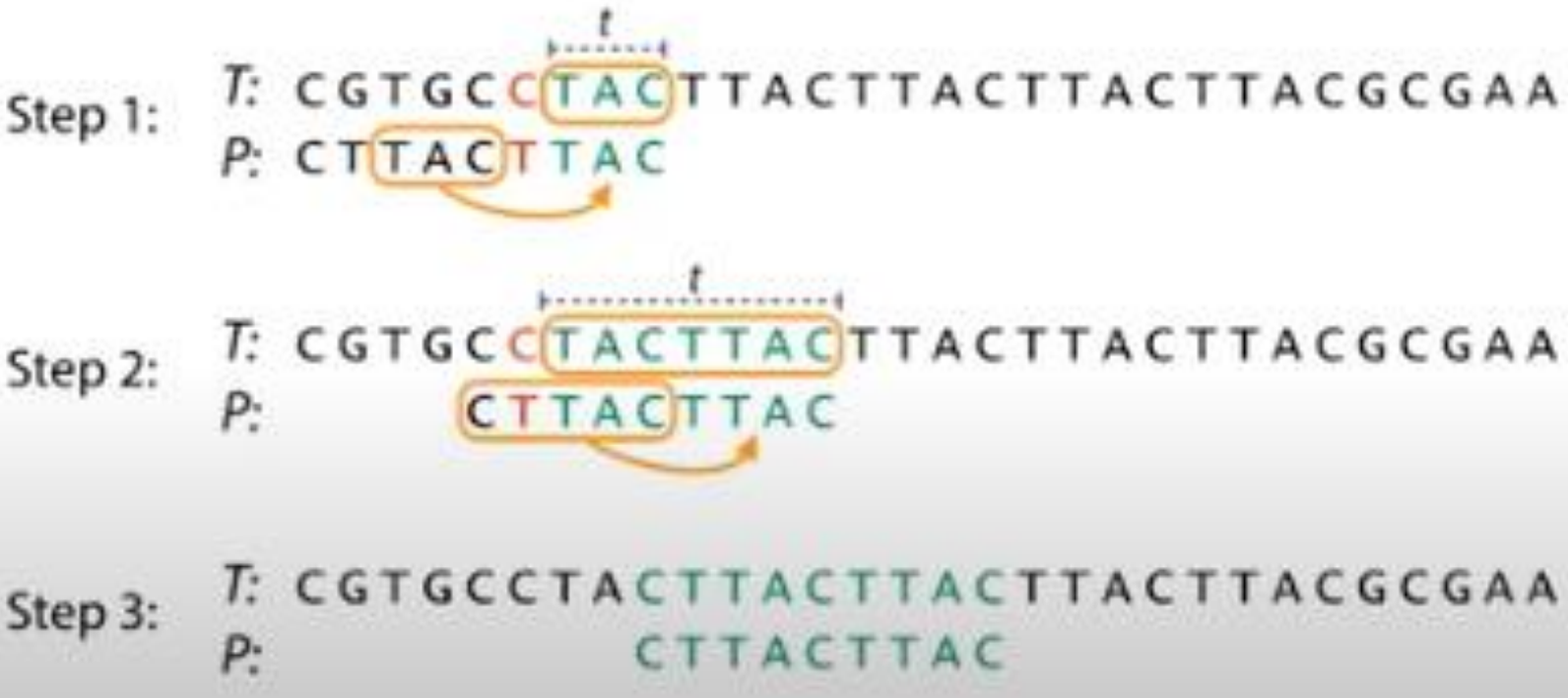
Boyer-Moore: Good suffix rule

Let t = substring matched by inner loop; skip until (a) there are no mismatches between P and t *or* (b) P moves past t



Boyer-Moore: Good suffix rule

Let t = substring matched by inner loop; skip until (a) there are no mismatches between P and t *or* (b) P moves past t



Boyer-Moore: Putting it together

Use bad character or good suffix rule, *whichever skips more*

Step 1:

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

P: G T A G C G G C G

Boyer-Moore: Putting it together

Use bad character or good suffix rule, *whichever skips more*

Step 1:

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

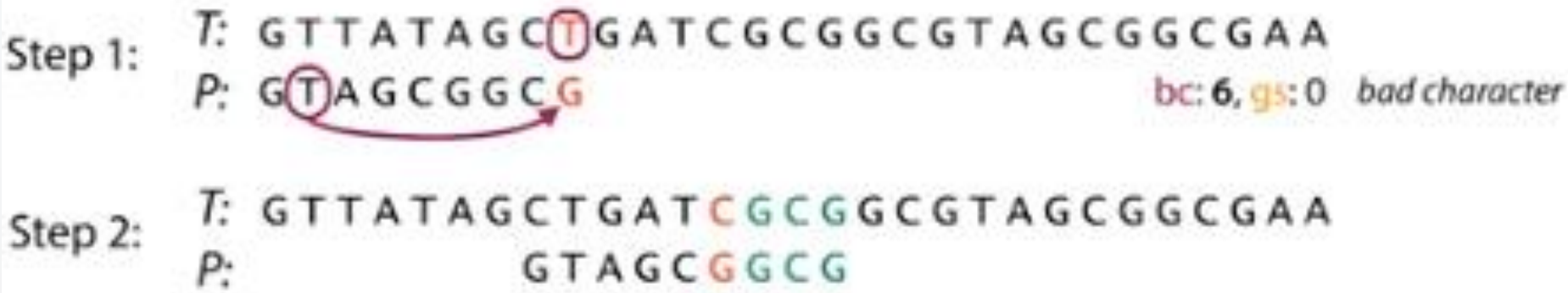
P: G T A G C G G C G

Boyer-Moore: Putting it together

Use bad character or good suffix rule, *whichever skips more*

Boyer-Moore: Putting it together

Use bad character or good suffix rule, *whichever skips more*

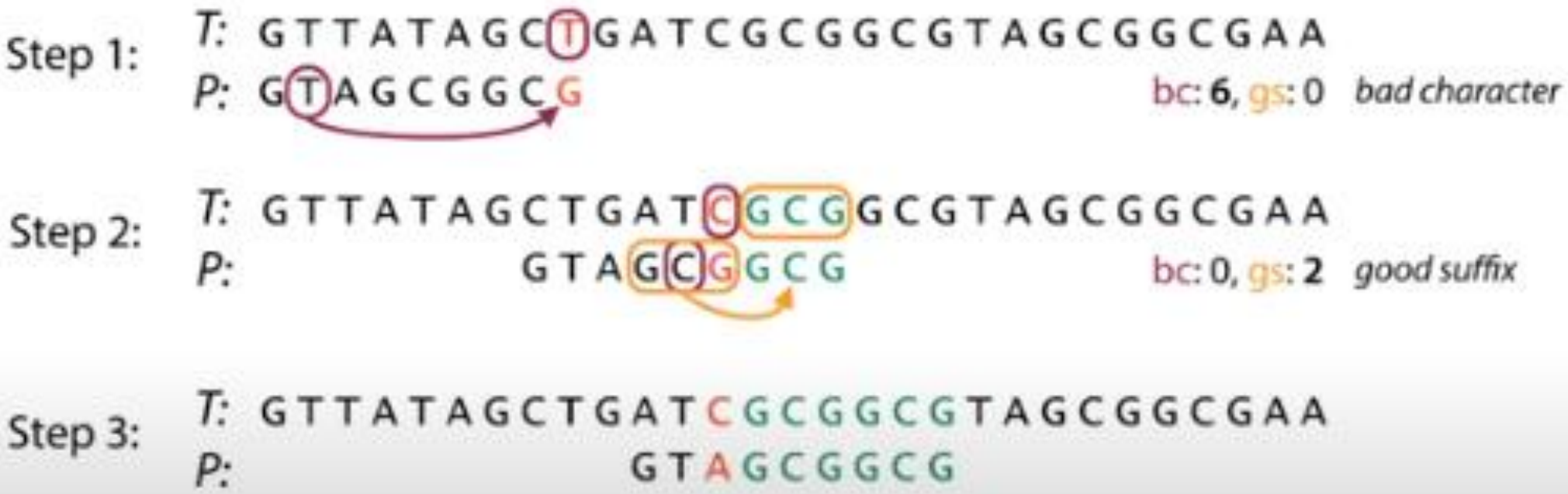


Boyer-Moore: Putting it together

Use bad character or good suffix rule, *whichever skips more*

Boyer-Moore: Putting it together

Use bad character or good suffix rule, *whichever skips more*

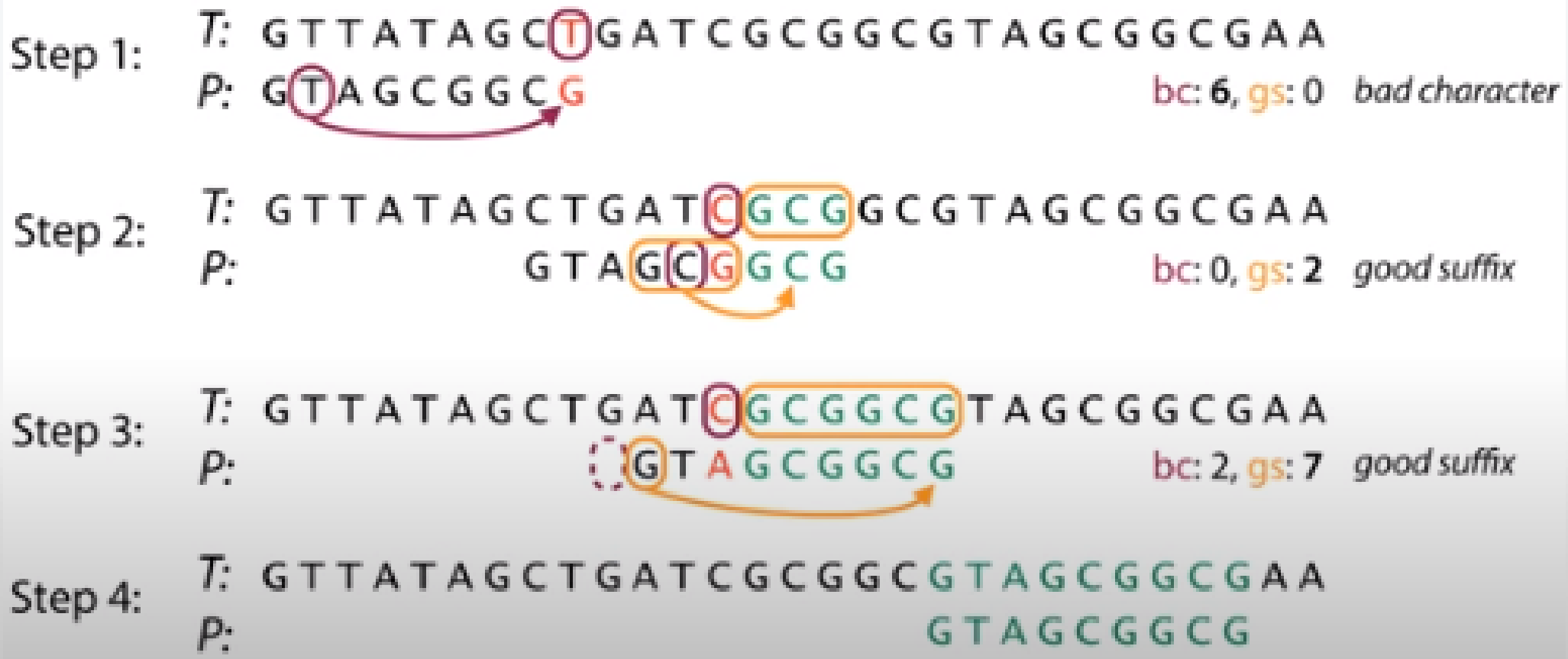


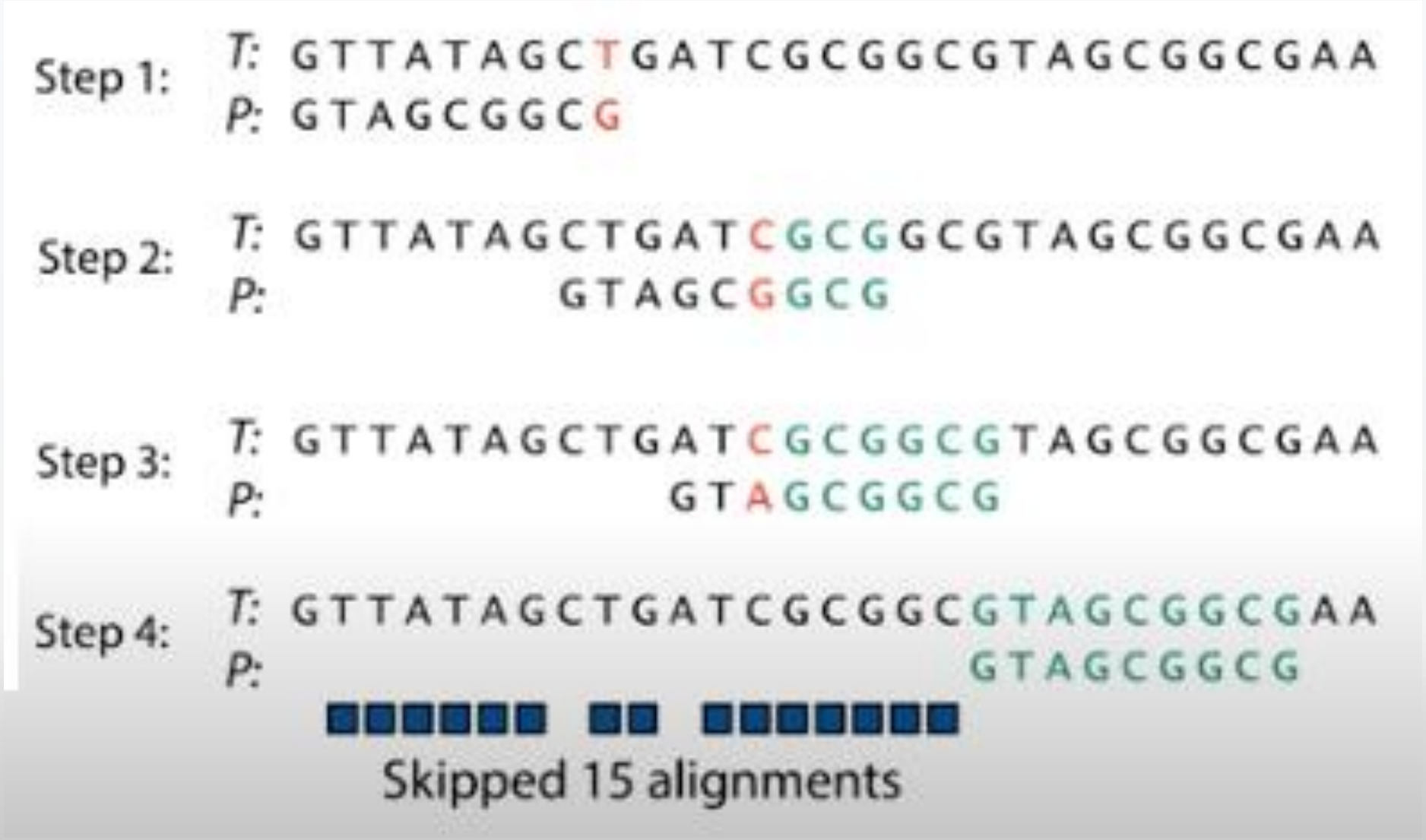
Boyer-Moore: Putting it together

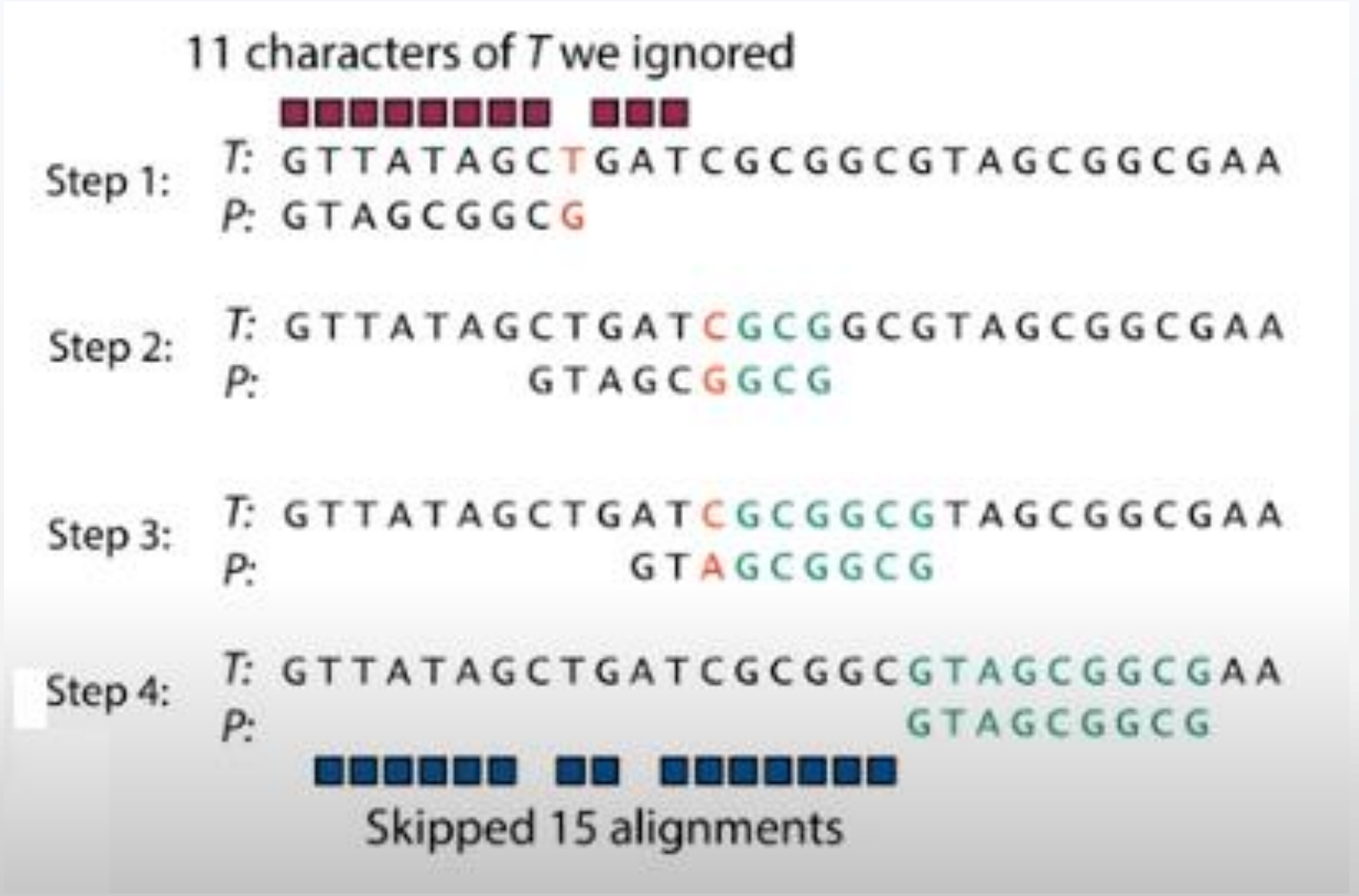
Use bad character or good suffix rule, *whichever skips more*

Boyer-Moore: Putting it together

Use bad character or good suffix rule, *whichever skips more*




Step 1:
T: GTTATAGCTGATCGCGGCGTAGCGGCGAA
P: GTAGCGGCG
Step 2:
T: GTTATAGCTGATCGCGGCGTAGCGGCGAA
P: GTAGCGGCG
Step 3:
T: GTTATAGCTGATCGCGGCGTAGCGGCGAA
P: GTAGCGGCG
Step 4:
T: GTTATAGCTGATCGCGGCGTAGCGGCGAA
P: GTAGCGGCG
Skipped 15 alignments


11 characters of T we ignored
Step 1:
T: GTTATAGCTGATCGCGGCGTAGCGGCGAA
P: GTAGCGGCG
Step 2:
T: GTTATAGCTGATCGCGGCGTAGCGGCGAA
P: GTAGCGGCG
Step 3:
T: GTTATAGCTGATCGCGGCGTAGCGGCGAA
P: GTAGCGGCG
Step 4:
T: GTTATAGCTGATCGCGGCGTAGCGGCGAA
P: GTAGCGGCG
Skipped 15 alignments

Pattern Matching Algorithms

- Boyer-Moore Algorithm (contd.)
  - The following steps describe the basic Boyer-Moore algorithm at a high level:

1. Align the target and text on the left.

2. Repeat until the target's last character is aligned beyond the end of the text:

a) Compare the characters in the target with the corresponding characters in the text, starting from the end of the target and moving backwards toward the beginning.
b) If all the characters match, you've found a match.
c) Suppose character X in the text doesn't match the corresponding character in the target. Slide the target to the right until the X aligns with the next character with the same value X in the target to the left of the current position. If no such character X exists to the left of the position in the target, slide the target to the right by its full length.

Time Complexity

- Worst-case performance $\Theta(m)$ preprocessing + $O(mn)$ matching
- Best-case performance $\Theta(m)$ preprocessing + $\Omega(n/m)$ matching

- When the pattern does occur in the text, running time of the original algorithm is $O(nm)$ in the worst case

Pattern Matching Algorithms

- Knuth-Morris-Prat Algorithm
  - When the finite automata for a pattern P is constructed, it is easy to put in the arrows that correspond to a successful match
  - For Example: the first step to draw for the pattern P = AABC is given as shown in Figure below:

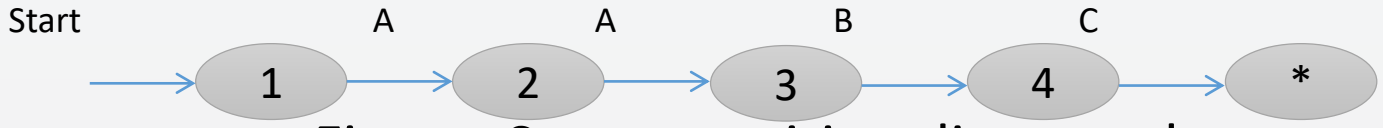


Figure: State transition diagram that accepts AABC

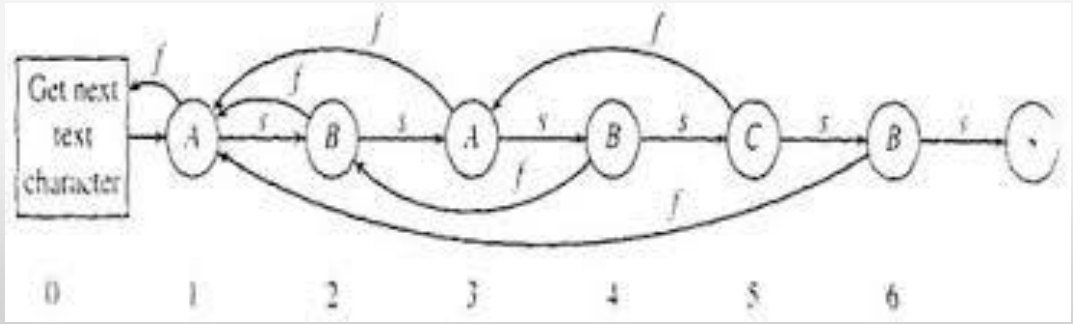
  - The difficult part is to insert other arrows. The Knuth-Morris-Prat (KMP) algorithm constructs the flowchart that can be used to scan the text

Pattern Matching Algorithms

- Knuth-Morris-Prat Algorithm
  - The KMP flowchart consists of states labeled by the characters in P
  - There are two links from each state, i.e., success link and failure link
  - The success link is one to follow if the desired character is read from the text and failure link is one to follow if the desired character is not read from the text
  - The difference between KMP flowchart and the finite automata is as follows:
    - The character labels of the KMP flowchart are on the nodes rather than on the arrows
    - The next character from the text is read only after a success link has been followed
    - If a failure link is followed, then the same text character is reconsidered
    - There is an extra node that causes a new text character to be read. The scan starts at this node

Pattern Matching Algorithms

- Knuth-Morris-Prat Algorithm
  - As in the finite automation, if a node labeled with the * is reached, a copy of the pattern has been found. If the end of the text is reached elsewhere, then the flowchart terminates successfully
  - Figure below shows the KMP flowchart for the patterns P=ABABCB, and the action of that KMP flowchart on the text T = ACABAABABA is shown in the table below:



Pattern Matching Algorithms

- Knuth-Morris-Prat Algorithm

Index	Character	KMP Cell #	Success/Failure	Remarks
1	A	1	S	Go to the next char
2	C	2	F	Reconsider the same text and go to KMP cell 1
2	C	1	F	Reconsider the same text and go to KMP cell 0
2	C	0	Get next char	Read the next text char
3	A	1	S	Read the next text char
4	B	2	S	Read the next text char
5	A	3	S	Read the next text char
6	A	4	F	Read the same text and move to KMP cell 2

Pattern Matching Algorithms

- Knuth-Morris-Prat Algorithm

Index	Character	KMP Cell #	Success/Failure	Remarks
6	A	2	F	Reconsider the same text char and move to KMP cell 1
6	A	1	S	Read the next text character
7	B	2	S	Read the next text character
8	A	3	S	Read the next text character
9	B	4	S	Read the next text character
10	A	5	F	Reconsider the same text char and move to KMP cell 3
10	A	3	S	Read the next text character
11	None	4	Failure	Pattern is not found in the text

Pattern Matching Algorithms

- Algorithm for KMP Scan
  - KMPSCAN(P,T,M,fail)

// P is the pattern of size m, T is the text and fail is the array of failure links //

```
{
    int match, j, k;
    match = -1;
    j = 1, k = 1;
  while (j is not greater than the last index of T)
  {
   if (k > m)
   match = j – m; // match found // }
```

Pattern Matching Algorithms

- Algorithm for KMP Scan

```
Break;
    if (k == 0)
        j++;
        k = 1;
  else if (T[i]  == P[u])
        j++;
        k++;
 else
      k = fail [k];  // follow fail arrow // }
 return match;
```