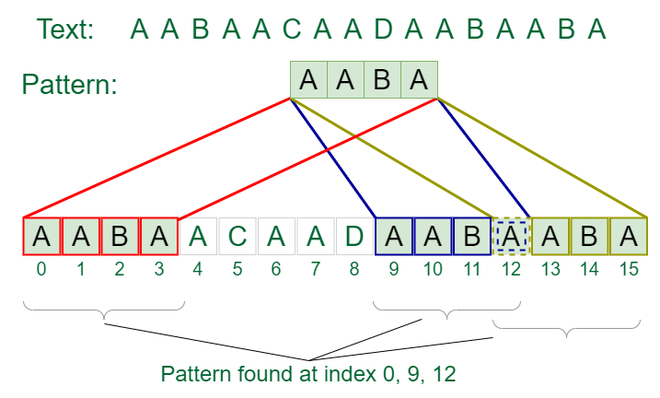
# **KMP Algorithm for Pattern Searching**

Given a text **txt[0 . . . N-1]** and a pattern **pat[0 . . . M-1]**, write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that **N> M**.

**Examples:**

***Input:*** *txt[] = “THIS IS A TEST TEXT”, pat[] = “TEST”****Output:*** *Pattern found at index 10*

***Input:***  *txt[] = “AABAACAADAABAABA”  
 pat[] = “AABA”****Output:*** *Pattern found at index 0, Pattern found at index 9, Pattern found at index 12*



We have discussed the Naive pattern-searching algorithm in the [previous post](https://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/). The worst case complexity of the Naive algorithm is O(m(n-m+1)). The time complexity of the KMP algorithm is O(n+m) in the worst case.

**KMP (Knuth Morris Pratt) Pattern Searching:**

The [Naive pattern-searching algorithm](https://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/) doesn’t work well in cases where we see many matching characters followed by a mismatching character.

**Examples:**

***1)*** *txt[] = “AAAAAAAAAAAAAAAAAB”, pat[] = “AAAAB”****2)*** *txt[] = “ABABABCABABABCABABABC”, pat[] = “ABABAC” (not a worst case, but a bad case for Naive)*

The KMP matching algorithm uses degenerating property (pattern having the same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst-case complexity to **O(n+m)**.

The basic idea behind KMP’s algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match.

***Matching Overview***

*txt = “AAAAABAAABA”   
pat = “AAAA”  
We compare first window of* ***txt*** *with* ***pat***

*txt = “****AAAA****ABAAABA”   
pat = “****AAAA****” [Initial position]  
We find a match. This is same as* [*Naive String Matching*](https://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/)*.*

*In the next step, we compare next window of* ***txt*** *with* ***pat****.*

*txt = “****AAAAA****BAAABA”   
pat = “****AAAA****” [Pattern shifted one position]*

*This is where KMP does optimization over Naive. In this second window, we only compare fourth A of pattern  
with fourth character of current window of text to decide whether current window matches or not. Since we know   
first three characters will anyway match, we skipped matching first three characters.*

***Need of Preprocessing?***

*An important question arises from the above explanation, how to know how many characters to be skipped. To know this,   
we pre-process pattern and prepare an integer array lps[] that tells us the count of characters to be skipped*

### **Preprocessing Overview:**

* KMP algorithm preprocesses pat[] and constructs an auxiliary **lps[]** of size **m** (same as the size of the pattern) which is used to skip characters while matching.
* Name **lps** indicates the longest proper prefix which is also a suffix. A proper prefix is a prefix with a whole string not allowed. For example, prefixes of “ABC” are “”, “A”, “AB” and “ABC”. Proper prefixes are “”, “A” and “AB”. Suffixes of the string are “”, “C”, “BC”, and “ABC”.
* We search for lps in subpatterns. More clearly we focus on sub-strings of patterns that are both prefix and suffix.
* For each sub-pattern pat[0..i] where i = 0 to m-1, lps[i] stores the length of the maximum matching proper prefix which is also a suffix of the sub-pattern pat[0..i].

*lps[i] = the longest proper prefix of pat[0..i] which is also a suffix of pat[0..i].*

**Note:** lps[i] could also be defined as the longest prefix which is also a proper suffix. We need to use it properly in one place to make sure that the whole substring is not considered.

Examples of lps[] construction:

*For the pattern “AAAA”, lps[] is [0, 1, 2, 3]*

*For the pattern “ABCDE”, lps[] is [0, 0, 0, 0, 0]*

*For the pattern “AABAACAABAA”, lps[] is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]*

*For the pattern “AAACAAAAAC”, lps[] is [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]*

*For the pattern “AAABAAA”, lps[] is [0, 1, 2, 0, 1, 2, 3]*

### **Preprocessing Algorithm:**

In the preprocessing part,

* We calculate values in **lps[]**. To do that, we keep track of the length of the longest prefix suffix value (we use **len** variable for this purpose) for the previous index
* We initialize **lps[0]** and **len** as 0.
* If **pat[len]** and **pat[i]** match, we increment **len** by 1 and assign the incremented value to lps[i].
* If pat[i] and pat[len] do not match and len is not 0, we update len to lps[len-1]
* See **computeLPSArray()** in the below code for details

**Illustration of preprocessing (or construction of lps[]):**

*pat[] = “AAACAAAA”*

***=> len = 0, i = 0:***

* *lps[0] is always 0, we move to i = 1*

***=> len = 0, i = 1:***

* *Since pat[len] and pat[i] match, do len++,*
* *store it in lps[i] and do i++.*
* *Set len = 1, lps[1] = 1, i = 2*

***=> len = 1, i = 2:***

* *Since pat[len] and pat[i] match, do len++,*
* *store it in lps[i] and do i++.*
* *Set len = 2, lps[2] = 2, i = 3*

***=> len = 2, i = 3:***

* *Since pat[len] and pat[i] do not match, and len > 0,*
* *Set len = lps[len-1] = lps[1] = 1*

***=> len = 1, i = 3:***

* *Since pat[len] and pat[i] do not match and len > 0,*
* *len = lps[len-1] = lps[0] = 0*

***=> len = 0, i = 3:***

* *Since pat[len] and pat[i] do not match and len = 0,*
* *Set lps[3] = 0 and i = 4*

***=> len = 0, i = 4:***

* *Since pat[len] and pat[i] match, do len++,*
* *Store it in lps[i] and do i++.*
* *Set len = 1, lps[4] = 1, i = 5*

***=> len = 1, i = 5:***

* *Since pat[len] and pat[i] match, do len++,*
* *Store it in lps[i] and do i++.*
* *Set len = 2, lps[5] = 2, i = 6*

***=> len = 2, i = 6:***

* *Since pat[len] and pat[i] match, do len++,*
* *Store it in lps[i] and do i++.*
* *len = 3, lps[6] = 3, i = 7*

***=> len = 3, i = 7:***

* *Since pat[len] and pat[i] do not match and len > 0,*
* *Set len = lps[len-1] = lps[2] = 2*

***=> len = 2, i = 7:***

* *Since pat[len] and pat[i] match, do len++,*
* *Store it in lps[i] and do i++.*
* *len = 3, lps[7] = 3, i = 8*

*We stop here as we have constructed the whole lps[].*

### **Implementation of KMP algorithm:**

Unlike the [Naive algorithm](https://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/), where we slide the pattern by one and compare all characters at each shift, we use a value from lps[] to decide the next characters to be matched. The idea is to not match a character that we know will anyway match.

How to use lps[] to decide the next positions (or to know the number of characters to be skipped)?

* We start the comparison of pat[j] with j = 0 with characters of the current window of text.
* We keep matching characters txt[i] and pat[j] and keep incrementing i and j while pat[j] and txt[i] keep **matching**.
* When we see a **mismatch**
  + We know that characters pat[0..j-1] match with txt[i-j…i-1] (Note that j starts with 0 and increments it only when there is a match).
  + We also know (from the above definition) that lps[j-1] is the count of characters of pat[0…j-1] that are both proper prefix and suffix.
  + From the above two points, we can conclude that we do not need to match these lps[j-1] characters with txt[i-j…i-1] because we know that these characters will anyway match. Let us consider the above example to understand this.

Below is the illustration of the above algorithm:

*Consider txt[] = “****AAAAABAAABA****“, pat[] = “****AAAA****“*

*If we follow the above LPS building process then* ***lps[] = {0, 1, 2, 3}***

***-> i = 0, j = 0:*** *txt[i] and pat[j] match, do i++, j++*

***-> i = 1, j = 1:*** *txt[i] and pat[j] match, do i++, j++*

***-> i = 2, j = 2:*** *txt[i] and pat[j] match, do i++, j++*

***-> i = 3, j = 3:*** *txt[i] and pat[j] match, do i++, j++*

***-> i = 4, j = 4:*** *Since j = M, print pattern found and reset j,* ***j*** *= lps[j-1] = lps[3] =* ***3***

*Here unlike Naive algorithm, we do not match first three   
characters of this window. Value of lps[j-1] (in above step) gave us index of next character to match.*

***-> i = 4, j = 3:*** *txt[i] and pat[j] match, do i++, j++*

***-> i = 5, j = 4:*** *Since j == M, print pattern found and reset j,* ***j*** *= lps[j-1] = lps[3] =* ***3****Again unlike Naive algorithm, we do not match first three characters of this window. Value of lps[j-1] (in above step) gave us index of next character to match.*

***-> i = 5, j = 3:*** *txt[i] and pat[j] do NOT match and j > 0, change only j.* ***j*** *= lps[j-1] = lps[2] =* ***2***

***-> i = 5, j = 2:*** *txt[i] and pat[j] do NOT match and j > 0, change only j.* ***j*** *= lps[j-1] = lps[1] =* ***1***

***-> i = 5, j = 1:*** *txt[i] and pat[j] do NOT match and j > 0, change only j.* ***j*** *= lps[j-1] = lps[0] =* ***0***

***-> i = 5, j = 0:*** *txt[i] and pat[j] do NOT match and j is 0, we do i++.*

***-> i = 6, j = 0:*** *txt[i] and pat[j] match, do i++ and j++*

***-> i = 7, j = 1:*** *txt[i] and pat[j] match, do i++ and j++*

*We continue this way till there are sufficient characters in the text to be compared with the characters in the pattern…*

**Time Complexity:** O(N+M) where N is the length of the text and M is the length of the pattern to be found.

**Auxiliary Space:** O(M)

# 

# 

# 

# **Boyer Moore Algorithm for Pattern Searching**

Pattern searching is an important problem in computer science. When we do search for a string in a notepad/word file, browser, or database, pattern searching algorithms are used to show the search results.

A typical problem statement would be-

” Given a text txt[0..n-1] and a pattern pat[0..m-1] where n is the length of the text and m is the length of the pattern, write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m. “  
**Examples:**

***Input:*** *txt[] = “THIS IS A TEST TEXT”*

*pat[] = “TEST”*

***Output:*** *Pattern found at index 10*

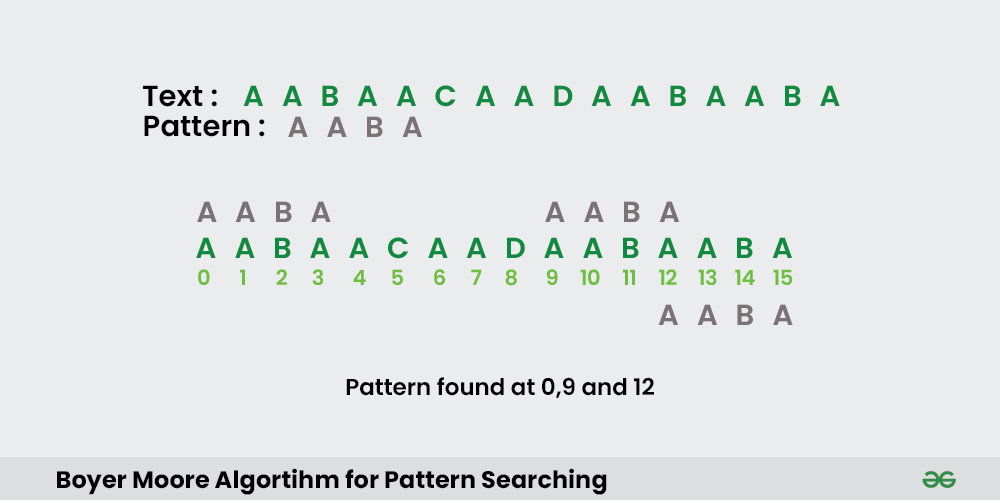
***Input:*** *txt[] = “AABAACAADAABAABA”*

*pat[] = “AABA”*

***Output:*** *Pattern found at index 0*

*Pattern found at index 9*

*Pattern found at index 12*

**

In this post, we will discuss the Boyer Moore pattern searching algorithm. Like [KMP](https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/) and [Finite Automata](https://www.geeksforgeeks.org/finite-automata-algorithm-for-pattern-searching/) algorithms, Boyer Moore algorithm also preprocesses the pattern.   
Boyer Moore is a combination of the following two approaches.

1. **Bad Character Heuristic**
2. **Good Suffix Heuristic**

Both of the above heuristics can also be used independently to search a pattern in a text. Let us first understand how two independent approaches work together in the Boyer Moore algorithm.

If we take a look at the [Naive algorithm](https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/), it slides the pattern over the text one by one. [KMP algorithm](https://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/) does preprocessing over the pattern so that the pattern can be shifted by more than one. The Boyer Moore algorithm does preprocessing for the same reason. It processes the pattern and creates different arrays for each of the two heuristics. At every step, it slides the pattern by the max of the slides suggested by each of the two heuristics. **So, it uses greatest offset suggested by the two heuristics at every step**.

Unlike the previous pattern searching algorithms, the **Boyer Moore algorithm starts matching from the last character of the pattern.**In this post, we will discuss the bad character heuristic and the Good Suffix heuristic in the next post.

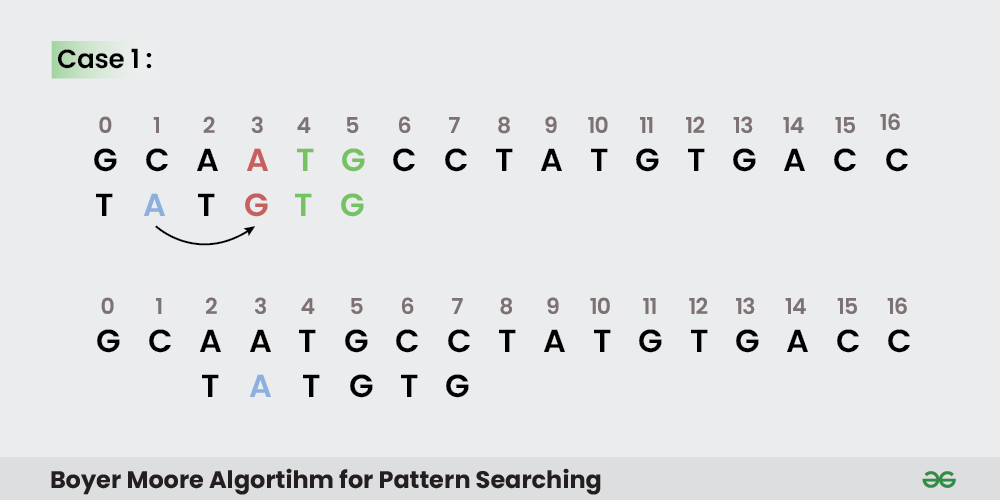
### **Bad Character Heuristic**

The idea of bad character heuristic is simple. The character of the text which doesn’t match with the current character of the pattern is called the **Bad Character**. Upon mismatch, we shift the pattern until –

1. The mismatch becomes a match.
2. Pattern P moves past the mismatched character.

### **Case 1 – Mismatch become match**

We will lookup the position of the last occurrence of the mismatched character in the pattern, and if the mismatched character exists in the pattern, then we’ll shift the pattern such that it becomes aligned to the mismatched character in the text T.



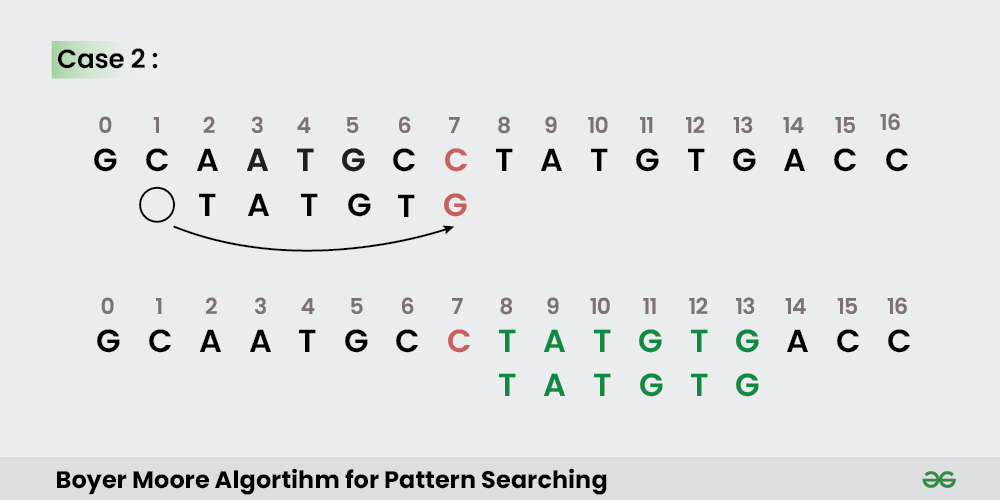
#### **Explanation:**

In the above example, we got a mismatch at position **3**.

Here our mismatching character is “A”. Now we will search for last occurrence of “A” in pattern. We got “A” at position 1 in pattern (displayed in Blue) and this is the last occurrence of it. Now we will shift pattern 2 times so that “A” in pattern get aligned with “A” in text.

### **Case 2 – Pattern move past the mismatch character**

We’ll lookup the position of last occurrence of mismatching character in pattern and if character does not exist we will shift pattern past the mismatching character.



#### **Explanation:**

Here we have a mismatch at position **7**.

The mismatching character “C” does not exist in pattern before position 7 so we’ll shift pattern past to the position 7 and eventually in above example we have got a perfect match of pattern (displayed in Green). We are doing this because “C” does not exist in the pattern so at every shift before position 7 we will get mismatch and our search will be fruitless.

### **Implementation:**

In the following implementation, we pre-process the pattern and store the last occurrence of every possible character in an array of size equal to alphabet size. If the character is not present at all, then it may result in a shift by m (length of pattern). Therefore, the bad character heuristic takes O(n/m) time in the best case.

**Time Complexity :** O(m\*n)

**Auxiliary Space:** O(1)

The Bad Character Heuristic may take **O(m\*n)** time in worst case. The worst case occurs when all characters of the text and pattern are same. For example, txt[] = “AAAAAAAAAAAAAAAAAA” and pat[] = “AAAAA”. The Bad Character Heuristic may take O(n/m) in the best case. The best case occurs when all the characters of the text and pattern are different.

**Good Suffix Heuristic**

It is for pattern searching. Just like bad character heuristic, a preprocessing table is generated for good suffix heuristic.

Let

**t**

be substring of text

**T**

which is matched with substring of pattern

**P**

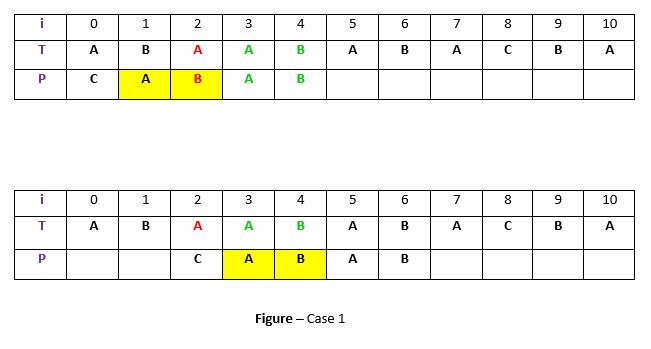
. Now we shift pattern until : 1) Another occurrence of t in P matched with t in T. 2) A prefix of P, which matches with suffix of t 3) P moves past t

**Case 1: Another occurrence of t in P matched with t in T**

Pattern P might contain few more occurrences of

**t**

. In such case, we will try to shift the pattern to align that occurrence with t in text T. For example-



**Explanation:**

In the above example, we have got a substring t of text T matched with pattern P (in green) before mismatch at index 2. Now we will search for occurrence of t (“AB”) in P. We have found an occurrence starting at position 1 (in yellow background) so we will right shift the pattern 2 times to align t in P with t in T. This is weak rule of original Boyer Moore and not much effective, we will discuss a

**Strong Good Suffix rule**

shortly.

**Case 2: A prefix of P, which matches with suffix of t in T**

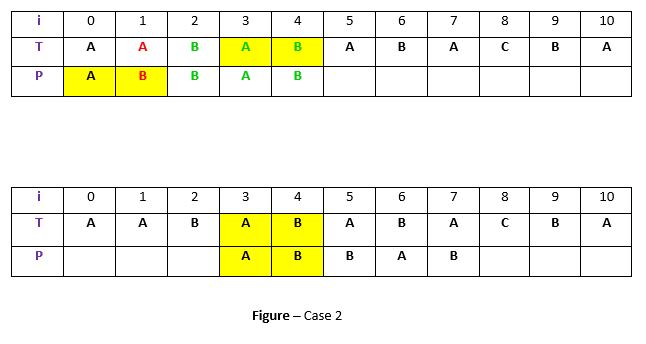
It is not always likely that we will find the occurrence of t in P. Sometimes there is no occurrence at all, in such cases sometimes we can search for some

**suffix of t**

matching with some

**prefix of P**

and try to align them by shifting P. For example –

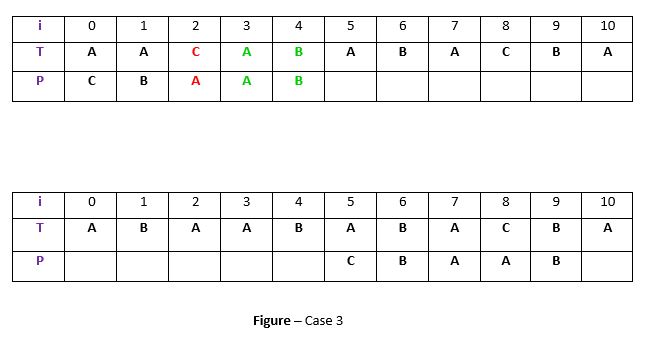


**Explanation:**

In above example, we have got t (“BAB”) matched with P (in green) at index 2-4 before mismatch . But because there exists no occurrence of t in P we will search for some prefix of P which matches with some suffix of t. We have found prefix “AB” (in the yellow background) starting at index 0 which matches not with whole t but the suffix of t “AB” starting at index 3. So now we will shift pattern 3 times to align prefix with the suffix.

**Case 3: P moves past t**

If the above two cases are not satisfied, we will shift the pattern past the t. For example –



**Explanation:**

If above example, there exist no occurrence of t (“AB”) in P and also there is no prefix in P which matches with the suffix of t. So, in that case, we can never find any perfect match before index 4, so we will shift the P past the t ie. to index 5.

**Strong Good suffix Heuristic**

Suppose substring

**q = P[i to n]**

got matched with

**t**

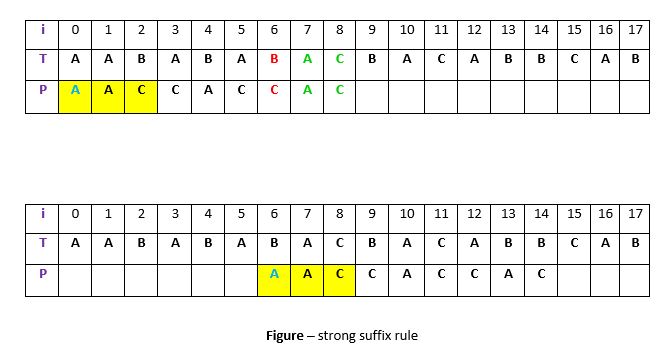
in T and

**c = P[i-1]**

is the mismatching character. Now unlike case 1 we will search for t in P which is not preceded by character

**c**

. The closest such occurrence is then aligned with t in T by shifting pattern P. For example –



**Explanation:**

In above example,

**q = P[7 to 8]**

got matched with t in T. The mismatching character

**c**

is “C” at position P[6]. Now if we start searching t in P we will get the first occurrence of t starting at position 4. But this occurrence is preceded by “C” which is equal to c, so we will skip this and carry on searching. At position 1 we got another occurrence of t (in the yellow background). This occurrence is preceded by “A” (in blue) which is not equivalent to c. So we will shift pattern P 6 times to align this occurrence with t in T.We are doing this because we already know that character

**c = “C”**

causes the mismatch. So any occurrence of t preceded by c will again cause mismatch when aligned with t, so that’s why it is better to skip this.

**Preprocessing for Good suffix heuristic**

As a part of preprocessing, an array

**shift**

is created. Each entry

**shift[i]**

contain the distance pattern will shift if mismatch occur at position

**i-1**

. That is, the suffix of pattern starting at position

**i**

is matched and a mismatch occur at position

**i-1**

. Preprocessing is done separately for strong good suffix and case 2 discussed above.

**1) Preprocessing for Strong Good Suffix**

Before discussing preprocessing, let us first discuss the idea of border. A

**border**

is a substring which is both proper suffix and proper prefix. For example, in string

**“ccacc”**

,

**“c”**

is a border,

**“cc”**

is a border because it appears in both end of string but

**“cca”**

is not a border. As a part of preprocessing an array

**bpos**

(border position) is calculated. Each entry

**bpos[i]**

contains the starting index of border for suffix starting at index i in given pattern P. The suffix

**?**

beginning at position m has no border, so

**bpos[m]**

is set to

**m+1**

where

**m**