UNIT V - data structure notes r18 jntuh

Computer Science and Engineering (Jawaharlal Nehru Technological University, Hyderabad)

UNIT - V

Pattern Matching and Tries: Pattern matching algorithms-Brute force, the Boyer –Moore algorithm, the Knuth-Morris-Pratt algorithm, Standard Tries, Compressed Tries, Suffix tries.

Pattern Matching

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or 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], 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

Different Types of Pattern Matching Algorithms

1. Navie Based Algorithm or Brute Force Algorithm
2. Boyer Moore Algorithm
3. Knuth-Morris Pratt (KMP) Algorithm

Navie Based Algorithm or Brute Force Algorithm

When we talk about a string matching algorithm, every one can get a simple string matching technique. That is starting from first letters of the text and first letter of the pattern check whether these two letters are equal. if it is, then check second letters of the text and pattern. If it is not equal, then move first letter of the pattern to the second letter of the text. then check these two letters. this is the simple technique everyone can thought.

Brute Force string matching algorithm is also like that. Therefore we call that as Naive string matching algorithm. Naive means basic.

Brute Force Algorithm

```
do
    if (text letter == pattern letter)
        compare next letter of pattern to next letter of text
    else
        move pattern down text by one letter
while (entire pattern found or end of text)
```

Lets learn this method using an example.

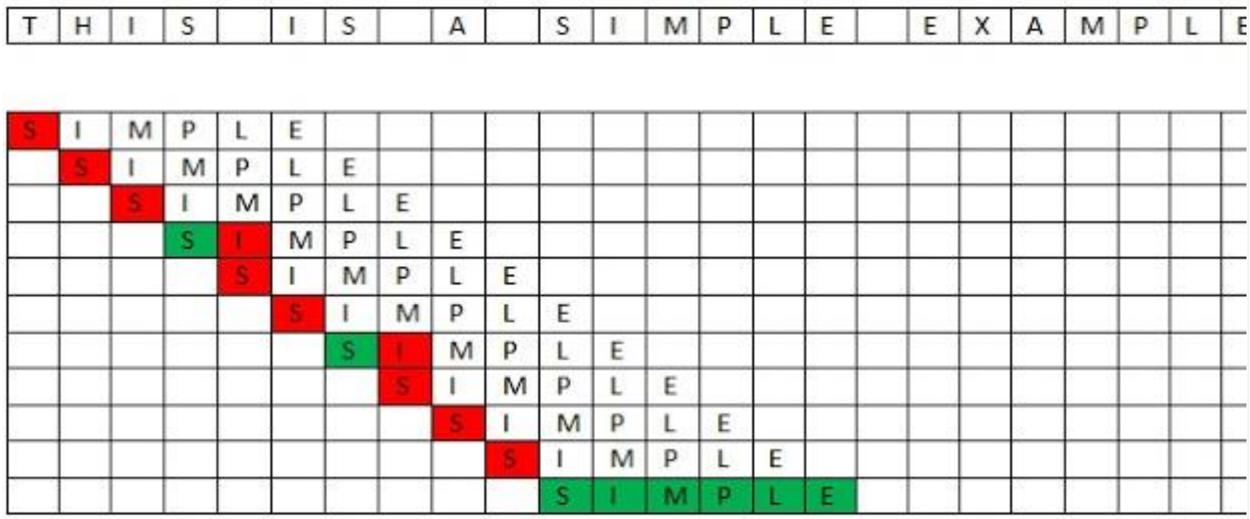
EXAMPLE 1

Let our text (T) as,

THIS IS A SIMPLE EXAMPLE

and our pattern (P) as,

SIMPLE



T	H	I	S		I	S		A		S	I	M	P	L	E		E	X	A	M	P	L	E
S	I	M	P	L	E																		
	S	I	M	P	L	E																	
		S	I	M	P	L	E																
			S	I	M	P	L	E															
				S	I	M	P	L	E														
					S	I	M	P	L	E													
						S	I	M	P	L	E												
							S	I	M	P	L	E											
								S	I	M	P	L	E										
									S	I	M	P	L	E									
										S	I	M	P	L	E								

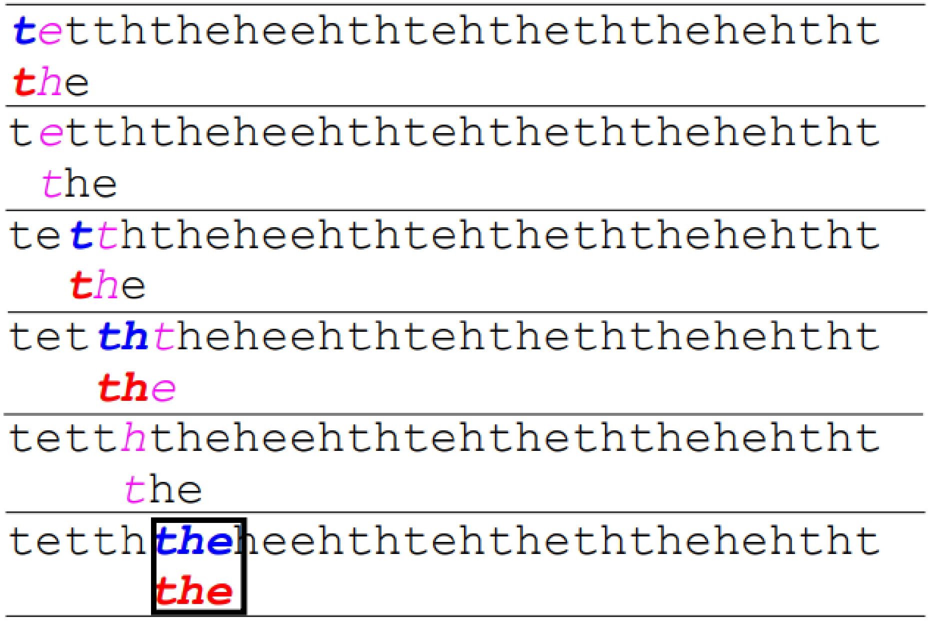
Red Boxes-Mismatch Green Boxes-Match

In above red boxes says mismatch letters against letters of the text and green boxes says match letters against letters of the text. According to the above

In first raw we check whether first letter of the pattern is matched with the first letter of the text. It is mismatched, because "S" is the first letter of pattern and "T" is the first letter of text. Then we move the pattern by one position. Shown in second raw.

Then check first letter of the pattern with the second letter of text. It is also mismatched. Likewise we continue the checking and moving process. In fourth raw we can see first letter of the pattern matched with text. Then we do not do any moving but we increase testing letter of the pattern. We only move the position of pattern by one when we find mismatches. Also in last raw, we can see all the letters of the pattern matched with the some letters of the text continuously.

Example 2



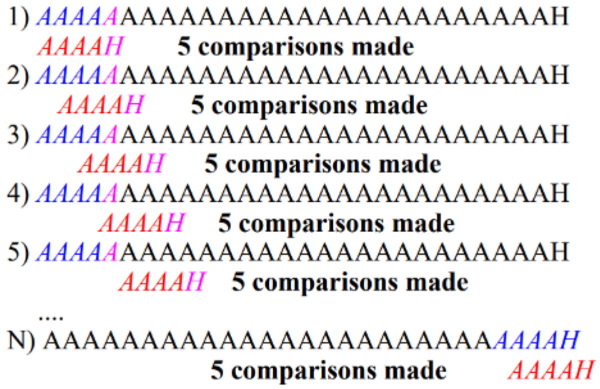
```
tetththeheehthtehtheththehehtht
the
tetththeheehthtehtheththehehtht
 the
tetththeheehthtehtheththehehtht
  the
tetththeheehthtehtheththehehtht
   the
tetththeheehthtehtheththehehtht
    the
tetththeheehthtehtheththehehtht
     the
```

Running Time Analysis Of Brute Force String Matching Algorithm

Worst Case

Given a pattern M characters in length, and a text N characters in length...
• Worst case: compares pattern to each substring of text of length M.
For example, M=5.

- Total number of comparisons: M (N-M+1) • Worst case time complexity: O(MN)



```
1) AAAAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAH       5 comparisons made
2) AAAAAAAAAAAAAAAAAAAAAAAAAAH
    AAAAH      5 comparisons made
3) AAAAAAAAAAAAAAAAAAAAAAAAAAH
     AAAAH     5 comparisons made
4) AAAAAAAAAAAAAAAAAAAAAAAAAAH
      AAAAH    5 comparisons made
5) AAAAAAAAAAAAAAAAAAAAAAAAAAH
       AAAAH   5 comparisons made
  ....
N) AAAAAAAAAAAAAAAAAAAAAAAAAAH
           5 comparisons made     AAAAH
```

- Total number of comparisons: M (N-M+1)
- Worst case time complexity: O(MN)

Best case

Given a pattern M characters in length, and a text N characters in length...

- **Best case if pattern found**: Finds pattern in first M positions of text.

For example, M=5.

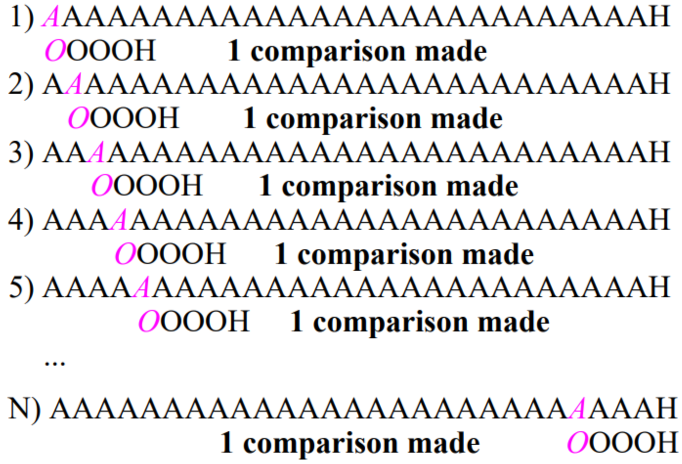
AAAAAAAAAAAAAAAAAAAAAAAAAAAH

AAAAA 5 comparisons made

- Total number of comparisons: M
- Best case time complexity: O(M)

Best case if pattern not found:

Always mismatch on first character. For example, M=5.



```
1) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   OOOOH       1 comparison made
2) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
    OOOOH      1 comparison made
3) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
     OOOOH     1 comparison made
4) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
      OOOOH    1 comparison made
5) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
       OOOOH   1 comparison made
  ...
N) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
           1 comparison made      OOOOH
```

- Total number of comparisons: N
- Best case time complexity: O(N)

Advantages

1. Very simple technique and also that does not require any preprocessing. Therefore total running time is the same as its matching time.

Disadvantages

1. Very inefficient method. Because this method takes only one position movement in each time

Boyer Moore Algorithm for Pattern Searching

The B-M algorithm takes a backward approach . the pattern string(p) is aligned with the start of the text string(T) and then compare the characters of pattern from right to left beginning with rightmost character

If a character is compared that is not within the pattern, no match can be found by comparing any furher characters at this position so the pattern can be shifted completely past the mismatching character.

For determining the possible shifts , B-M algorithm uses 2 preprocessing strategies simultaneously whenever a mismatch occurs, the algorithm computes a shift using both strategies and selects the longer one. thus it makes use of the most efficient stategy for each individual case

NOTE : Boyer Moore algorithm starts matching from the last character of the pattern.

The 2 strategies are called heuristics of B-M as they are used to reduce the search. They are

1) Bad Character Heuristic
2) Good Suffix Heuristic

Bad Character Heuristic

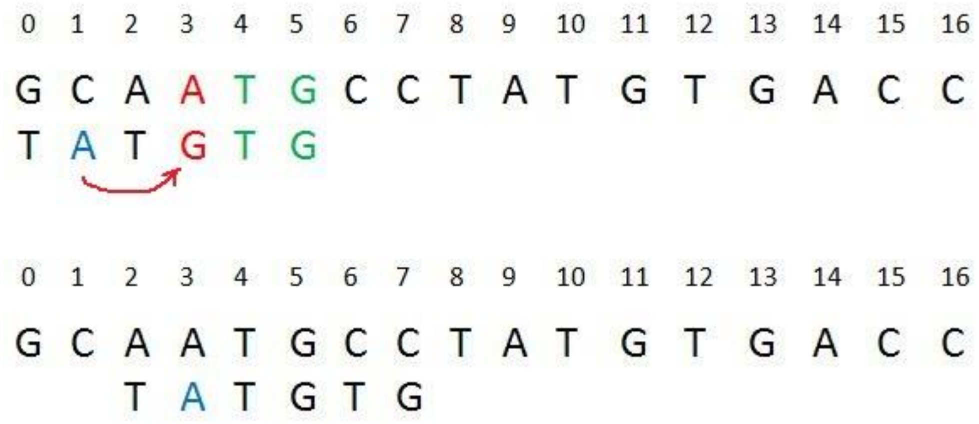
The idea of bad character heuristic is simple. The character of the text which doesn’t match with the current character of the pattern is called the **Bad Character**. Upon mismatch, we shift the pattern until –

1) The mismatch becomes a match

2) Pattern P move past the mismatched character.

Case 1 – Mismatch become match

We will lookup the position of last occurrence of mismatching character in pattern and if mismatching character exist in pattern then we'll shift the pattern such that it get aligned to the mismatching character in text T.

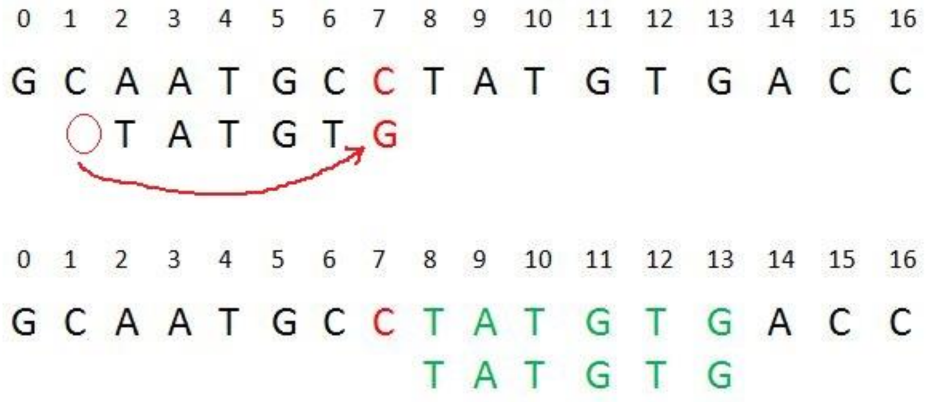


case 1

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.



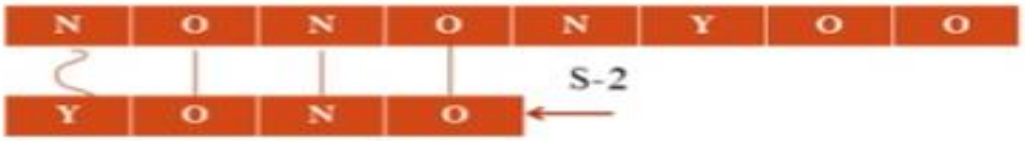
case2

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" do not exist in pattern so at every shift before position 7 we will get mismatch and our search will be fruitless.

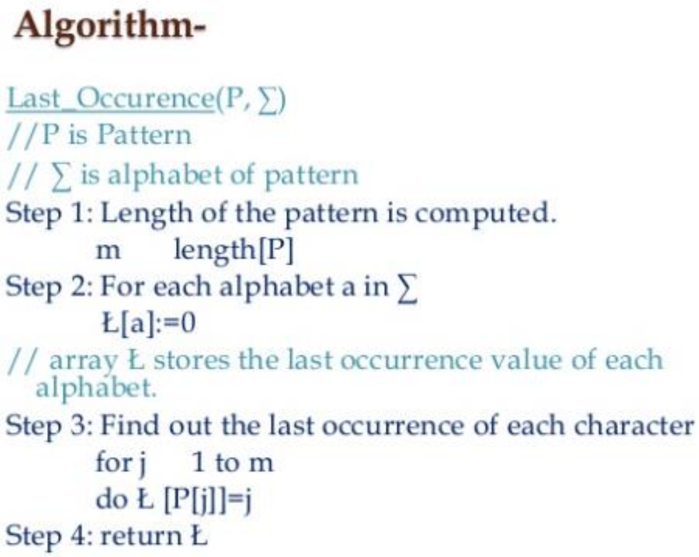
Problem in Bad Character Heuristic

In some cases Bad Character Heuristic produces negative results

For Example:



This means we need some extra information to produce a shift an encountering a bad character. The information is about last position of evry character in the pattern and also the set of every character in the pattern and also the set of characters used in the pattern



Algorithm-

```
Last_Occurence(P, ∑)
//P is Pattern
// ∑ is alphabet of pattern
Step 1: Length of the pattern is computed.
          m      length[P]
Step 2: For each alphabet a in ∑
          Ł[a]:=0
// array Ł stores the last occurrence value of each
   alphabet.
Step 3: Find out the last occurrence of each character
          for j     1 to m
          do Ł [P[j]]=j
Step 4: return Ł
```

2.Good Suffix Heuristic

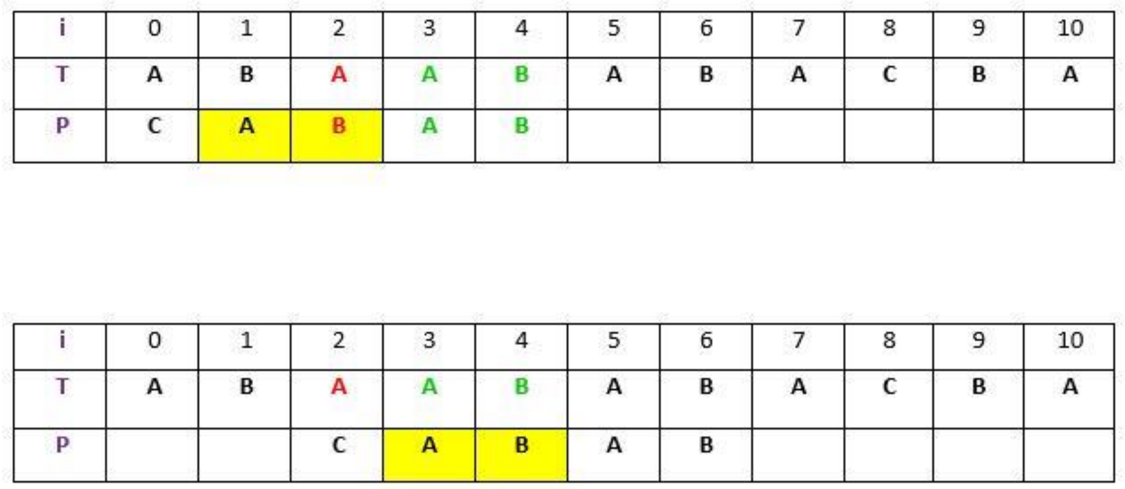
Let **t** be substring of text **T** which is matched with substring of pattern **P**. Now we shift pattern until :

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-



i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P	C	A	B	A	B						

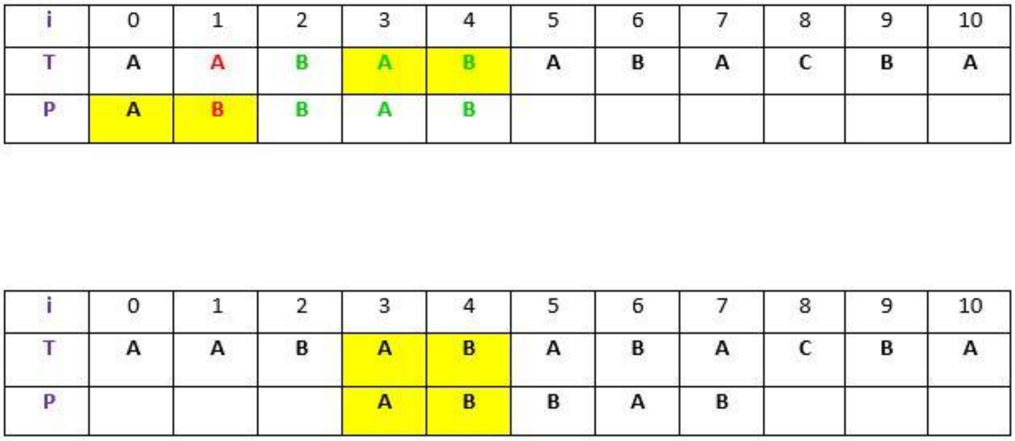
i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P			C	A	B	A	B				

Figure – Case 1

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

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 –



i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	B	A	B	A	B	A	C	B	A
P	A	B	B	A	B						

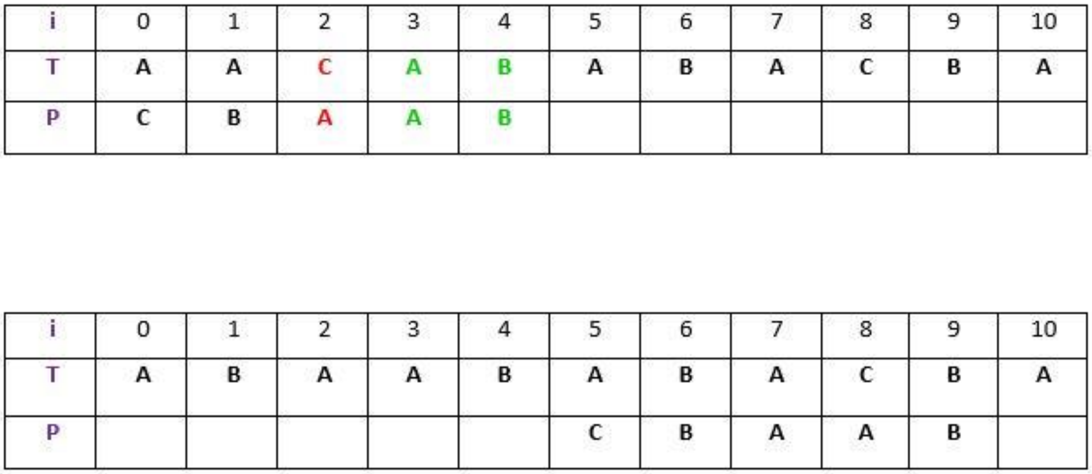
i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	B	A	B	A	B	A	C	B	A
P				A	B	B	A	B			

Figure – Case 2

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 –



i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	C	A	B	A	B	A	C	B	A
P	C	B	A	A	B						

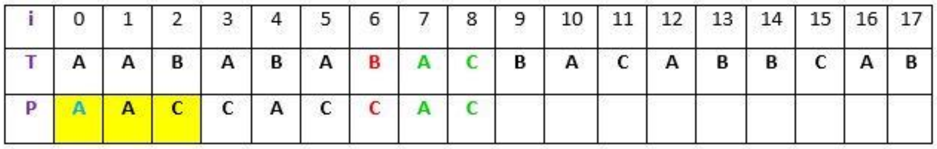
i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P						C	B	A	A	B	

Figure – Case 3

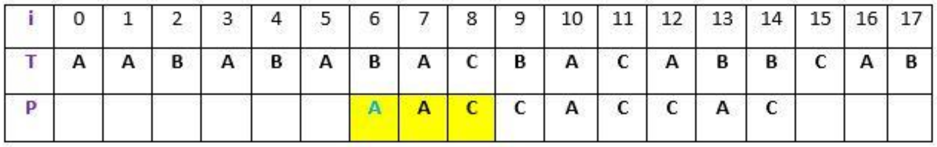
Explanation: If above example, there exist no occurrence of t ("AB") in P and also there is no prefix in P which matches with the suffix of t. So, in that case, we can never find any perfect match before index 4, so we will shift the P past the t ie. to index 5.

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 –



i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T	A	A	B	A	B	A	B	A	C	B	A	C	A	B	B	C	A	B
P	A	A	C	C	A	C	C	A	C									



i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T	A	A	B	A	B	A	B	A	C	B	A	C	A	B	B	C	A	B
P							A	A	C	C	A	C	C	A	C			

Figure – strong suffix rule

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 **ϕ** beginning at position m has no border, so **bpos[m]** is set to **m+1** where **m** is the length of the pattern.
The shift position is obtained by the borders which cannot be extended to the left.

Complexity of Boyer Moore Algorithm

This algorithm takes o(mn) in the worst case and O(nlog(m)/m) on average case, which is the sub linear in the sense that not all characters are inspected

Applications

This algorithm is highly useful in tasks like recursively searching files for virus patterns,searching databases for keys or data ,text and word processing and any other task that requires handling large amount of data at very high speed

Knuth-Morris Pratt (KMP) Algorithm for Pattern Searching

The Naive pattern searching algorithm doesn't work well in cases where we see many matching characters followed by a mismatching character. Following are some examples.

txt[] = "AAAAAAAAAAAAAAAAAB"

pat[] = "AAAAB"

txt[] = "ABABABCABABABCABABABC"

pat[] = "ABABAC" (not a worst case, but a bad case for Naive

KMP Algorithm is one of the most popular patterns matching algorithms. KMP stands for Knuth Morris Pratt. KMP algorithm was invented by Donald Knuth and Vaughan Pratt together and independently by James H Morris in the year 1970. In the year 1977, all the three jointly published KMP Algorithm.

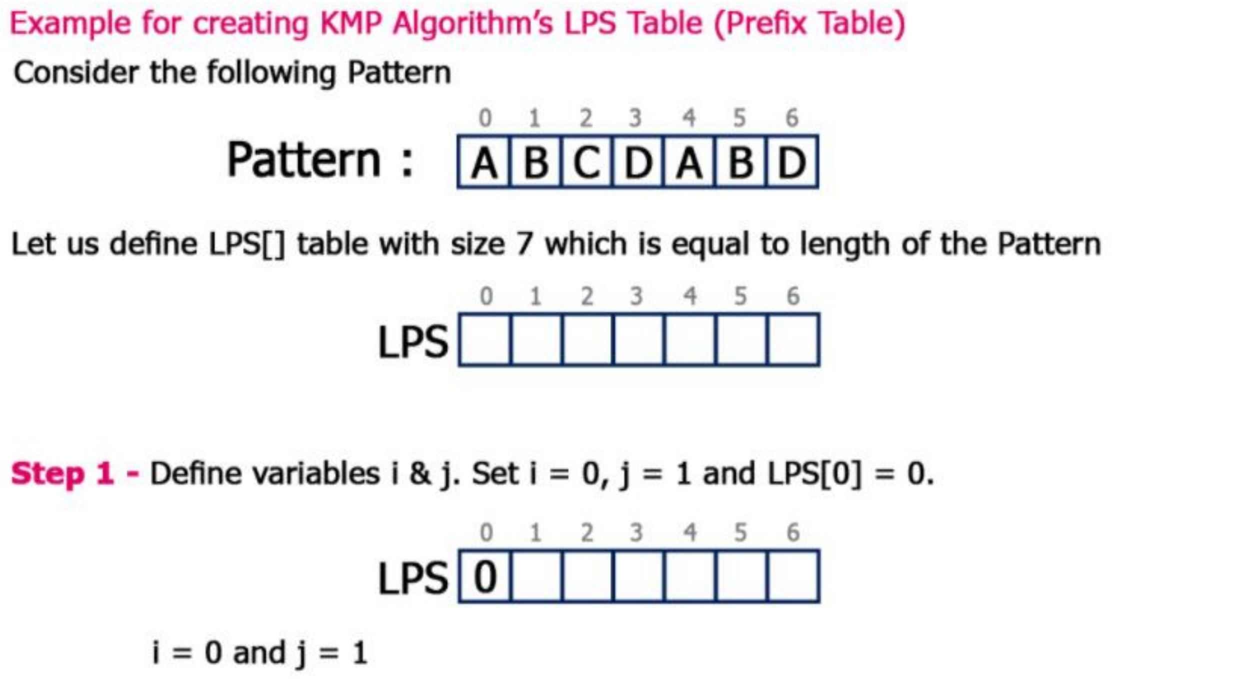
KMP algorithm was the first linear time complexity algorithm for string matching.
KMP algorithm is one of the string matching algorithms used to find a Pattern in a Text.

KMP algorithm is used to find a "Pattern" in a "Text". This algorithm campares character by character from left to right. But whenever a mismatch occurs, it uses a preprocessed table called "**Prefix Table**" to skip characters comparison while matching. Some times prefix table is also known as **LPS Table**. Here LPS stands for "**Longest proper Prefix which is also Suffix".**

Steps for Creating LPS Table (Prefix Table)

- **Step 1** - Define a one dimensional array with the size equal to the length of the Pattern. (LPS[size])
- **Step 2** - Define variables i & j. Set i = 0, j = 1 and LPS[0] = 0.
- **Step 3 -** Compare the characters at Pattern[i] and Pattern[j].
- **Step 4** - If both are matched then set LPS[j] = i+1 and increment both i & j values by one. Goto to Step 3.
- **Step 5** - If both are not matched then check the value of variable 'i'. If it is '0' then set LPS[j] = 0 and increment 'j' value by one, if it is not '0' then set i = LPS[i-1]. Goto Step 3.
- **Step 6**- Repeat above steps until all the values of LPS[] are filled.

Let us use above steps to create prefix table for a pattern...



Example for creating KMP Algorithm's LPS Table (Prefix Table)

Consider the following Pattern

	0	1	2	3	4	5	6
Pattern :	A	B	C	D	A	B	D

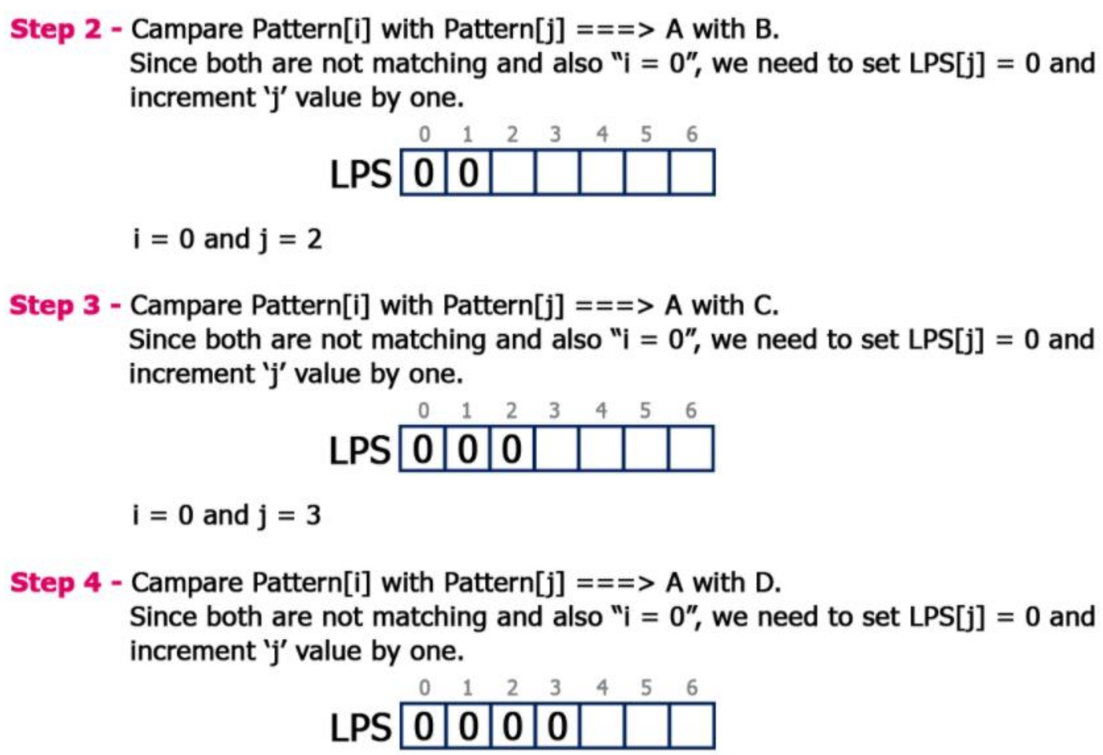
Let us define LPS[] table with size 7 which is equal to length of the Pattern

	0	1	2	3	4	5	6
LPS							

Step 1 - Define variables i & j. Set i = 0, j = 1 and LPS[0] = 0.

	0	1	2	3	4	5	6
LPS	0						

i = 0 and j = 1



Step 2 - Campare Pattern[i] with Pattern[j] ===> A with B.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0					

i = 0 and j = 2

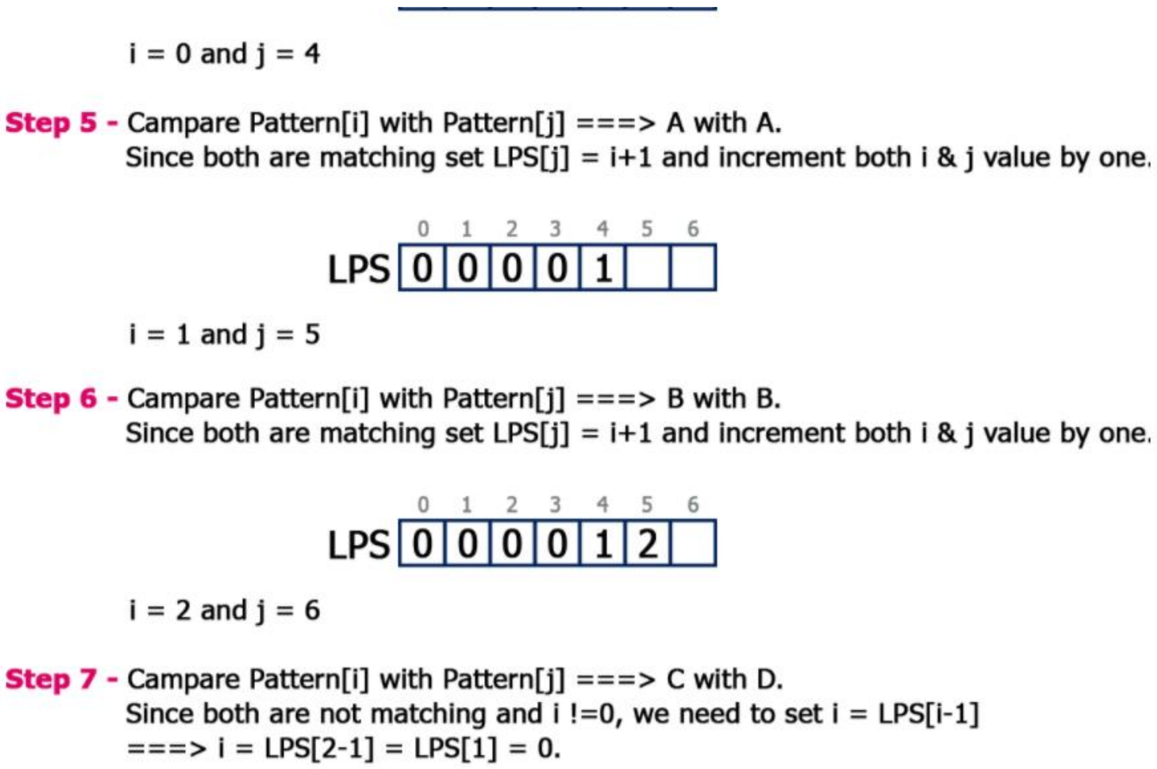
Step 3 - Campare Pattern[i] with Pattern[j] ===> A with C.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0	0				

i = 0 and j = 3

Step 4 - Campare Pattern[i] with Pattern[j] ===> A with D.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0			



i = 0 and j = 4

Step 5 - Campare Pattern[i] with Pattern[j] ===> A with A.
Since both are matching set LPS[j] = i+1 and increment both i & j value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1		

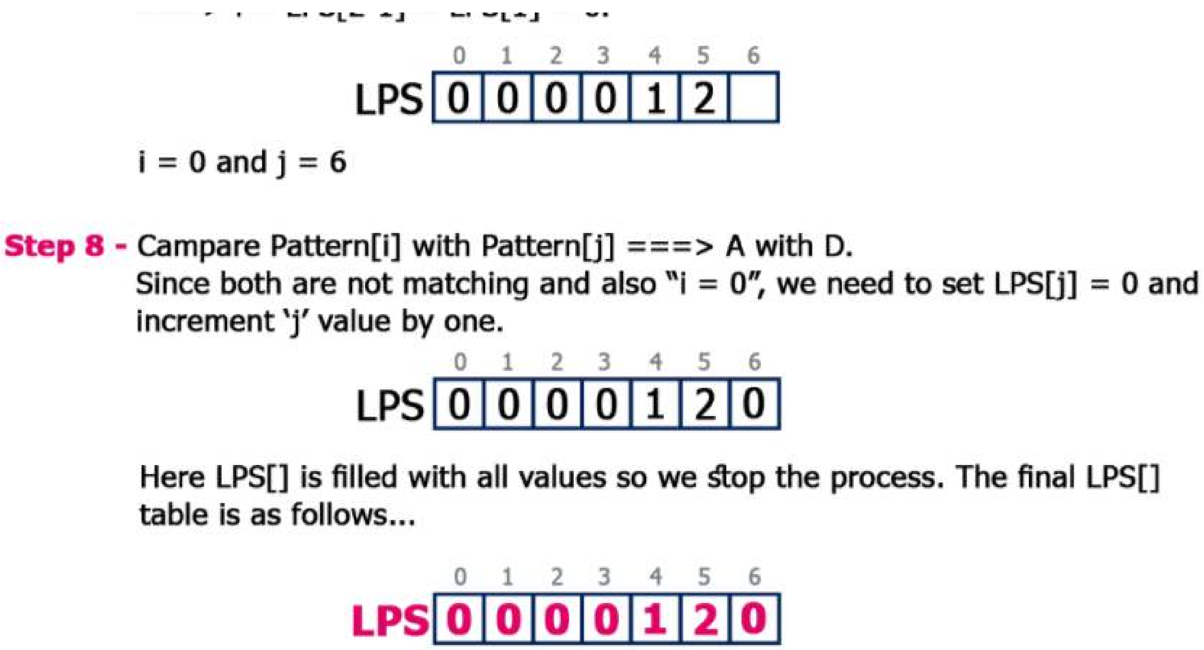
i = 1 and j = 5

Step 6 - Campare Pattern[i] with Pattern[j] ===> B with B.
Since both are matching set LPS[j] = i+1 and increment both i & j value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	

i = 2 and j = 6

Step 7 - Campare Pattern[i] with Pattern[j] ===> C with D.
Since both are not matching and i !=0, we need to set i = LPS[i-1]
===> i = LPS[2-1] = LPS[1] = 0.



	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	

i = 0 and j = 6

Step 8 - Campare Pattern[i] with Pattern[j] ===> A with D.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0

Here LPS[] is filled with all values so we stop the process. The final LPS[] table is as follows...

	0	1	2	3	4	5	6
LPS	**0**	**0**	**0**	**0**	**1**	**2**	**0**

How to use LPS Table

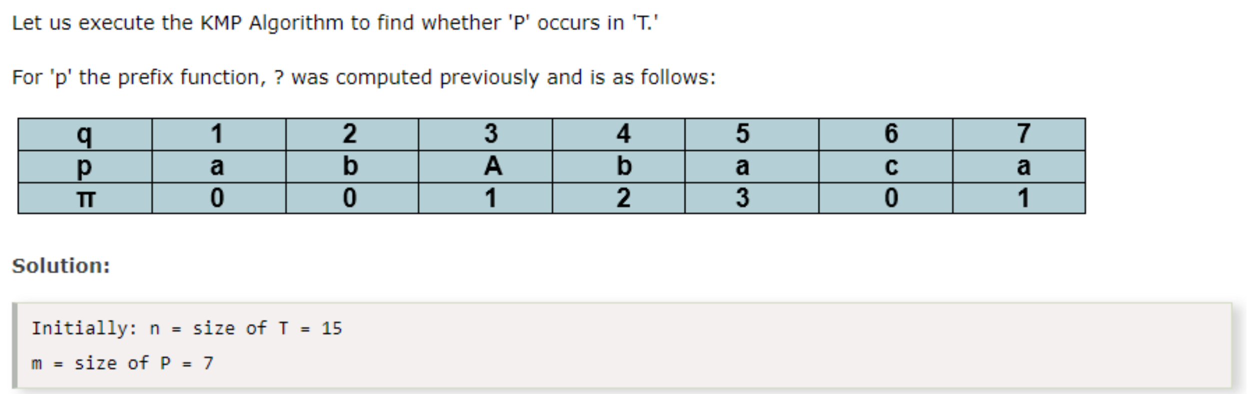
We use the LPS table to decide how many characters are to be skipped for comparison when a mismatch has occurred.
When a mismatch occurs, check the LPS value of the previous character of the mismatched

character in the pattern. If it is '0' then start comparing the first character of the pattern with the next character to the mismatched character in the text. If it is not '0' then start comparing the character which is at an index value equal to the LPS value of the previous character to the mismatched character in pattern with the mismatched character in the Text.

How the KMP Algorithm Works

Let us see a working example of KMP Algorithm to find a Pattern in a Text

EXAMPLE 1



Let us execute the KMP Algorithm to find whether 'P' occurs in 'T.'

For 'p' the prefix function, ? was computed previously and is as follows:

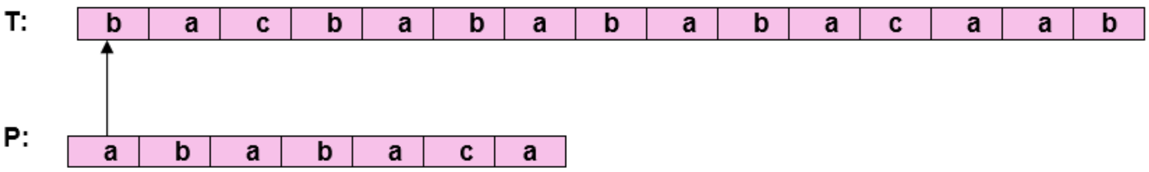
q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

Solution:

```
Initially: n = size of T = 15
m = size of P = 7
```

Step1: i=1, q=0

Comparing P [1] with T [1]



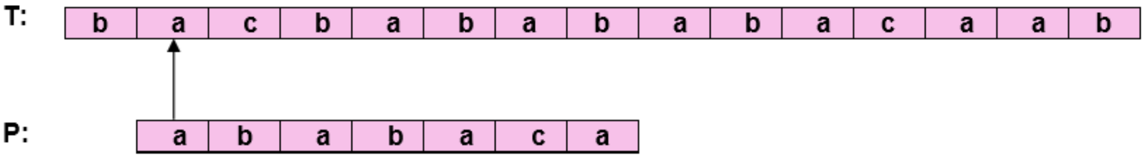
T:	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

P:	a	b	a	b	a	c	a

P [1] does not match with T [1]. ‘p’ will be shifted one position to the right.

Step2: i = 2, q = 0

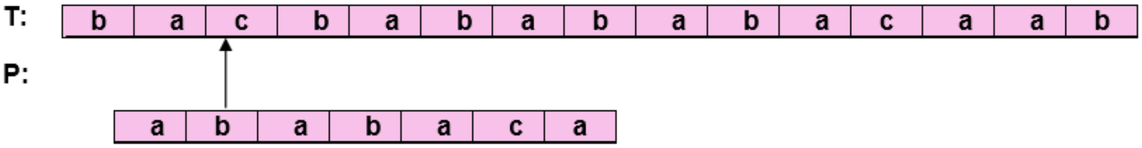
Comparing P [1] with T [2]



P [1] matches T [2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1

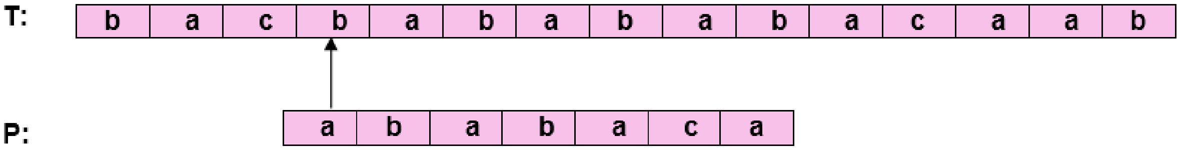
Comparing P [2] with T [3]   P [2] doesn't match with T [3]



Backtracking on p, Comparing P [1] and T [3]

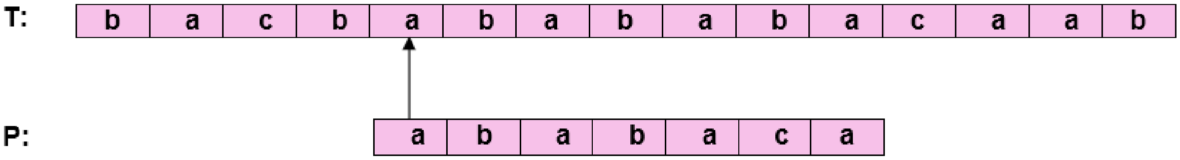
Step4: i = 4, q = 0

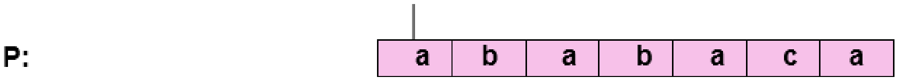
Comparing P [1] with T [4]   P [1] doesn't match with T [4]



Step5: i = 5, q = 0

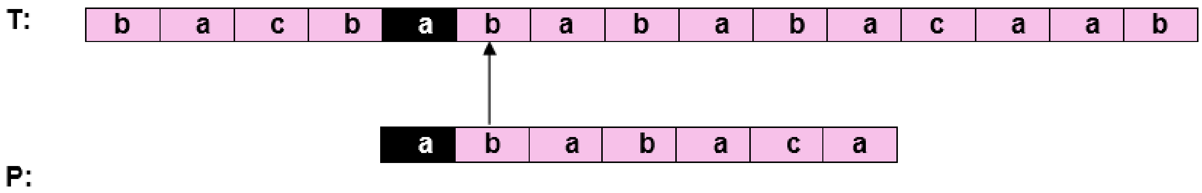
Comparing P [1] with T [5]   P [1] match with T [5]





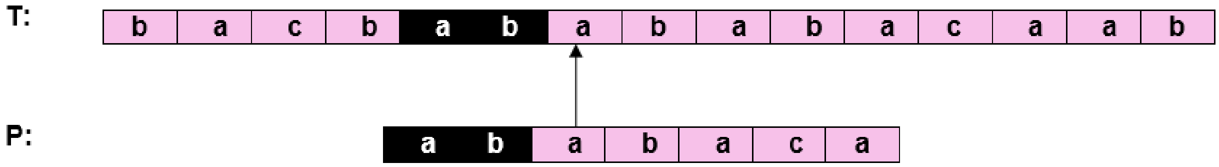
Step6: i = 6, q = 1

Comparing P [2] with T [6] P [2] matches with T [6]



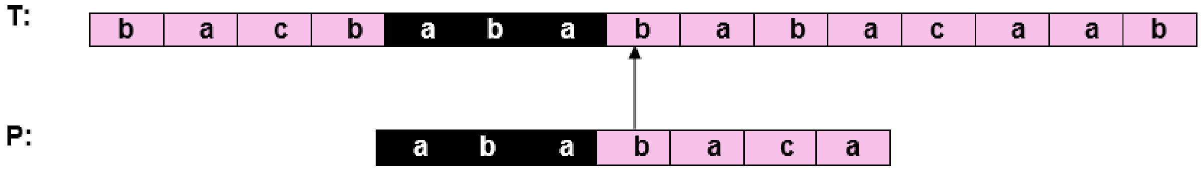
Step7: i = 7, q = 2

Comparing P [3] with T [7] P [3] matches with T [7]



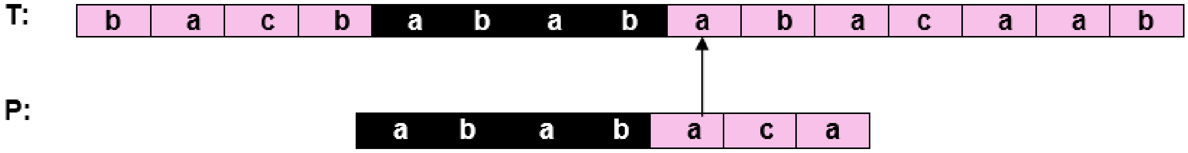
Step8: i = 8, q =3

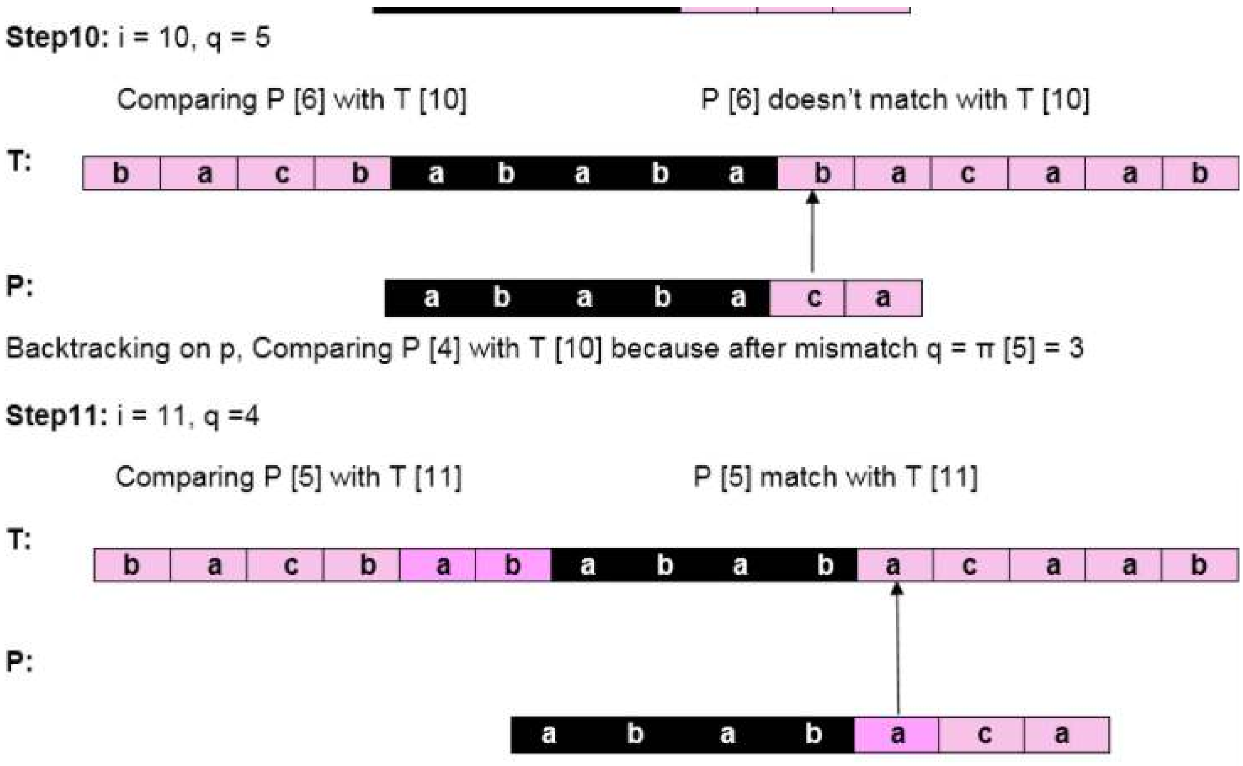
Comparing P [4] with T [8] P [4] matches with T [8]



Step9: i = 9, q = 4

Comparing P [5] with T [9] P [5] matches with T [9]





Step10: i = 10, q = 5

Comparing P [6] with T [10] P [6] doesn't match with T [10]

T: b a c b a b a b a b a c a a b

P: a b a b a c a

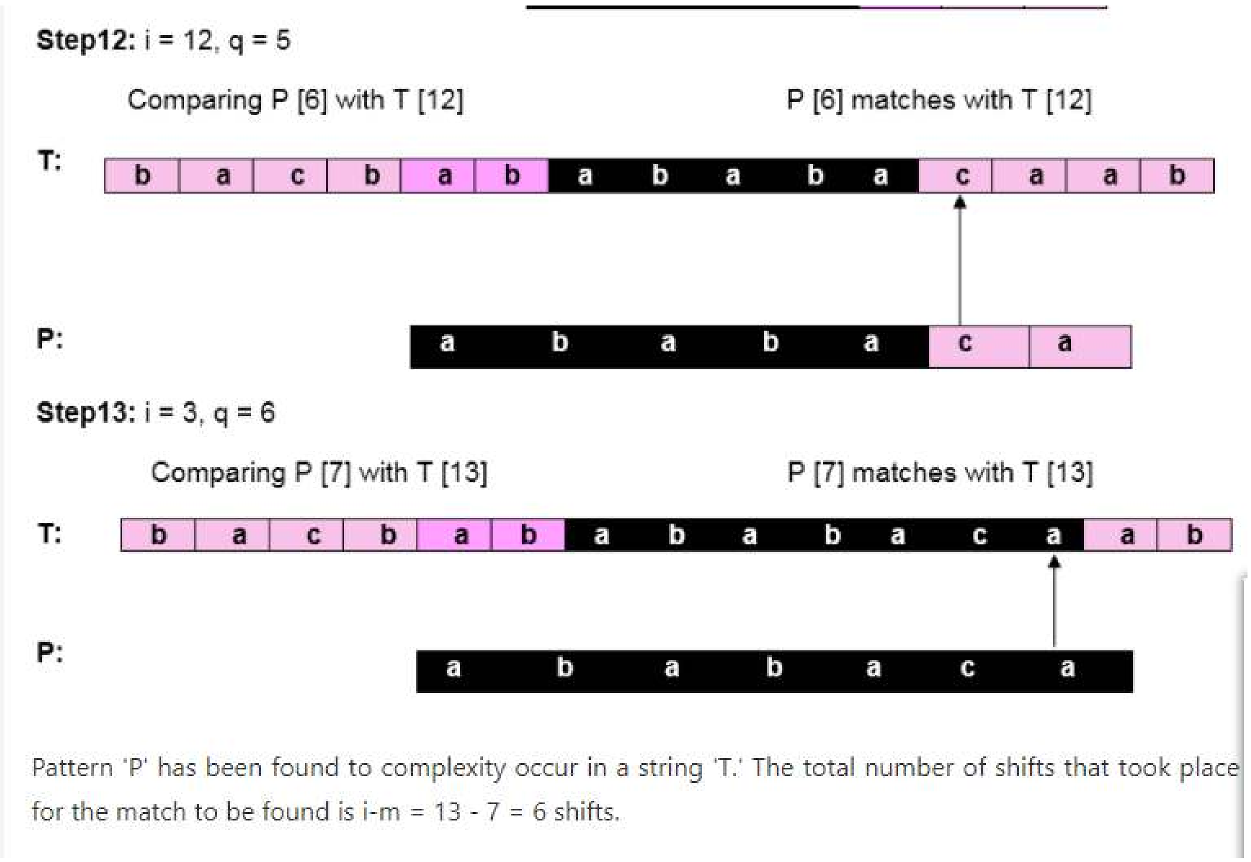
Backtracking on p, Comparing P [4] with T [10] because after mismatch q = π [5] = 3

Step11: i = 11, q =4

Comparing P [5] with T [11] P [5] match with T [11]

T: b a c b a b a b a b a c a a b

P: a b a b a c a



Step12: i = 12, q = 5

Comparing P [6] with T [12] P [6] matches with T [12]

T: b a c b a b a b a b a c a a b

P: a b a b a c a

Step13: i = 3, q = 6

Comparing P [7] with T [13] P [7] matches with T [13]

T: b a c b a b a b a b a c a a b

P: a b a b a c a

Pattern 'P' has been found to complexity occur in a string 'T.' The total number of shifts that took place for the match to be found is i-m = 13 - 7 = 6 shifts.

Example 2

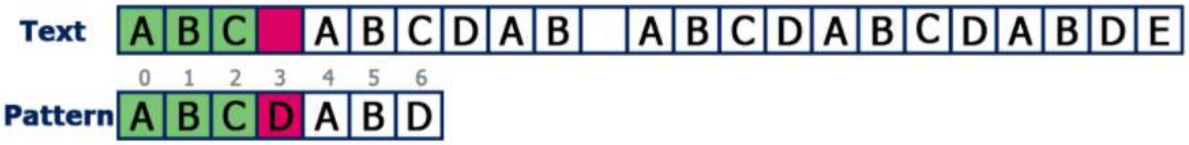
Consider the following Text and Pattern

Text : ABC ABCDAB ABCDABCDABDE
Pattern : ABCDABD

LPS[] table for the above pattern is as follows...

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0

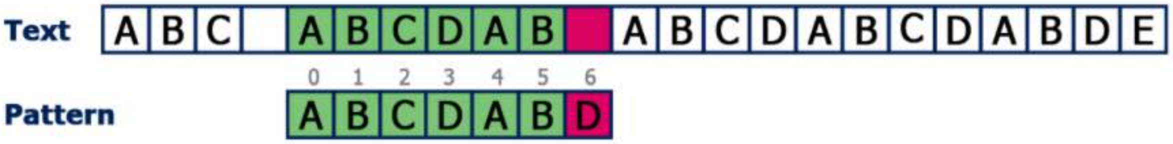
Step 1 - Start comparing first character of Pattern with first character of Text from left to right



Text	A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
	0	1	2	3	4	5	6																
Pattern	A	B	C	D	A	B	D																

Here mismatch occured at Pattern[3], so we need to consider LPS[2] value. Since LPS[2] value is '0' we must compare first character in Pattern with next character in Text.

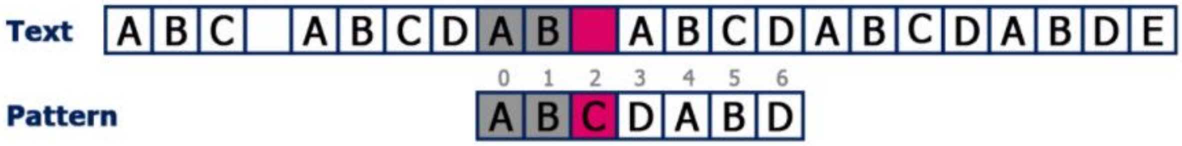
Step 2 - Start comparing first character of Pattern with next character of Text.



Text	A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
					0	1	2	3	4	5	6												
Pattern					A	B	C	D	A	B	D												

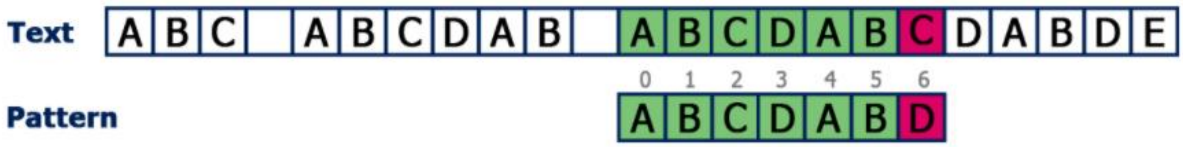
Here mismatch occured at Pattern[6], so we need to consider LPS[5] value. Since LPS[5] value is '2' we compare Pattern[2] character with mismatched character in Text.

Step 3 - Since LPS value is '2' no need to compare Pattern[0] & Pattern[1] values



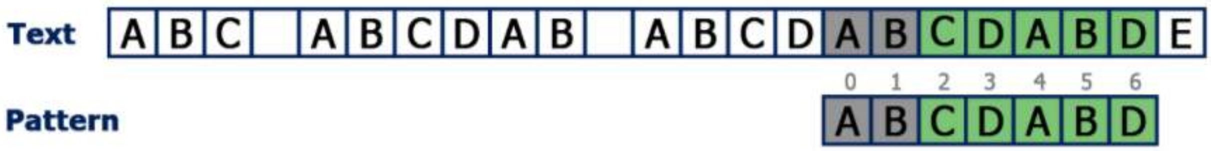
Here mismatch occured at Pattern[2], so we need to consider LPS[1] value. Since LPS[1] value is '0' we must compare first character in Pattern with next character in Text.

Step 4 - Compare Pattern[0] with next character in Text.



Here mismatch occured at Pattern[6], so we need to consider LPS[5] value. Since LPS[5] value is '2' we compare Pattern[2] character with mismatched character in Text.

Step 5 - Compare Pattern[2] with mismatched character in Text.



Here all the characters of Pattern matched with a substring in Text which is starting from index value 15. So we conclude that given Pattern found at index 15 in Text.

KMP ALGORITHM COMPLEXITY

O(m)- it is to compute to prefix function values

O(n)-it is to compare the pattern to the text

O(n+m)- Total time taken by KMP Algorithm.

Advantages

- The running time of KMP algorithm is O(n+m). which is very fast
- The algorithm never needs to move backwards in the input text T. It makes the algorithm good for processing very large files.

Disadvantages

- Does not work well as the size of the alphabet increase. By which more chances of mismatch occurs

TRIES DATA STRUCTURE

Trie is an efficient information re***Trie***val data structure. The term tries comes from the word retrieval

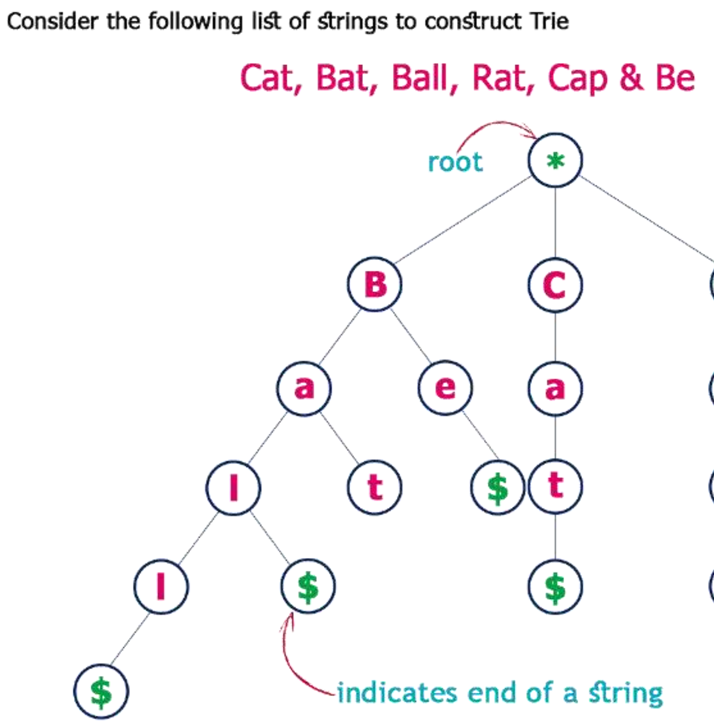
Definition of a Trie

- Data structure for representing a collection of strings
- In computer science , a trie also called digital tree or radix tree or prefix tree.
- Tries support fast string matching.

Properties of Tries

- A Multi way tree
- Each node has from 1 to n children
- Each edge of the tree is labeled with a character
- Each leaf node corresponds to the stored string which is a concatenation of characters on a path from the root to this node.

EXAMPLE


Consider the following list of strings to construct Trie
Cat, Bat, Ball, Rat, Cap & Be
root
*
B
C
R
a
e
a
a
l
t
$
t
t
l
$
$
$
$
indicates end of a string

Trie | (Insert and Search)

Trie is an efficient information retrieval data structure. Using Trie, search complexities can be brought to an optimal limit (key length).
Given multiple strings. The task is to insert the string in a Trie

Examples:

Example 1: str = {"cat", "there", "caller", "their", "calling", “bat”}

```
   root
  /   \
 c     t
 |     |
 a     h
 |\    |
 l t   e
 |     | \
 l     i  r
 |\    |  |
 e i   r  e
 | |
 r n
   |
   g
```

Example 2: str = {"Candy", "cat", "Caller", "calling"}

```
    root
     |
     c
     |
     a
  /  |\
 l   n  t
 |   |
 l   d
 |\  |
 e  i y
 | |
 r  n
    |
    g
```

Approach: An efficient approach is to treat every character of the input key as an individual trie node and insert it into the trie. Note that the children are an array of pointers (or references) to next level trie nodes. The key character acts as an index into the array of children. If the input key is new or an extension of the existing key, we need to construct non-existing nodes of the key, and mark end of the word for the last node. If the input key is a prefix of the existing key in Trie, we simply mark the last node of the key as the end of a word. The key length determines Trie depth.

Trie deletion

Here is an algorithm how to delete a node from trie.
During delete operation we delete the key in bottom up manner using recursion. The following are possible conditions when deleting key from trie,

1. Key may not be there in trie. Delete operation should not modify trie.

2. Key present as unique key (no part of key contains another key (prefix), nor the key itself is prefix of another key in trie). Delete all the nodes.

3. Key is prefix key of another long key in trie. Unmark the leaf node.

4. Key present in trie, having atleast one other key as prefix key. Delete nodes from end of key until first leaf node of longest prefix key.

Time Complexity: The time complexity of the deletion operation is O(n) where n is the key length

Advantages of Trie Data Structure

Tries is a tree that stores strings. The maximum number of children of a node is equal to the size of the alphabet. Trie supports search, insert and delete operations in O(L) time where **L** is the length of the key.

Hashing:- In hashing, we convert the key to a small value and the value is used to index data. Hashing supports search, insert and delete operations in O(L) time on average.

Self Balancing BST : The time complexity of the search, insert and delete operations in a self-balancing Binary Search Tree (BST) (like Red-Black Tree, AVL Tree, Splay Tree, etc) is O(L * Log n) where n is total number words and L is the length of the word. The advantage of Self-balancing BSTs is that they maintain order which makes operations like minimum, maximum, closest (floor or ceiling) and kth largest faster.

Why Trie? :-

1. With Trie, we can insert and find strings in ***O(L)*** time where *L* represent the length of a single word. This is obviously faster than BST. This is also faster than Hashing because of the ways it is implemented. We do not need to compute any hash function. No collision handling is required (like we do in open addressing and separate chaining)

2. Another advantage of Trie is, we can easily print all words in alphabetical order which is not easily possible with hashing.

3. We can efficiently do prefix search (or auto-complete) with Trie.

Issues with Trie :-

The main disadvantage of tries is that they need a lot of memory for storing the strings. For each node we have too many node pointers(equal to number of characters of the alphabet), if space is concerned, then **Ternary Search Tree** can be preferred for dictionary implementations. In Ternary Search Tree, the time complexity of search operation is O(h) where h is the height of the tree. Ternary Search Trees also supports other operations supported by Trie like prefix search, alphabetical order printing, and nearest neighbor search.

The final conclusion is regarding *tries data structure* is that they are faster but require *huge memory* for storing the strings.

APPLICATIONS OF TRIES

String handling and processing are one of the most important topics for programmers. Many real time applications are based on the string processing like:

1. Search Engine results optimization

2. Data Analytics

3. Sentimental Analysis

The data structure that is very important for string handling is the **Trie** data structure that is based on **prefix of string**

TYPES OF TRIES

Tries are classified into three categories:

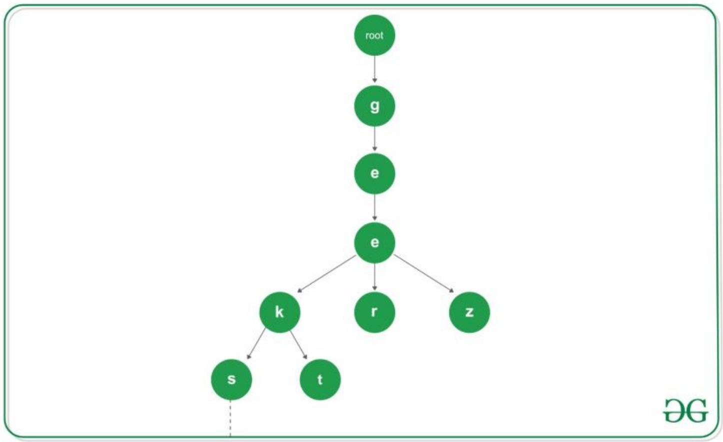
1. Standard Tries
2. Compressed Tries
3. Suffix Tries

STANDARD TRIES

A standard trie have the following properties:}

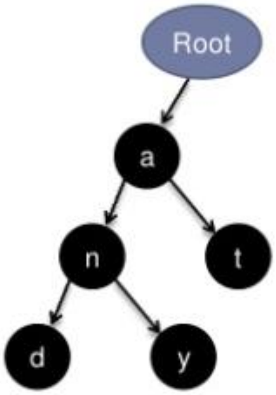
- It is an ordered tree like data structure.
- Each node(except the root node) in a standard trie is labeled with a character.
- The children of a node are in alphabetical order.
- Each node or branch represents a possible character of keys or words.
- Each node or branch may have multiple branches.
- The last node of every key or word is used to mark the end of word or node.
- The path from external node to the root yields the string of S.

Below is the illustration of the Standard Trie



Standard Trie Insertion

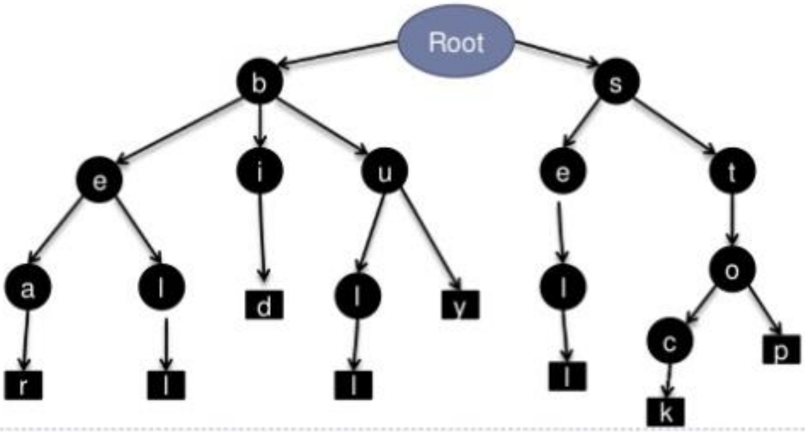
Strings={ a,an,and,any}



Example of Standard Trie

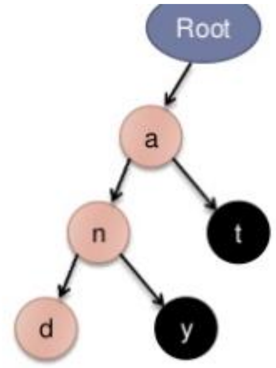
Standard trie for the following strings

S={ bear, bell, bid, bull, buy, sell, stock, stop}



Handling Keys(strings)

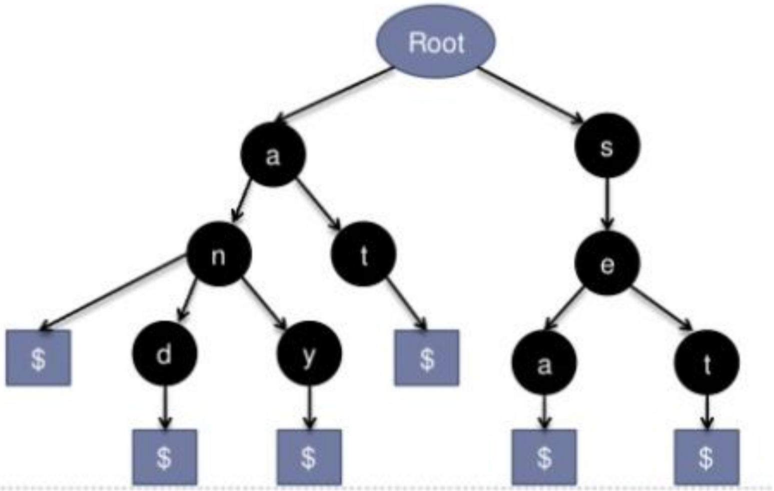
- When a key is prefix of another key
  How can we know that “an “ is a word
  Example : an, and



Standard Trie Searching

Search hit where search node has a $ symbol

- Search - sea

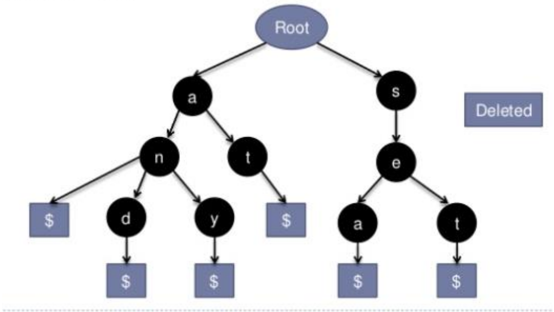


Standard Trie Deletion

To perform the deletion there exist cases

1. Word not found
   Return false
2. Word exist as a standalone word
   I. Part of any other node
   Example:

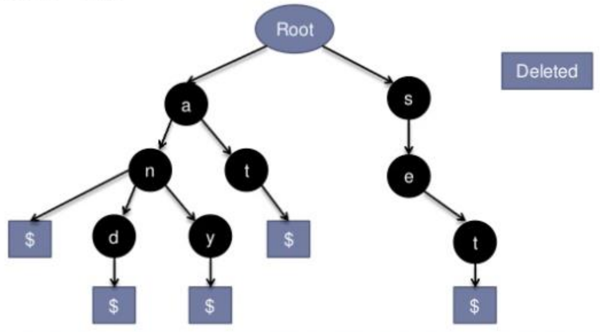
Delete - sea



II. Does not part of any other node

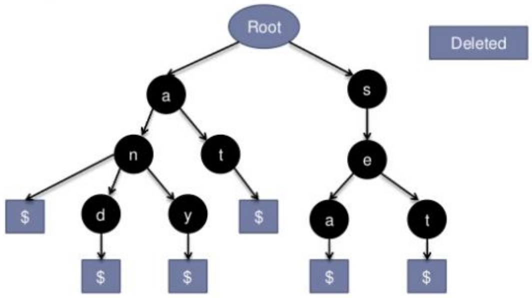
EXAMPLE

Delete - set



3. Word exist as a prefix of another word.

Delete - an



COMPRESSED TRIE

A Compressed trie have the following properties:

1. A Compressed Trie is an advanced version of the standard trie.
2. Each nodes(except the **leaf** nodes) have atleast 2 children.
3. It is used to achieve space optimization.
4. To derive a Compressed Trie from a Standard Trie, compression of chains of redundant nodes is performed.

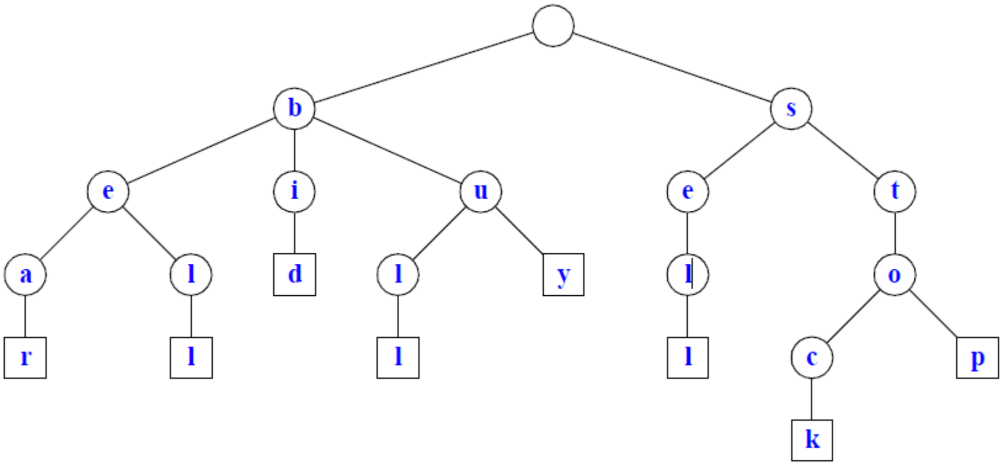
5. It consists of grouping, re-grouping and un-grouping of keys of characters.

6. While performing the insertion operation, it may be required to un-group the already grouped characters.

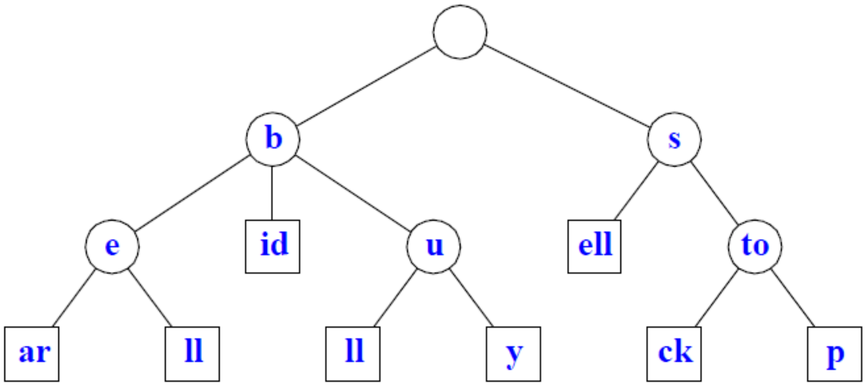
7. While performing the deletion operation, it may be required to re-group the already grouped characters.

Compressed trie is constructed from standard trie

- Standard Trie:



- Compressed Trie:

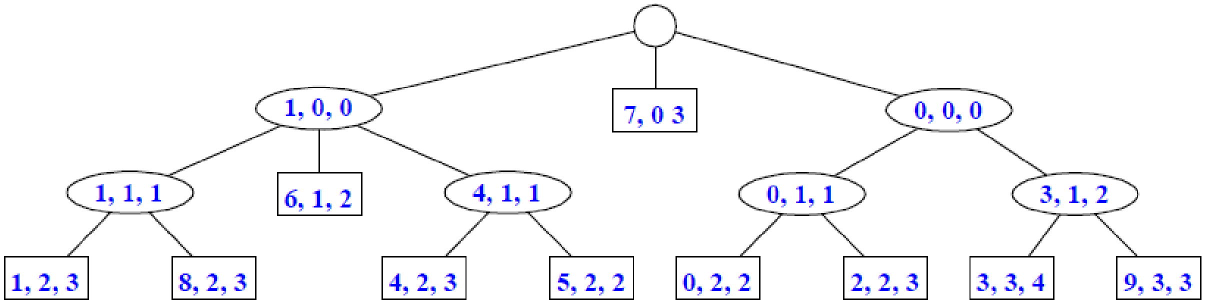


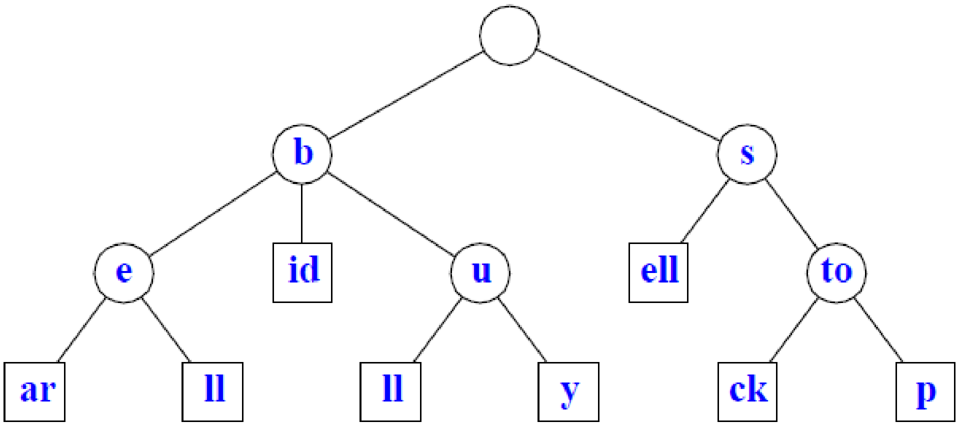
Storage of Compressed Trie

A compressed Trie can be stored at O9s) where s= | S| by using O(1) Space index ranges at the nodes

In the below representation each node is represented with (I,j,k) value
I ---- indicate index of the string
j—starting index of the character of string I
k--- ending index of the character of the string I
Ex: In the given diagram node (4,2,3) having the characters**(ll)** which belongs to s[4] so i=4, index of l character in s[4] is 2 so j=2 and ending index is 3 so k=3

	0	1	2	3	4
S[0] =	s	e	e		
S[1] =	b	e	a	r	
S[2] =	s	e	l	l	
S[3] =	s	t	o	c	k
S[4] =	b	u	l	l	
S[5] =	b	u	y		
S[6] =	b	i	d		
S[7] =	h	e	a	r	
S[8] =	b	e	l	l	
S[9] =	s	t	o	p	





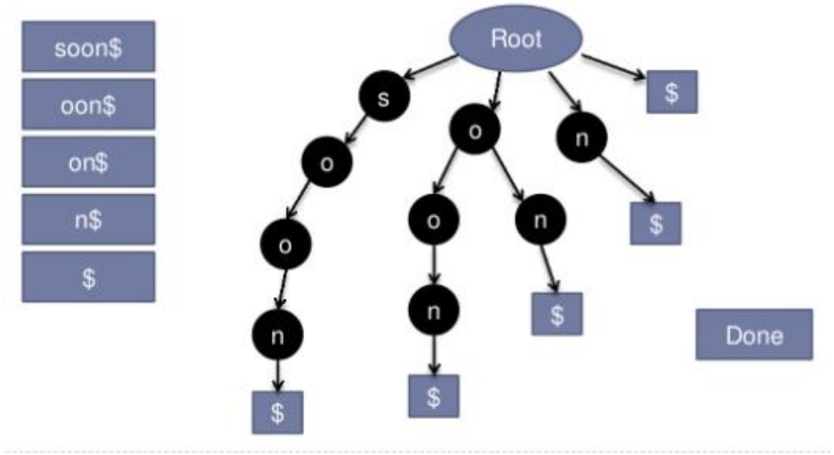
SUFFIX TRIES

A Suffix trie have the following properties:

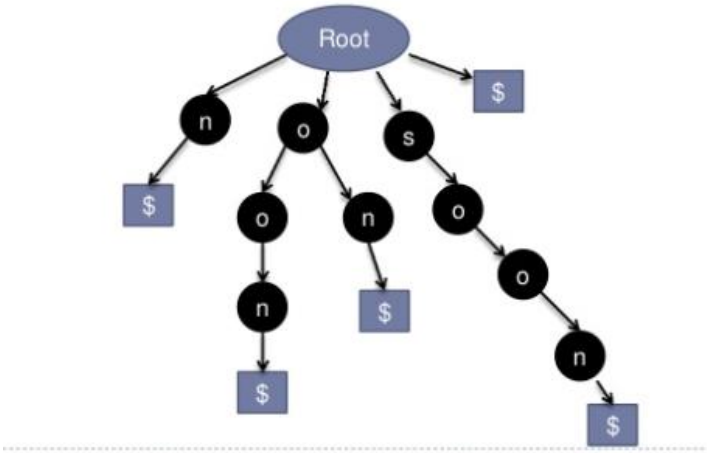
1. Suffix trie is a compressed trie for all the suffixes of the text
2. Suffix trie are space efficient data structure to store a string that allows many kinds of queries to be answered quickly.

Example

Let us consider an example text “soon$”



After alphabetically order the trie look like



Advantages of suffix tries

1. Insertion is faster compared to the hash table
2. Look up is faster than hash table implementation
3. There are no collision of different keys in tries

UNIT - V

Pattern Matching and Tries: Pattern matching algorithms-Brute force, the Boyer –Moore algorithm, the Knuth-Morris-Pratt algorithm, Standard Tries, Compressed Tries, Suffix tries.

Pattern Matching

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or 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], 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

Different Types of Pattern Matching Algorithms

1. Navie Based Algorithm or Brute Force Algorithm
2. Boyer Moore Algorithm
3. Knuth-Morris Pratt (KMP) Algorithm

Navie Based Algorithm or Brute Force Algorithm

When we talk about a string matching algorithm, every one can get a simple string matching technique. That is starting from first letters of the text and first letter of the pattern check whether these two letters are equal. if it is, then check second letters of the text and pattern. If it is not equal, then move first letter of the pattern to the second letter of the text. then check these two letters. this is the simple technique everyone can thought.

Brute Force string matching algorithm is also like that. Therefore we call that as Naive string matching algorithm. Naive means basic.

Brute Force Algorithm

```
do
    if (text letter == pattern letter)
        compare next letter of pattern to next letter of text
    else
        move pattern down text by one letter
while (entire pattern found or end of text)
```

Lets learn this method using an example.

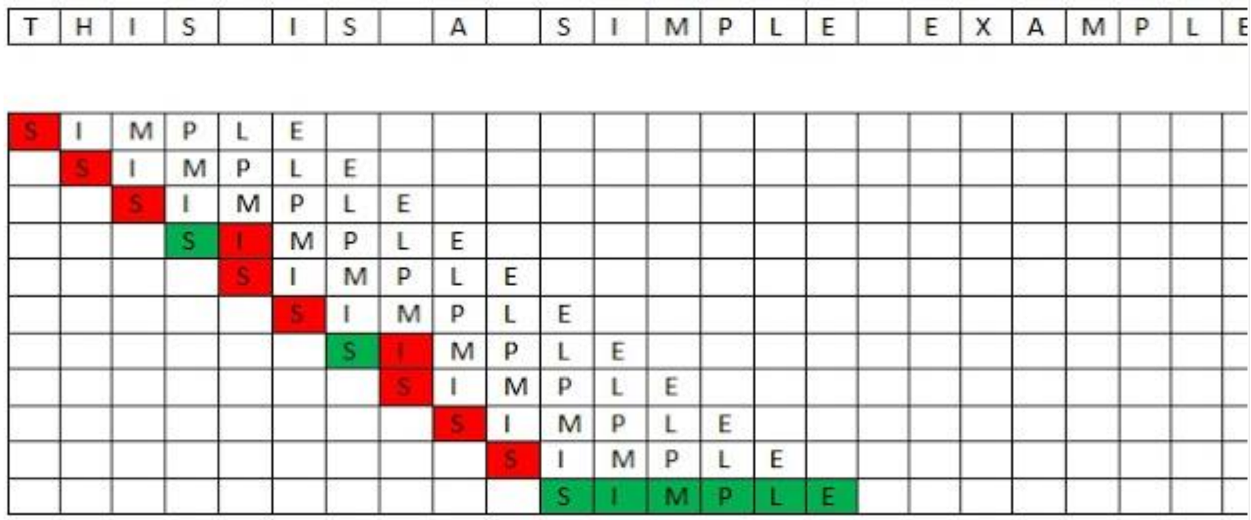
EXAMPLE 1

Let our text (T) as,

THIS IS A SIMPLE EXAMPLE

and our pattern (P) as,

SIMPLE



T	H	I	S		I	S		A		S	I	M	P	L	E		E	X	A	M	P	L	E
🟥S	I	M	P	L	E																		
	🟥S	I	M	P	L	E																	
		🟥S	I	M	P	L	E																
			🟩S	🟥I	M	P	L	E															
				🟥S	I	M	P	L	E														
					🟥S	I	M	P	L	E													
						🟩S	🟥I	M	P	L	E												
							🟥S	I	M	P	L	E											
								🟥S	I	M	P	L	E										
									🟥S	I	M	P	L	E									
										🟩S	🟩I	🟩M	🟩P	🟩L	🟩E								

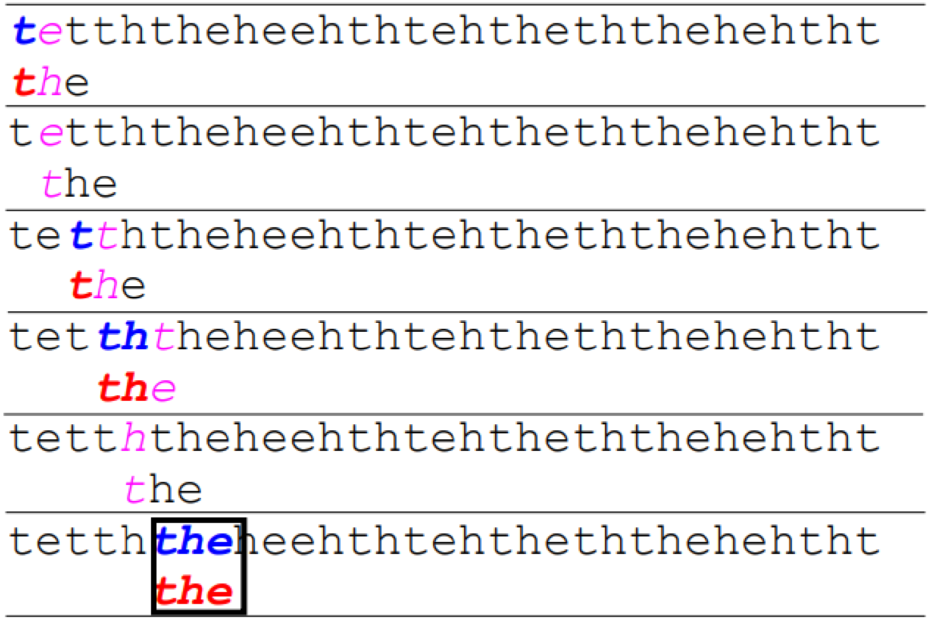
Red Boxes-Mismatch   Green Boxes-Match

In above red boxes says mismatch letters against letters of the text and green boxes says match letters against letters of the text. According to the above

In first raw we check whether first letter of the pattern is matched with the first letter of the text. It is mismatched, because "S" is the first letter of pattern and "T" is the first letter of text. Then we move the pattern by one position. Shown in second raw.

Then check first letter of the pattern with the second letter of text. It is also mismatched. Likewise we continue the checking and moving process. In fourth raw we can see first letter of the pattern matched with text. Then we do not do any moving but we increase testing letter of the pattern. We only move the position of pattern by one when we find mismatches. Also in last raw, we can see all the letters of the pattern matched with the some letters of the text continuously.

Example 2



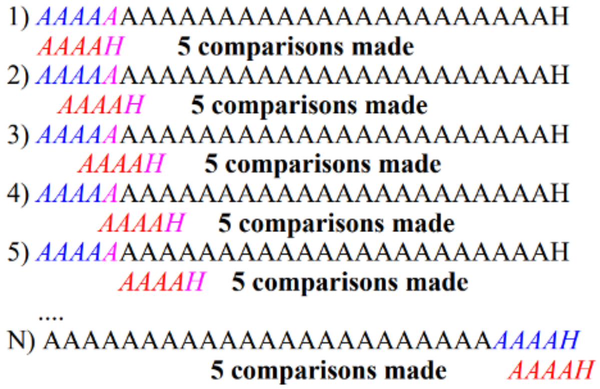
```
tetththeheehthtehtheththehehtht
the
tetththeheehthtehtheththehehtht
 the
tetththeheehthtehtheththehehtht
  the
tetththeheehthtehtheththehehtht
   the
tetththeheehthtehtheththehehtht
    the
tetththeheehthtehtheththehehtht
     the
```

Running Time Analysis Of Brute Force String Matching Algorithm

Worst Case

Given a pattern M characters in length, and a text N characters in length...
• Worst case: compares pattern to each substring of text of length M.
For example, M=5.

- Total number of comparisons: M (N-M+1) • Worst case time complexity: O(MN)



1) *AAAAA*AAAAAAAAAAAAAAAAAAAAAAH
AAAAH **5 comparisons made**
2) *AAAAA*AAAAAAAAAAAAAAAAAAAAAAH
AAAAH **5 comparisons made**
3) *AAAAA*AAAAAAAAAAAAAAAAAAAAAAH
AAAAH **5 comparisons made**
4) *AAAAA*AAAAAAAAAAAAAAAAAAAAAAH
AAAAH **5 comparisons made**
5) *AAAAA*AAAAAAAAAAAAAAAAAAAAAAH
AAAAH **5 comparisons made**
....
N) AAAAAAAAAAAAAAAAAAAAAAA*AAAAH*
5 comparisons made *AAAAH*

- Total number of comparisons: M (N-M+1)
- Worst case time complexity: O(MN)

Best case

Given a pattern M characters in length, and a text N characters in length...

- **Best case if pattern found**: Finds pattern in first M positions of text.

For example, M=5.

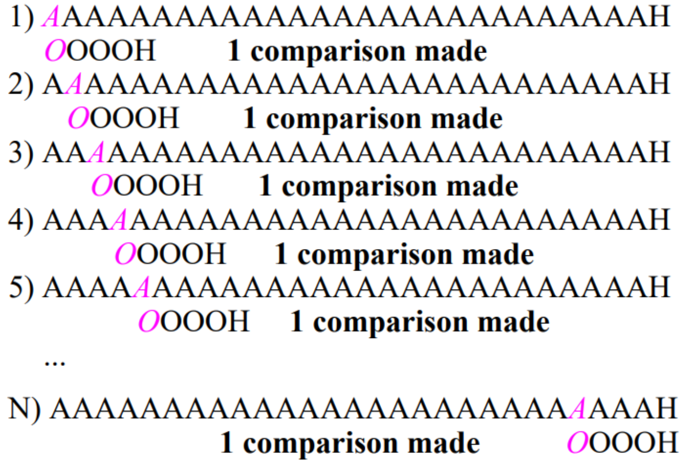
AAAAAAAAAAAAAAAAAAAAAAAAAAAAH

AAAAA 5 comparisons made

- Total number of comparisons: M
- Best case time complexity: O(M)

Best case if pattern not found:

Always mismatch on first character. For example, M=5.



1) *A*AAAAAAAAAAAAAAAAAAAAAAAAAAAH
*O*OOOH **1 comparison made**
2) A*A*AAAAAAAAAAAAAAAAAAAAAAAAAAH
*O*OOOH **1 comparison made**
3) AA*A*AAAAAAAAAAAAAAAAAAAAAAAAAH
*O*OOOH **1 comparison made**
4) AAA*A*AAAAAAAAAAAAAAAAAAAAAAAAH
*O*OOOH **1 comparison made**
5) AAAA*A*AAAAAAAAAAAAAAAAAAAAAAAH
*O*OOOH **1 comparison made**
...
N) AAAAAAAAAAAAAAAAAAAAAAAA*A*AAAH
1 comparison made *O*OOOH

- Total number of comparisons: N
- Best case time complexity: O(N)

Advantages

1. Very simple technique and also that does not require any preprocessing. Therefore total running time is the same as its matching time.

Disadvantages

1. Very inefficient method. Because this method takes only one position movement in each time

Boyer Moore Algorithm for Pattern Searching

The B-M algorithm takes a backward approach . the pattern string(p) is aligned with the start of the text string(T) and then compare the characters of pattern from right to left beginning with rightmost character

If a character is compared that is not within the pattern, no match can be found by comparing any furher characters at this position so the pattern can be shifted completely past the mismatching character.

For determining the possible shifts , B-M algorithm uses 2 preprocessing strategies simultaneously whenever a mismatch occurs, the algorithm computes a shift using both strategies and selects the longer one. thus it makes use of the most efficient stategy for each individual case

NOTE : Boyer Moore algorithm starts matching from the last character of the pattern.

The 2 strategies are called heuristics of B-M as they are used to reduce the search. They are

1) Bad Character Heuristic
2) Good Suffix Heuristic

Bad Character Heuristic

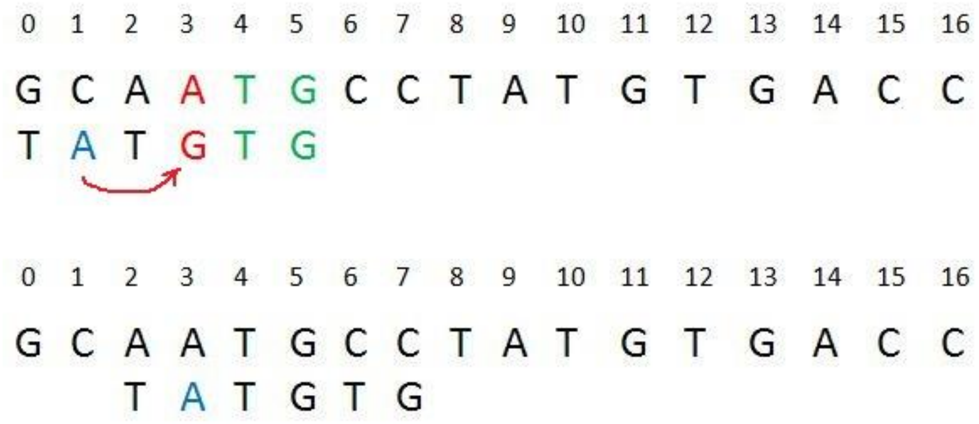
The idea of bad character heuristic is simple. The character of the text which doesn't match with the current character of the pattern is called the **Bad Character**. Upon mismatch, we shift the pattern until –

1) The mismatch becomes a match

2) Pattern P move past the mismatched character.

Case 1 – Mismatch become match

We will lookup the position of last occurrence of mismatching character in pattern and if mismatching character exist in pattern then we'll shift the pattern such that it get aligned to the mismatching character in text T.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
G	C	A	A	T	G	C	C	T	A	T	G	T	G	A	C	C
T	A	T	G	T	G											

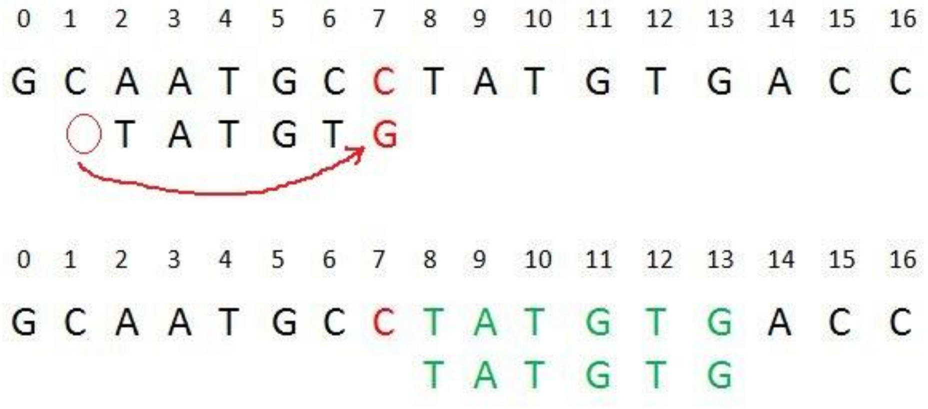
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
G	C	A	A	T	G	C	C	T	A	T	G	T	G	A	C	C
		T	A	T	G	T	G									

case 1

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.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
G	C	A	A	T	G	C	C	T	A	T	G	T	G	A	C	C
	○	T	A	T	G	T	G									

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
G	C	A	A	T	G	C	C	T	A	T	G	T	G	A	C	C
								T	A	T	G	T	G			

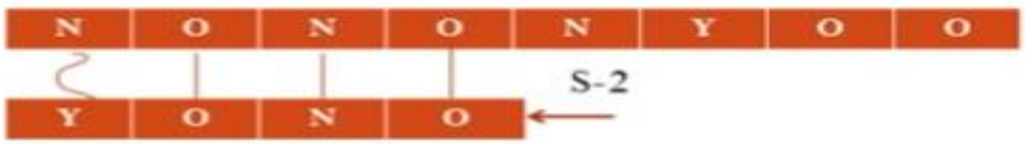
case2

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" do not exist in pattern so at every shift before position 7 we will get mismatch and our search will be fruitless.

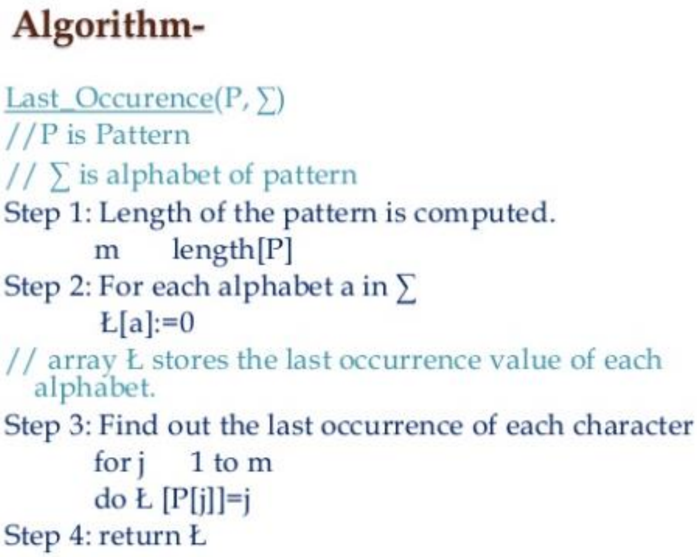
Problem in Bad Character Heuristic

In some cases Bad Character Heuristic produces negative results

For Example:



This means we need some extra information to produce a shift an encountering a bad character. The information is about last position of evry character in the pattern and also the set of every character in the pattern and also the set of characters used in the pattern



Algorithm-

```
Last_Occurence(P, ∑)
//P is Pattern
// ∑ is alphabet of pattern
Step 1: Length of the pattern is computed.
         m      length[P]
Step 2: For each alphabet a in ∑
          Ł[a]:=0
// array Ł stores the last occurrence value of each
   alphabet.
Step 3: Find out the last occurrence of each character
         for j     1 to m
         do Ł [P[j]]=j
Step 4: return Ł
```

2.Good Suffix Heuristic

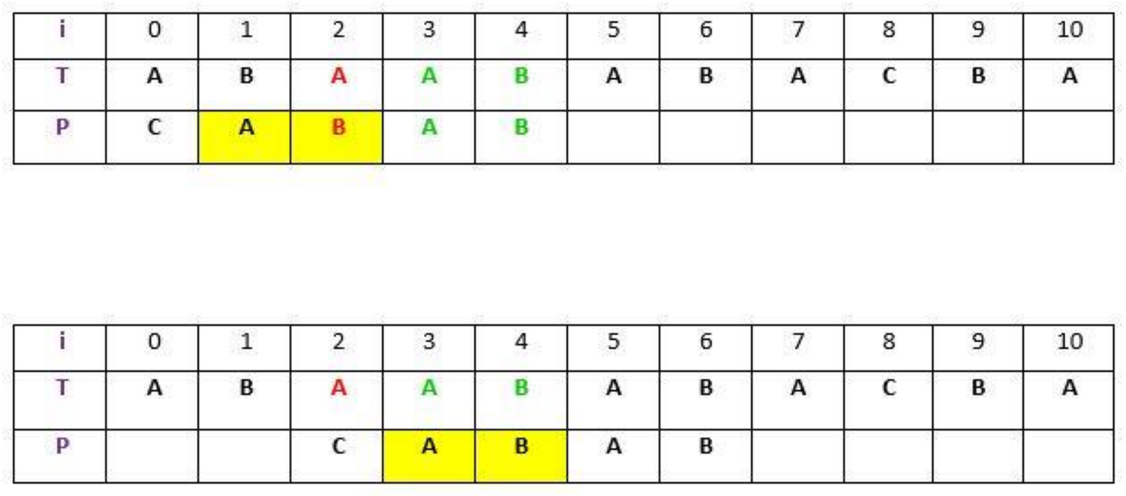
Let **t** be substring of text **T** which is matched with substring of pattern **P**. Now we shift pattern until :

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-



i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P	C	A	B	A	B						

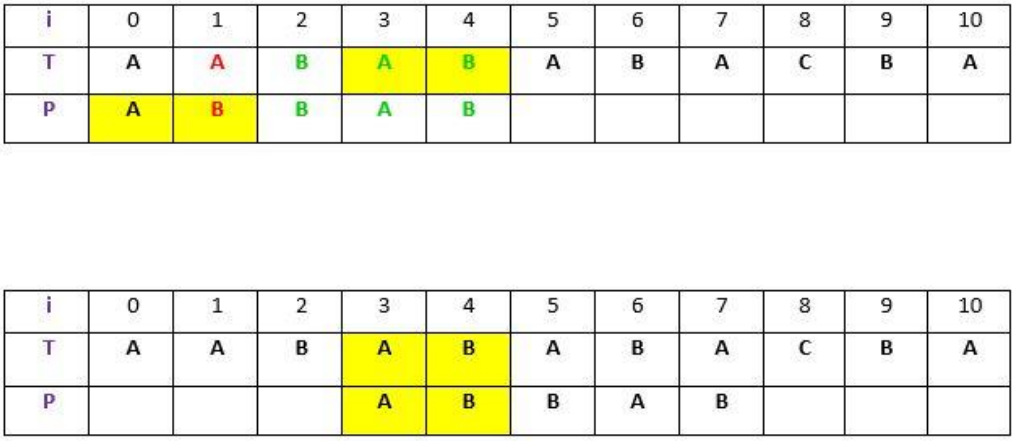
i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P			C	A	B	A	B				

Figure – Case 1

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

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 –



i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	B	A	B	A	B	A	C	B	A
P	A	B	B	A	B						

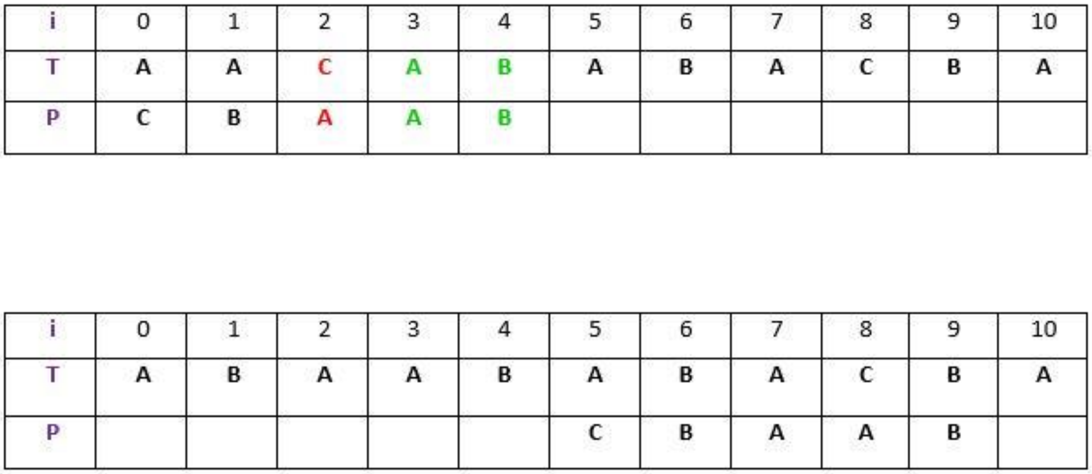
i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	B	A	B	A	B	A	C	B	A
P				A	B	B	A	B			

Figure – Case 2

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 –



i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	C	A	B	A	B	A	C	B	A
P	C	B	A	A	B						

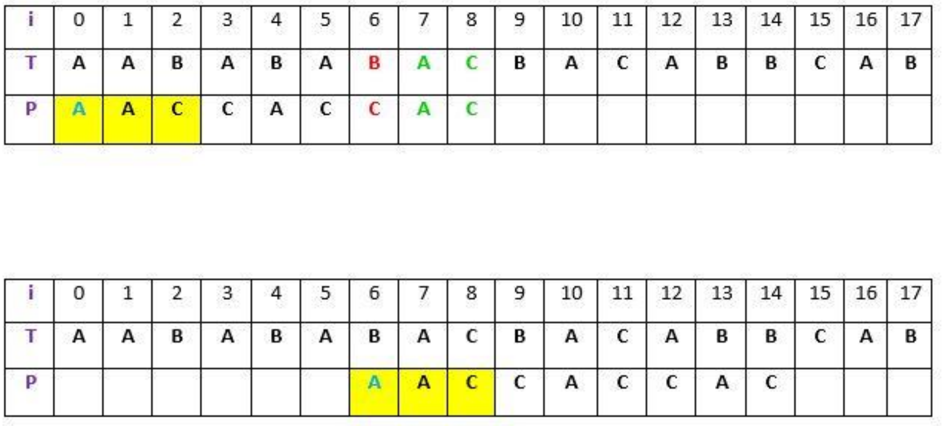
i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P						C	B	A	A	B	

Figure – Case 3

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 –



i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T	A	A	B	A	B	A	B	A	C	B	A	C	A	B	B	C	A	B
P	A	A	C	C	A	C	C	A	C									

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T	A	A	B	A	B	A	B	A	C	B	A	C	A	B	B	C	A	B
P							A	A	C	C	A	C	C	A	C			

Figure – strong suffix rule

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 **ϕ** beginning at position m has no border, so **bpos[m]** is set to **m+1** where **m** is the length of the pattern.

The shift position is obtained by the borders which cannot be extended to the left.

Complexity of Boyer Moore Algorithm

This algorithm takes o(mn) in the worst case and O(nlog(m)/m) on average case, which is the sub linear in the sense that not all characters are inspected

Applications

This algorithm is highly useful in tasks like recursively searching files for virus patterns,searching databases for keys or data ,text and word processing and any other task that requires handling large amount of data at very high speed

Knuth-Morris Pratt (KMP) Algorithm for Pattern Searching

The Naive pattern searching algorithm doesn't work well in cases where we see many matching characters followed by a mismatching character. Following are some examples.

txt[] = "AAAAAAAAAAAAAAAAAB"

pat[] = "AAAAB"

txt[] = "ABABABCABABABCABABABC"

pat[] = "ABABAC" (not a worst case, but a bad case for Naive

KMP Algorithm is one of the most popular patterns matching algorithms. KMP stands for Knuth Morris Pratt. KMP algorithm was invented by Donald Knuth and Vaughan Pratt together and independently by James H Morris in the year 1970. In the year 1977, all the three jointly published KMP Algorithm.

KMP algorithm was the first linear time complexity algorithm for string matching.
KMP algorithm is one of the string matching algorithms used to find a Pattern in a Text.

KMP algorithm is used to find a "Pattern" in a "Text". This algorithm campares character by character from left to right. But whenever a mismatch occurs, it uses a preprocessed table called "**Prefix Table**" to skip characters comparison while matching. Some times prefix table is also known as **LPS Table**. Here LPS stands for "**Longest proper Prefix which is also Suffix".**

Steps for Creating LPS Table (Prefix Table)

- **Step 1** - Define a one dimensional array with the size equal to the length of the Pattern. (LPS[size])
- **Step 2** - Define variables i & j. Set i = 0, j = 1 and LPS[0] = 0.
- **Step 3 -** Compare the characters at Pattern[i] and Pattern[j].
- **Step 4** - If both are matched then set LPS[j] = i+1 and increment both i & j values by one. Goto to Step 3.
- **Step 5** - If both are not matched then check the value of variable 'i'. If it is '0' then set LPS[j] = 0 and increment 'j' value by one, if it is not '0' then set i = LPS[i-1]. Goto Step 3.
- **Step 6**- Repeat above steps until all the values of LPS[] are filled.

Let us use above steps to create prefix table for a pattern...

Example for creating KMP Algorithm's LPS Table (Prefix Table)

Consider the following Pattern

	0	1	2	3	4	5	6
Pattern :	A	B	C	D	A	B	D

Let us define LPS[] table with size 7 which is equal to length of the Pattern

	0	1	2	3	4	5	6
LPS							

Step 1 - Define variables i & j. Set i = 0, j = 1 and LPS[0] = 0.

	0	1	2	3	4	5	6
LPS	0						

i = 0 and j = 1

Step 2 - Campare Pattern[i] with Pattern[j] ===> A with B.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0					

i = 0 and j = 2

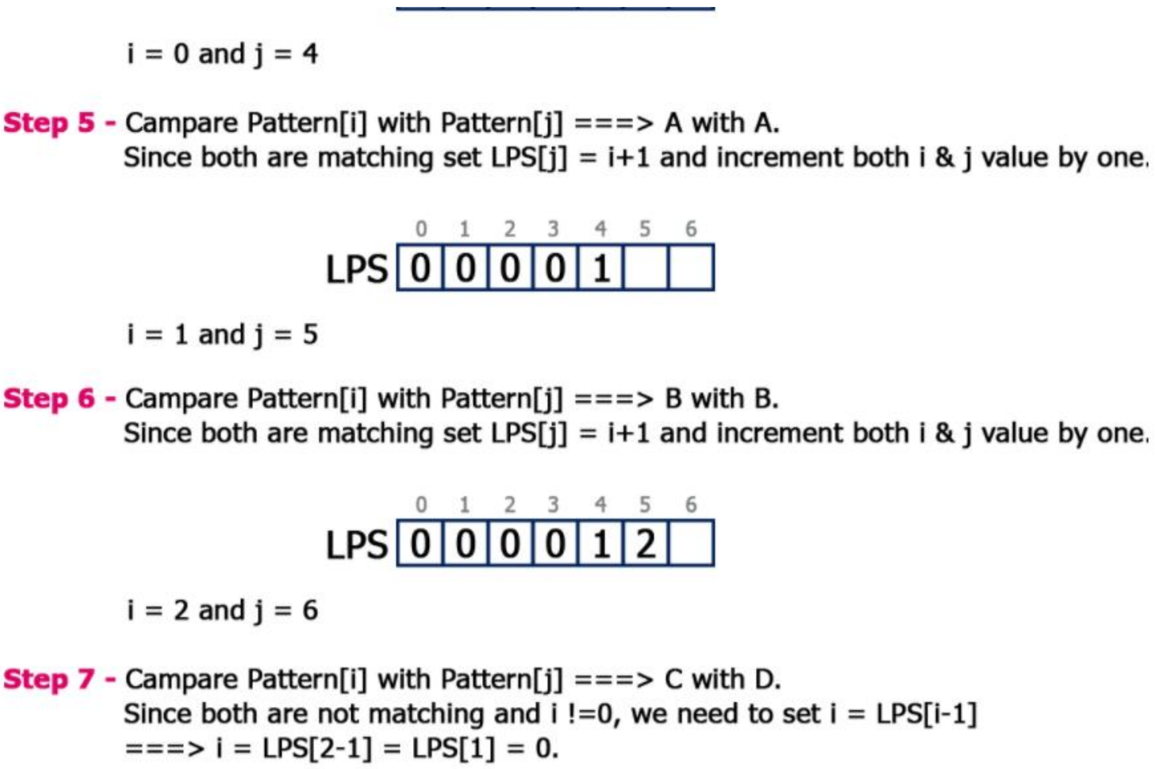
Step 3 - Campare Pattern[i] with Pattern[j] ===> A with C.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0	0				

i = 0 and j = 3

Step 4 - Campare Pattern[i] with Pattern[j] ===> A with D.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0			



i = 0 and j = 4

Step 5 - Campare Pattern[i] with Pattern[j] ===> A with A.
Since both are matching set LPS[j] = i+1 and increment both i & j value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1		

i = 1 and j = 5

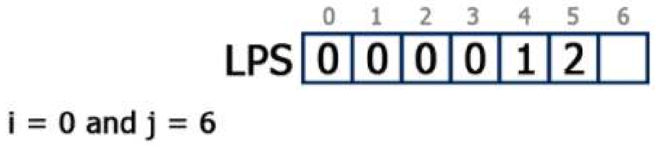
Step 6 - Campare Pattern[i] with Pattern[j] ===> B with B.
Since both are matching set LPS[j] = i+1 and increment both i & j value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	

i = 2 and j = 6

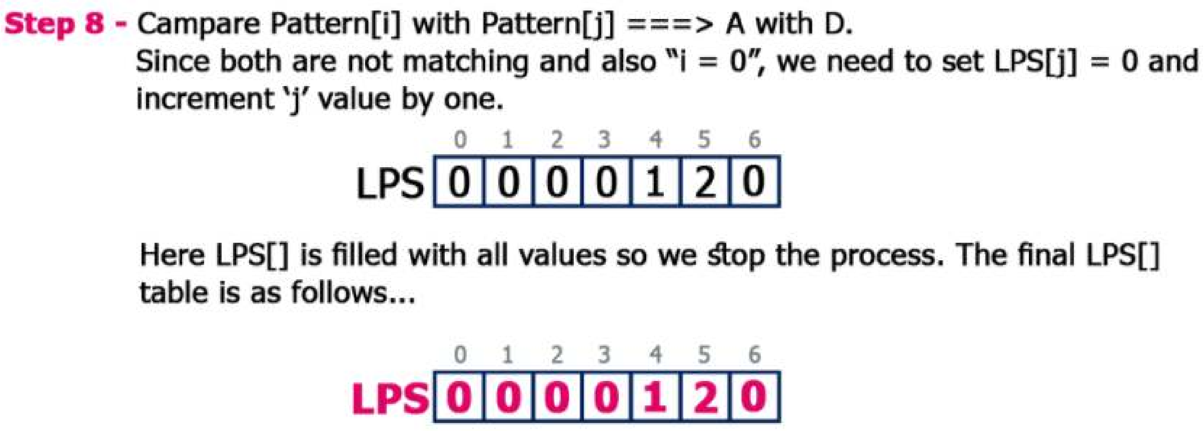
Step 7 - Campare Pattern[i] with Pattern[j] ===> C with D.
Since both are not matching and i !=0, we need to set i = LPS[i-1]
===> i = LPS[2-1] = LPS[1] = 0.

===> i = LPS[2-1] = LPS[1] = 0.



	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	

i = 0 and j = 6



Step 8 - Campare Pattern[i] with Pattern[j] ===> A with D.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0

Here LPS[] is filled with all values so we stop the process. The final LPS[] table is as follows...

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0

How to use LPS Table

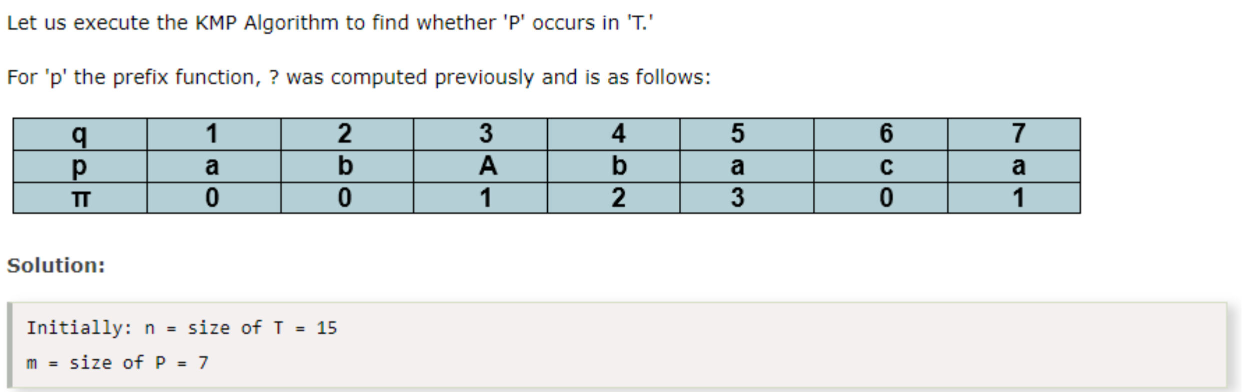
We use the LPS table to decide how many characters are to be skipped for comparison when a mismatch has occurred.

When a mismatch occurs, check the LPS value of the previous character of the mismatched character in the pattern. If it is '0' then start comparing the first character of the pattern with the next character to the mismatched character in the text. If it is not '0' then start comparing the character which is at an index value equal to the LPS value of the previous character to the mismatched character in pattern with the mismatched character in the Text.

How the KMP Algorithm Works

Let us see a working example of KMP Algorithm to find a Pattern in a Text

EXAMPLE 1



Let us execute the KMP Algorithm to find whether 'P' occurs in 'T.'

For 'p' the prefix function, ? was computed previously and is as follows:

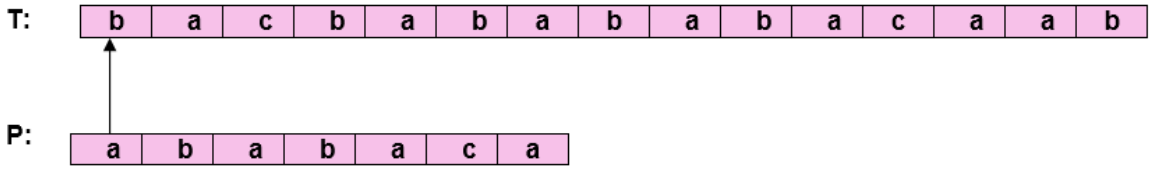
q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

Solution:

```
Initially: n = size of T = 15
m = size of P = 7
```

Step1: i=1, q=0

Comparing P [1] with T [1]



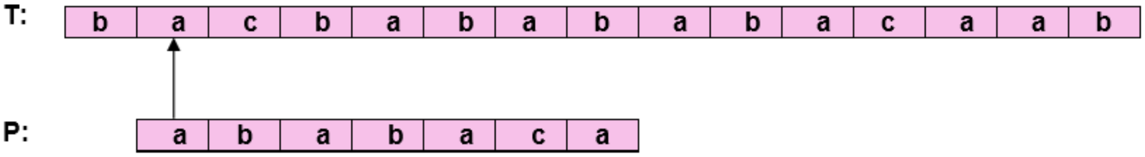
T:	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

P:	a	b	a	b	a	c	a

P [1] does not match with T [1]. ‘p’ will be shifted one position to the right.

Step2: i = 2, q = 0

Comparing P [1] with T [2]



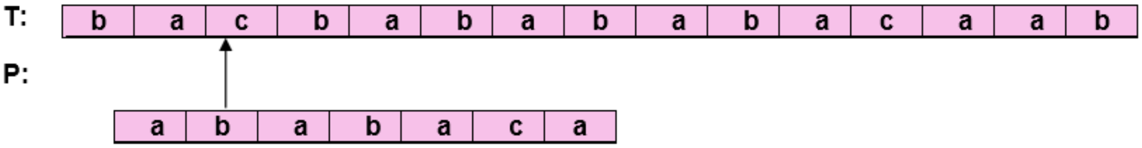
T: b a c b a b a b a b a c a a b

P: a b a b a c a

P [1] matches T [2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1

Comparing P [2] with T [3]   P [2] doesn't match with T [3]



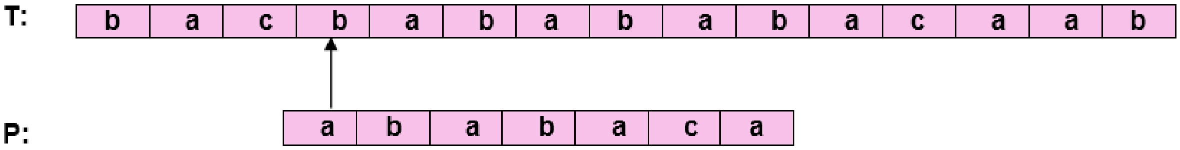
T: b a c b a b a b a b a c a a b

P: a b a b a c a

Backtracking on p, Comparing P [1] and T [3]

Step4: i = 4, q = 0

Comparing P [1] with T [4]   P [1] doesn't match with T [4]

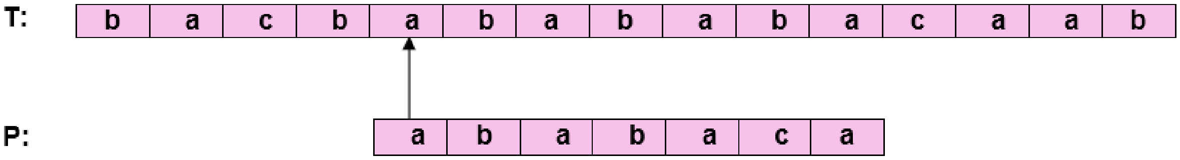


T: b a c b a b a b a b a c a a b

P: a b a b a c a

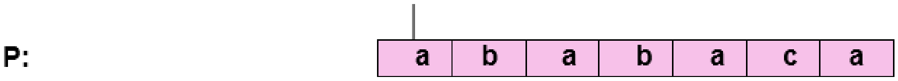
Step5: i = 5, q = 0

Comparing P [1] with T [5]   P [1] match with T [5]



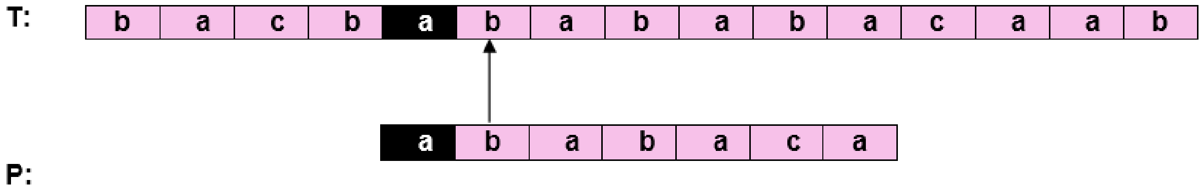
T: b a c b a b a b a b a c a a b

P: a b a b a c a



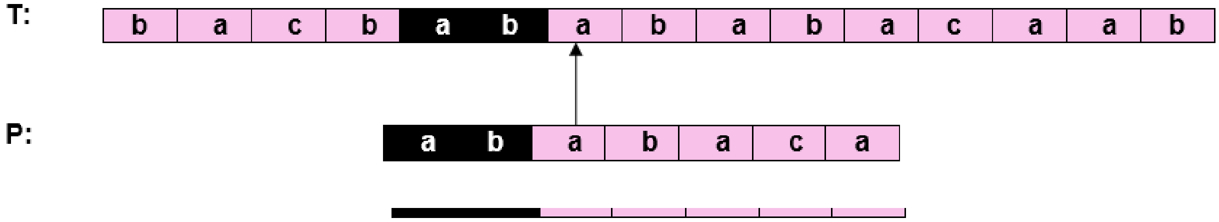
Step6: i = 6, q = 1

Comparing P [2] with T [6]   P [2] matches with T [6]



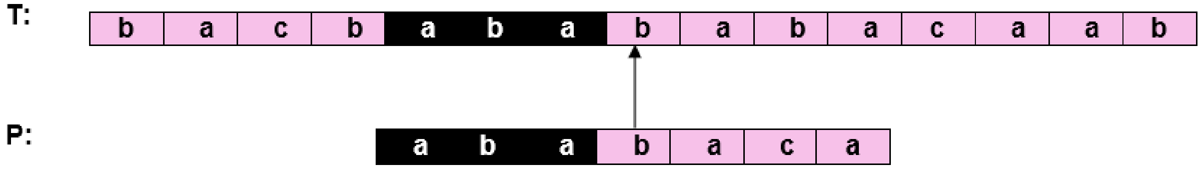
Step7: i = 7, q = 2

Comparing P [3] with T [7]   P [3] matches with T [7]



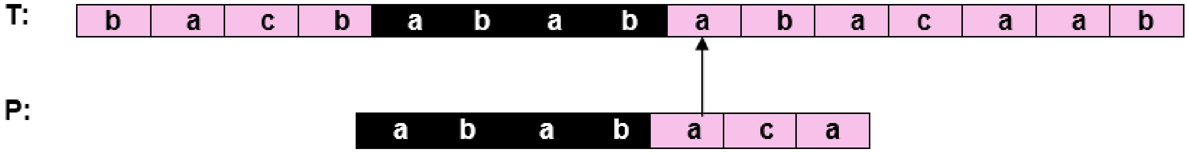
Step8: i = 8, q =3

Comparing P [4] with T [8]   P [4] matches with T [8]



Step9: i = 9, q = 4

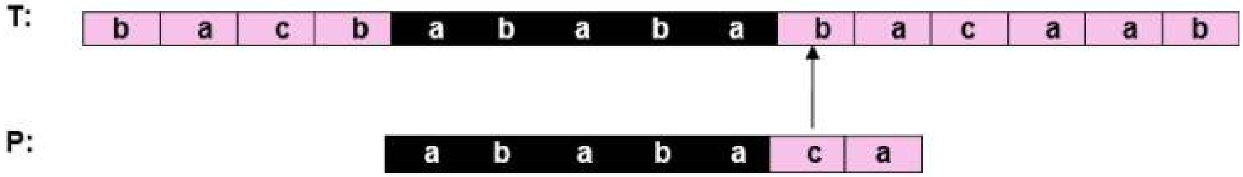
Comparing P [5] with T [9]   P [5] matches with T [9]



Step10: i = 10, q = 5

Comparing P [6] with T [10]

P [6] doesn't match with T [10]



T: b a c b a b a b a b a c a a b

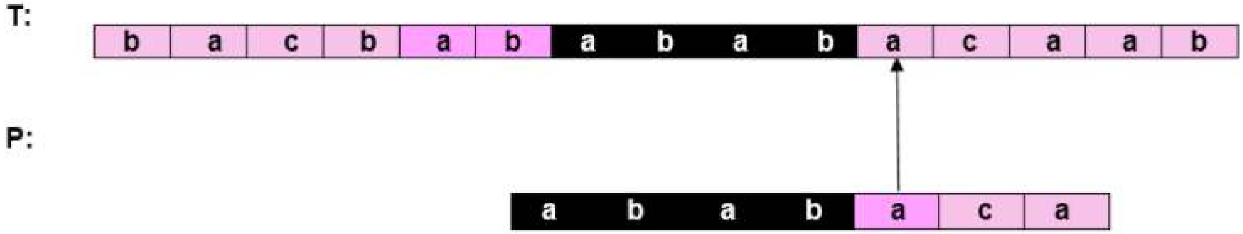
P: a b a b a c a

Backtracking on p, Comparing P [4] with T [10] because after mismatch q = π [5] = 3

Step11: i = 11, q =4

Comparing P [5] with T [11]

P [5] match with T [11]



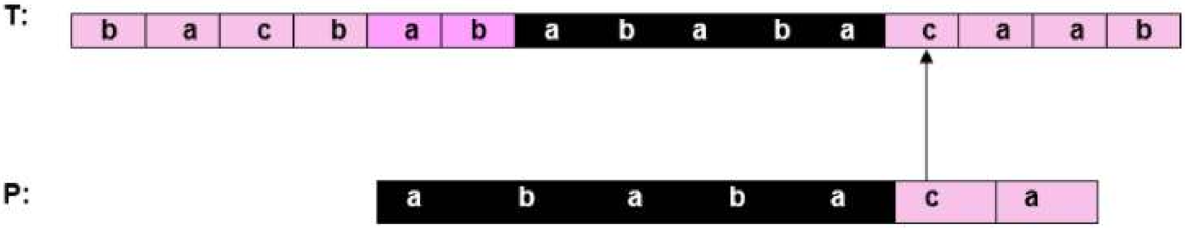
T: b a c b a b a b a b a c a a b

P: a b a b a c a

Step12: i = 12, q = 5

Comparing P [6] with T [12]

P [6] matches with T [12]



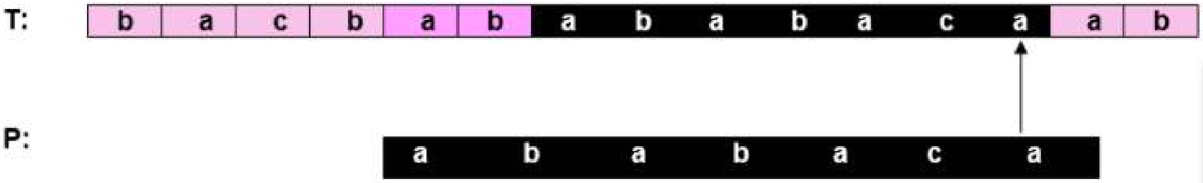
T: b a c b a b a b a b a c a a b

P: a b a b a c a

Step13: i = 3, q = 6

Comparing P [7] with T [13]

P [7] matches with T [13]

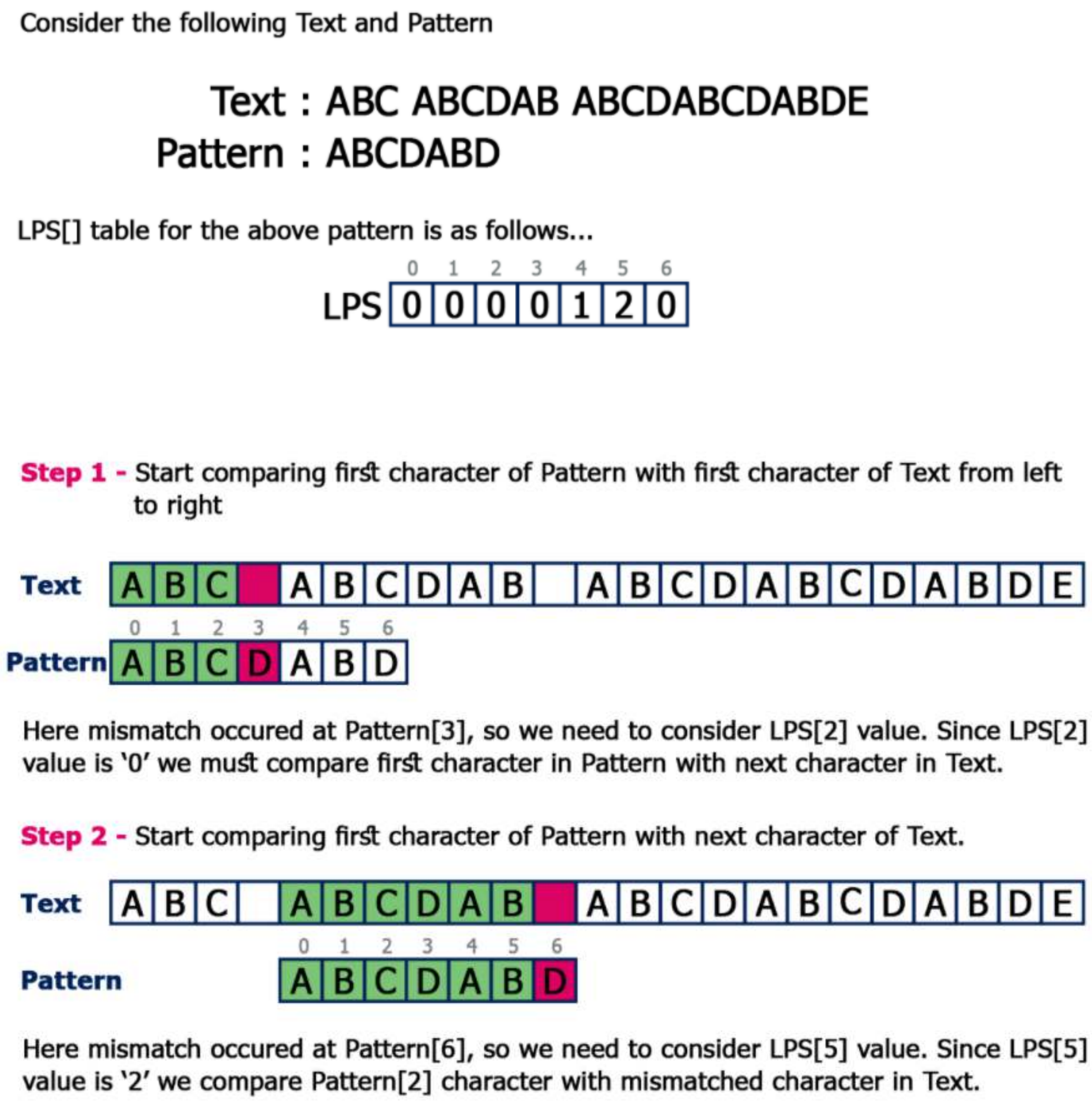


T: b a c b a b a b a b a c a a b

P: a b a b a c a

Pattern 'P' has been found to complexity occur in a string 'T.' The total number of shifts that took place for the match to be found is i-m = 13 - 7 = 6 shifts.

Example 2



Consider the following Text and Pattern

Text : ABC ABCDAB ABCDABCDABDE

Pattern : ABCDABD

LPS[] table for the above pattern is as follows...

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0

Step 1 - Start comparing first character of Pattern with first character of Text from left to right

Text	A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
	0	1	2	3	4	5	6																
Pattern	A	B	C	D	A	B	D																

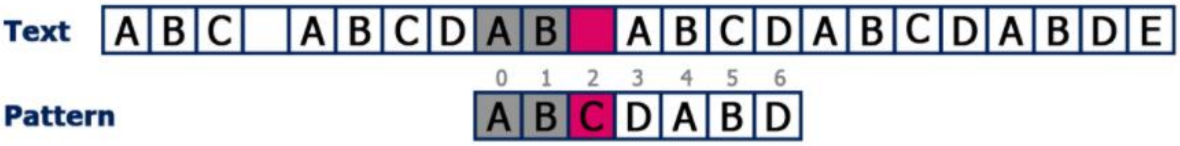
Here mismatch occured at Pattern[3], so we need to consider LPS[2] value. Since LPS[2] value is '0' we must compare first character in Pattern with next character in Text.

Step 2 - Start comparing first character of Pattern with next character of Text.

Text	A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
					0	1	2	3	4	5	6												
Pattern					A	B	C	D	A	B	D												

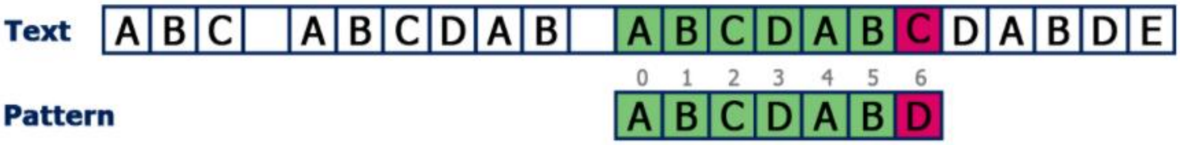
Here mismatch occured at Pattern[6], so we need to consider LPS[5] value. Since LPS[5] value is '2' we compare Pattern[2] character with mismatched character in Text.

Step 3 - Since LPS value is '2' no need to compare Pattern[0] & Pattern[1] values



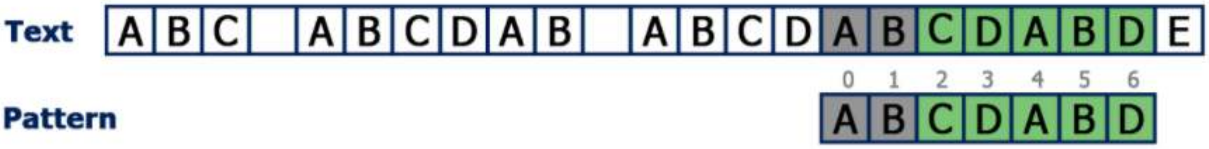
Here mismatch occured at Pattern[2], so we need to consider LPS[1] value. Since LPS[1] value is '0' we must compare first character in Pattern with next character in Text.

Step 4 - Compare Pattern[0] with next character in Text.



Here mismatch occured at Pattern[6], so we need to consider LPS[5] value. Since LPS[5] value is '2' we compare Pattern[2] character with mismatched character in Text.

Step 5 - Compare Pattern[2] with mismatched character in Text.



Here all the characters of Pattern matched with a substring in Text which is starting from index value 15. So we conclude that given Pattern found at index 15 in Text.

KMP ALGORITHM COMPLEXITY

O(m)- it is to compute to prefix function values
O(n)-it is to compare the pattern to the text
O(n+m)- Total time taken by KMP Algorithm.

Advantages

- The running time of KMP algorithm is O(n+m). which is very fast
- The algorithm never needs to move backwards in the input text T. It makes the algorithm good for processing very large files.

Disadvantages

- Does not work well as the size of the alphabet increase. By which more chances of mismatch occurs