Name: Vishal Shashikant Salvi.

UID: 2019230069

Class: SE Comps

Batch: C

Experiment No 8

Aim: To implement the KMP algorithm for string Matching

Theory:

The Knuth-Morris-Pratt (KMP)Algorithm

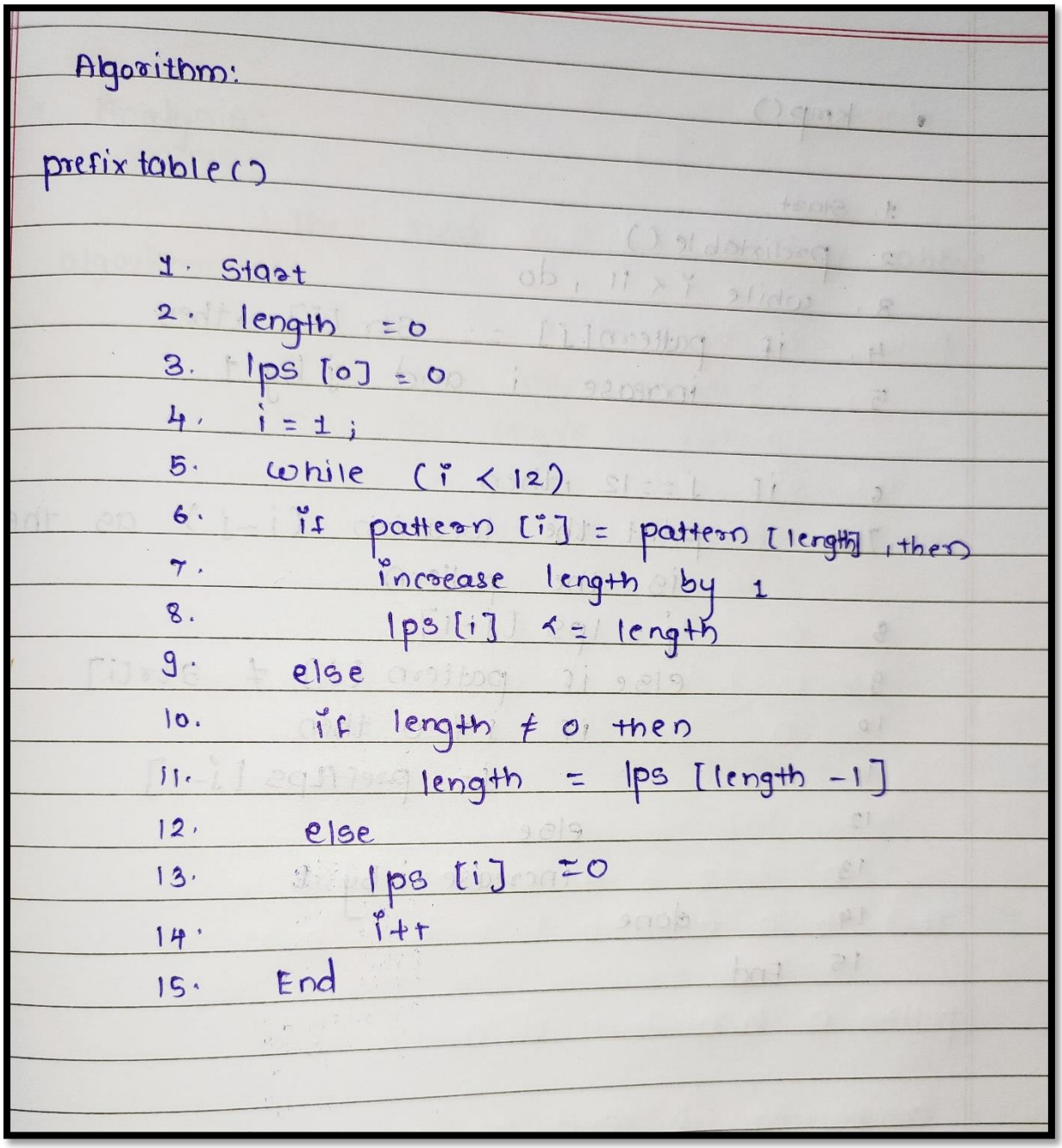
Knuth-Morris and Pratt introduce a linear time algorithm for the string-matching problem. A matching time of O (n) is achieved by avoiding comparison with an element of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs

Components of KMP Algorithm:

1. The Prefix Function (Π): The Prefix Function, Π for a pattern encapsulates knowledge about how the pattern matches against the shift of itself. This information can be used to avoid a useless shift of the pattern 'p.' In other words, this enables avoiding backtracking of the string 'S.'

2. The KMP Matcher: With string 'S,' pattern 'p' and prefix function 'Π' as inputs, find the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrences are found.

Algorithm:



Algorithm:

prefix table()

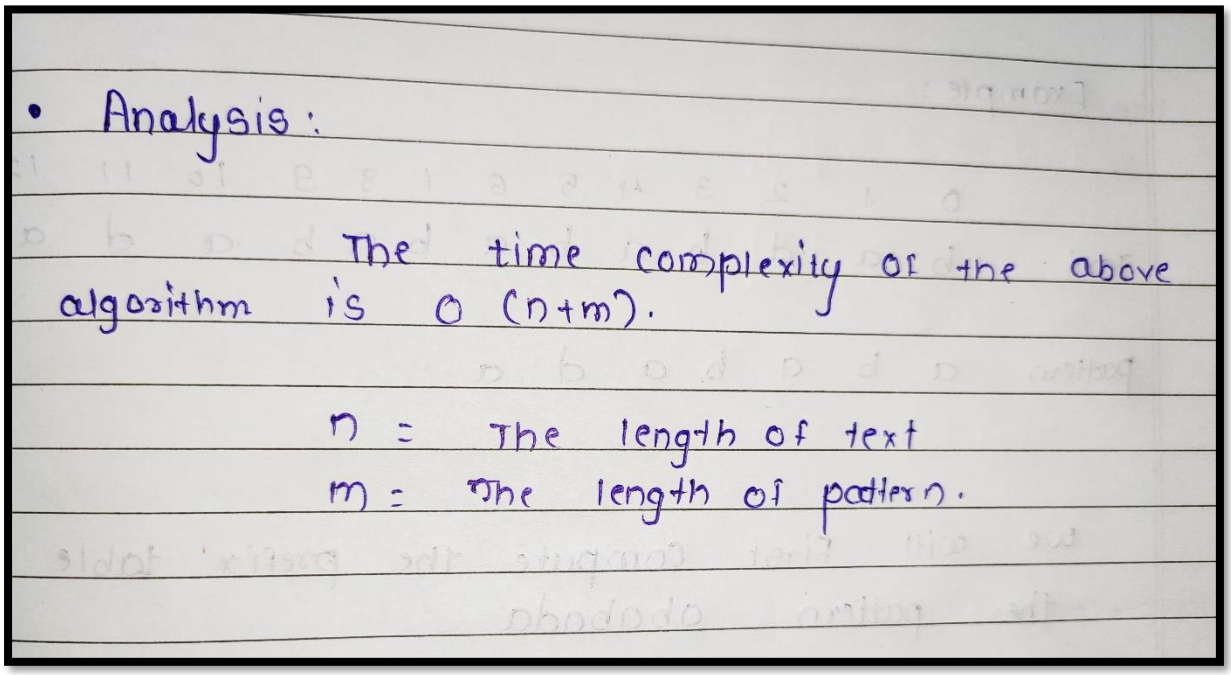
1. Start
2. length = 0
3. lps [0] = 0
4. i = 1;
5. while (i < 12)
6. if pattern [i] = pattern [length], then
7. increase length by 1
8. lps [i] = length
9. else
10. if length ≠ 0 then
11. length = lps [length - 1]
12. else
13. lps [i] = 0
14. i++
15. End

- kmp()

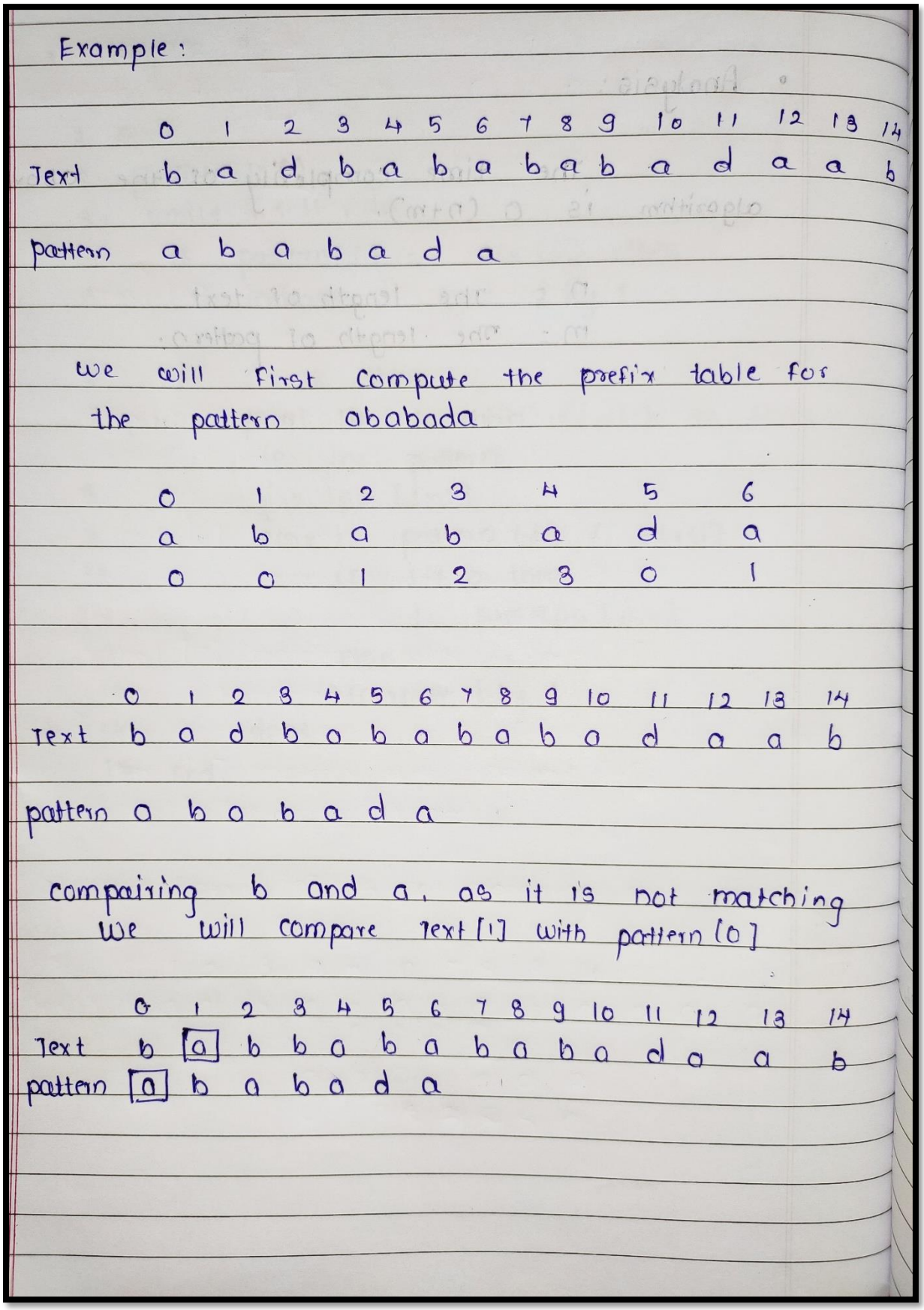
```
1. Start
2. prefixtable()
3. while i < 11, do
4.    if pattern[j] == Str[i], then
5.        increase i and j by 1

6.    if j==12, then
7.        print the location (i-j) as there
          is the pattern
8.        j = lps[j-1]
9.        else if pattern[j] ≠ Str[i]
10.           if j ≠ 0 then
11.               j = prefAlps[j-1]
12.           else
13.               increase i by 1
14.       done
15. End
```

Analysis:


• Analysis :
The time complexity of the above algorithm is O (n+m).
n = The length of text
m = The length of pattern.

Example:



Example :

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Text	b	a	d	b	a	b	a	b	a	b	a	d	a	a	b

pattern	a	b	a	b	a	d	a

we will first compute the prefix table for the pattern ababada

0	1	2	3	4	5	6
a	b	a	b	a	d	a
0	0	1	2	3	0	1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Text	b	a	d	b	a	b	a	b	a	b	a	d	a	a	b

pattern	a	b	a	b	a	d	a

comparing b and a, as it is not matching we will compare text[1] with pattern[0]

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Text	b	[a]	b	b	a	b	a	b	a	b	a	d	a	a	b
pattern	[a]	b	a	b	a	d	a								

As Text[1] is matching with Pattern[0], we will now compare Text[2] with pattern[1].

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Text	b	a	[d]	b	a	b	a	b	a	b	a	d	a	a	b
Pattern	a	[b]	a	b	a	d	a								

As text[2] is not matching with patter[1] we will backtrack on pattern and compare pattern[0] with Text[3].

Beacause we consult prefix-table[1] which is 0.

Hence pattern[0] is compared with Text[3].

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Text	b	a	d	[b]	a	b	a	b	a	b	a	d	a	a	b
				i											
pattern				[a]	b	a	b	a	d	a					
				j											

Again Text[3] is not matching with pattern[0]. we will then ask prefix-table[0] for the location of pattern.

As prefix table[0] is 0, we will compare pattern[0] with text[4].

```
        0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
text    b a d b a b a b a b a  a  a  b
          i
pattern [a] b a b a d a
          j
```

Text [4] matches with pattern [0].
Increment i and j

Text [5] matches with pattern [1]
Increment i and j

Text [6] matches with pattern [2]
Increment i and j

Text [7] matches with pattern [3]
Increment i and j

Text [8] matches with pattern [4]
Increment i and j

But Text [9] is not matching with pattern [5]

```
        0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
text    b a d b a b a b a b a  d  a  a  b
            ↑ ↑ ↑ ↑ ↑
          [a|b|a|b|a|d|a]
           0 1 2 3 4 5 6
```

Hence we must backtrack on pattern array. That means j will be positioning on location 4. Consult prefix-table [4] which denotes the value 3.

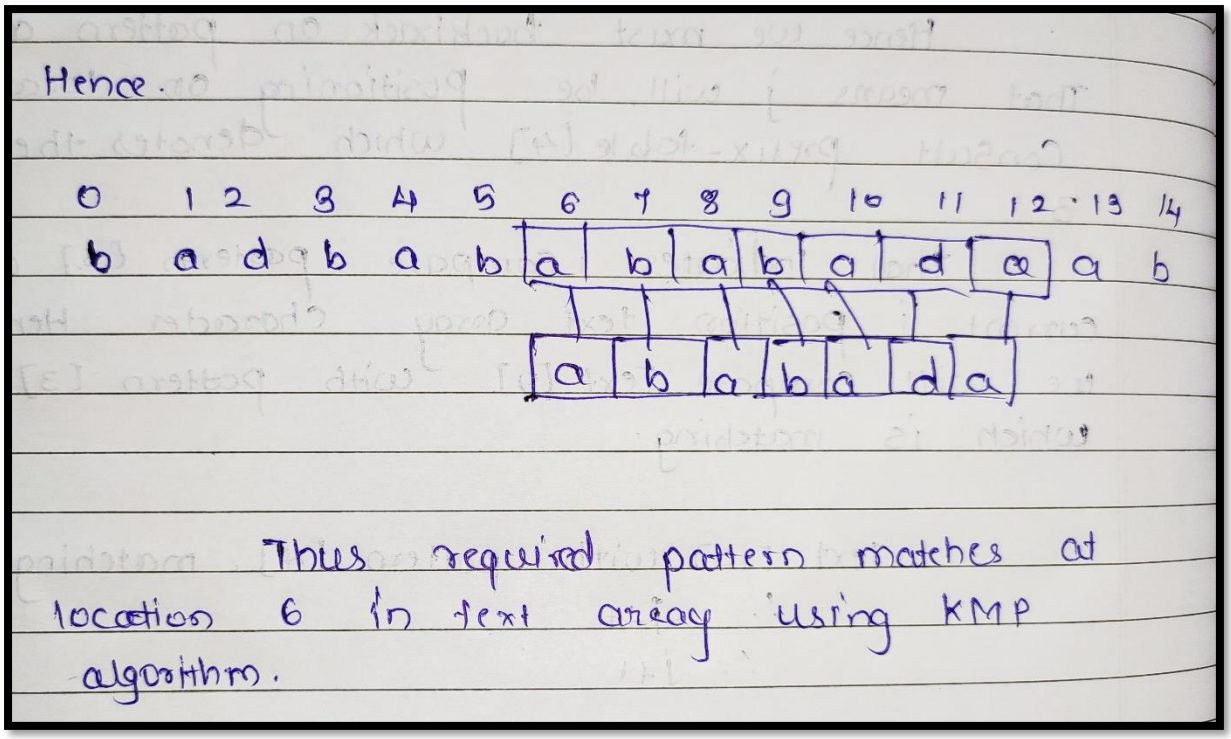
That indicates i compare pattern [3] with current i position text array character. Hence we will compare Text [9] with pattern [3], which is matching.

Text [10] with pattern [4], matching
∴ i++
∴ j++

Text [11] with pattern [5] matching
∴ i++
∴ j++

Text [12] with pattern [6] matching
∴ i++
∴ j++

Thus we have reached on the last characters of pattern, at the same time i is positioned at location 12 in the text array. The last characters of patteren is also matching with Text [12]. hence we can declare that a match of pattern is found in the text array at
i - length of pattern + 1.
i.e. 12 - 7 + 1
i.e. 6.



Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

char str[100],pattern[100];
int lps[100], locationArray[20]={[0 ... 19]=-1},ind;
int j=0,i=0,l1,l2;



void prefixtable()
{
        int len = 0, i;
        lps[0] = 0;
        i = 1;

        while(i < l2)
```

```
{
        if(pattern[i] == pattern[len])
        {
                len++;
                lps[i] = len;
                i++;
 }
 else
 {
                if( len != 0 )
                {
           len = lps[len-1];
                }
       else
    {
                lps[i] = 0;
                i++;
                }

         }
}
printf("\n*************************************************");
printf("\nLongest Prefix Sufix");
printf("\n");
printf("\t----------------------------------------\n");
printf("\tPattern:|");
for(i=0;i<l2;i++)
{
        printf(" %c |",pattern[i]);
}
```

```
    printf("\n");
    printf("\t-----------------------------------------\n");
    printf("\tLPS:    |");


    for(i=0;i<l2;i++)
    {
        printf(" %d |",lps[i]);
    }
    printf("\n");
    printf("\t-----------------------------------------\n");

}

void KMP()
{
        int j=0,i=0;
        prefixtable();
        printf("\n");
        while(i<l1)
        {

         if(pattern[j] == str[i])
         {
           j++;
           i++;
         }
         if(j==l2)

         {
```

```
            locationArray[ind] = i-j;
            ind++;
          j = lps[j-1];
    }

    else if(pattern[j] != str[i])
          {
          if(j != 0)
          {
            j = lps[j-1];

          }
          else
          {
                  i = i+1;
          }
      }
   }
}


int main()
{
          printf("\nEnter the String:");
          scanf("%s",str);
          printf("\nEnter the Pattern:");
          scanf("%s",pattern);

          l1=strlen(str);
          l2=strlen(pattern);
```

```
        KMP();
         if(locationArray[0]==-1)
   {

      printf("\n*****************************************************\n");
      printf("\tPattern not found.\n");
      printf("\n*****************************************************\n");

   }
  else{
     for(int i = 0; i<ind; i++) {
        printf("\n*****************************************************\n");
        printf( "\tPattern found at index: %d \n",locationArray[i] );
        printf("\n*****************************************************\n");

     }
   return 0;
  }

}
```

Output:

```
Enter the String:vishalsalvi

Enter the Pattern:salvi

**************************************************
Longest Prefix Sufix
        ----------------------------------------
        Pattern:| s | a | l | v | i |
        ----------------------------------------
        LPS:    | 0 | 0 | 0 | 0 | 0 |
        ----------------------------------------


**************************************************
        Pattern found at index: 6

**************************************************
```

```
Enter the String:vishalsalvi

Enter the Pattern:sssalvi

**************************************************
Longest Prefix Sufix
        ----------------------------------------
        Pattern:| s | s | s | a | l | v | i |
        ----------------------------------------
        LPS:    | 0 | 1 | 2 | 0 | 0 | 0 | 0 |
        ----------------------------------------


**************************************************
        Pattern not found.

**************************************************
```

Conclusion:

Thus, the basic idea behind this algorithm is to build a prefix array. This prefix array is built by using the prefix and suffix information of pattern. The overlapping prefix and suffix are used in LMP algorithm. The KMP algorithm achieves the efficiency of O(m+n) which is optimal in worst case.