Experiment No. 3

Problem Statement: Write a program to implement the Knuth Morris Pratt (KMP) algorithm.

Aim: Given a (short) pattern and a (long) text, both strings, determine whether the pattern appears somewhere in the text.

Theory:

The naive pattern searching algorithm doesn't work well in cases where we see many matching characters followed by a mismatching character.The KMP matching algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst-case complexity to O(n). 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 next window. We take advantage of this information to avoid matching the characters that we know will anyway match.

The Failure Function

The KMP algorithm preprocess the pattern P by computing a failure function f that indicates the largest possible shift s using previously performed comparisons. Specifically, the failure function f(j) is defined as the length of the longest prefix of P that is a suffix of P[i . . j].

KMP_FAILURE (P)

Input: Pattern with m characters

Output: Failure function f for P[i . . j]

```
    i = 1
    j = 0
    f(0) = 0
    while i < m do
        if P[j] = P[i]
            f(i) = j +1
            i = i +1
            j = j + 1
        elseif
            j = f(j - 1)
        else
            f(i) = 0
            i = i +1
```

Note that the failure function f for P, which maps j to the length of the longest prefix of P that is a suffix of P[1 . . j], encodes repeated substrings inside the pattern itself.

As an example, consider the pattern P = a b a c a b. The failure function, f(j), using above algorithm is

j	a	1	2	3	4	5
P[j]	a	b	a	c	a	b
f(j)	0	0	1	0	1	2

By observing the above mapping, we can see that the longest prefix of pattern, P, is "a b" which is also a suffix of pattern P.

Consider an attempt to match at position i, that is when the pattern P[0 … m -1] is aligned with text P[i … i + m -1].

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

When shifting, it is reasonable to expect that a prefix v of the pattern matches some suffix of the portion u of the text. In our example, u = a b a c a and v = a b a c a, therefore, 'a' a prefix of v matches with 'a' a suffix of u. Let l(j) be the length of the longest string P[0 … j-1] of pattern that matches with text followed by a character c different from P[j]. Then after a shift, the comparisons can resume between characters T[i + j] and P[l(j)], i.e., T(5) and P(1)

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

```
KMP_MATCH (T, P)
Input: Strings T[0 … n] and P[0 ... m]
Output: Starting index of substring of T matching P
    f = compute failure function of Pattern P
    i = 0
    j = 0
    while i < length[T] do
        if j == m-1 then
            return i- m+1// we have a match
            i = i +1
            j = j +1
        else
            if j > 0
                j = f(j -1)
            else
                i = i +1
```

NAME:HARSHBHAI SOLANKI
ROLL.N0:62
SE-4-D

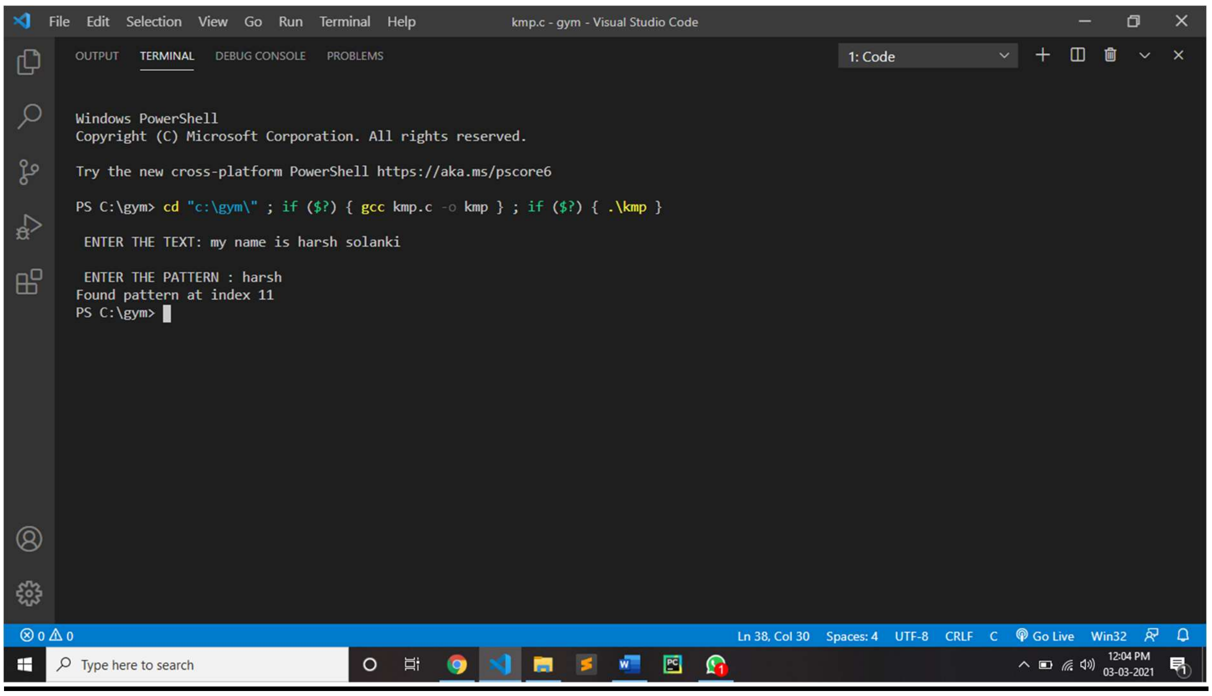
EXPERIMENT N0-3

AIM: Write a program to implement the KMP algorithm.

CODE:

```
#include <stdio.h>
#include <sptring.h>
char txt[100], pat[100];
int M, N, lps[100], j = 0, i = 0;
void computeArray()
{
   int len = 0, i;
   lps[0] = 0;
   i = 1;
   while (i < M)
   {
      if (pat[i] == pat[len])
      {
         len++;
         lps[i] = len;
         i++;
      }
      else
      {
         if (len != 0)
            len = lps[len - 1];
         else
         {
            lps[i] = 0;
            i++;
         }
      }
   }
}
void KMPSearch()
{
   int j = 0, i = 0;
   M = strlen(pat);
   N = strlen(txt);
   computeArray();
   while (i < N)
   {
      if (pat[j] == txt[i])
      {
         j++;
         i++;
      }
      if (j == M)
      {
         printf("Found pattern at index %d \n", i - j);
         j = lps[j - 1];
      }
      else if (pat[j] != txt[i])
      {
         if (j != 0)
            j = lps[j - 1];
            else
               i = i + 1;
         }
      }
}
int main()
{
    printf("\n ENTER THE TEXT: ");
    gets(txt);
    printf("\n ENTER THE PATTERN : ");
    gets(pat);
    KMPSearch();
    return 0;
}
```

<u>OUTPUT:</u>



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\gym> cd "c:\gym\" ; if ($?) { gcc kmp.c -o kmp } ; if ($?) { .\kmp }

 ENTER THE TEXT: my name is harsh solanki

 ENTER THE PATTERN : harsh
Found pattern at index 11
PS C:\gym>
```

File Edit Selection View Go Run Terminal Help
kmp.c - gym - Visual Studio Code
OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS
1: Code
Ln 38, Col 30 Spaces: 4 UTF-8 CRLF C Go Live Win32
Type here to search
12:04 PM
03-03-2021

CONCLUSION:

By performing Knuth Morris Pratt algorithm we can conclude that The time complexity of KMP algorithm apart from Suffix Array Calculation is O(m) and Suffix generation array takes O(n). So the total time complexity of KMP algorithm is O(m+n). Here m= size of array and n= size of pattern to be searched. Worst case for KMP algorithm is O(n). and the space complexity is O (m)