Practical-3.3

Student Name: Rajdeep Jaiswal **UID:** 20BCS2761
Branch: CSE **Section/Group: 902 WM B**
Semester: 05
Subject Name: Design & Analysis Algorithm **Subject Code:** 20CSP-312

1. **Aim:**

Code and analyze to find all occurrences of a pattern p in a given string s.

2. **Task to be done:**

To implement Knuth-Morris Pratt algorithm.

3. Algorithm: Unlike Naive algorithm, 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 next positions (or to know a number of characters to be skipped)?

1. We start comparison of pat[j] with j = 0 with characters of current window of text.
2. We keep matching characters txt[i] and pat[j] and keep incrementing i and j while pat[j] and txt[i] keep matching.
3. When we see a mismatch

4. We know that characters pat[0..j-1] match with txt[i-j…i-1] (Note that j starts with 0 and increment it only when there is a match).

5. We also know (from above definition) that lps[j-1] is count of characters of pat[0…j-1] that are both proper prefix and suffix.

6. From 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 above example to understand this.

Code:

```
 #include <bits/stdc++.h>

void computeLPSArray(char* pat, int M, int* lps);

// Prints occurrences of txt[] in pat[]
void KMPSearch(char* pat, char* txt)
{
   int M = strlen(pat);
   int N = strlen(txt);

   // create lps[] that will hold the longest prefix suffix
   // values for pattern
   int lps[M];

   // Preprocess the pattern (calculate lps[] array)
   computeLPSArray(pat, M, lps);

   int i = 0; // index for txt[]
```

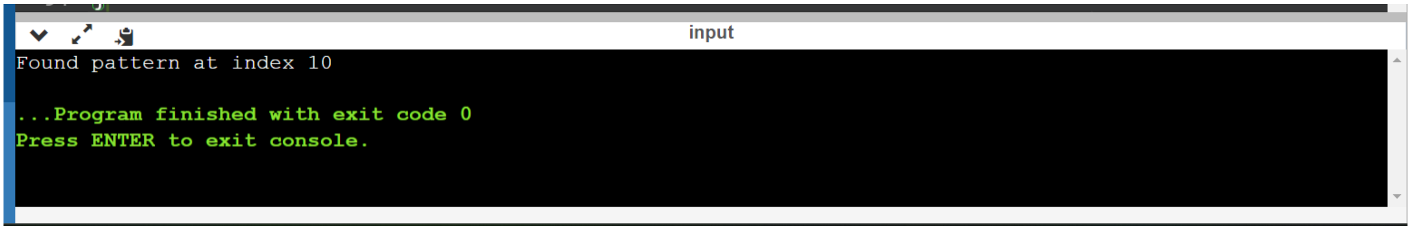
```
    int j = 0; // index for pat[]
    while ((N - i) >= (M - j)) {
        if (pat[j] == txt[i]) {
            j++;
            i++;
        }

        if (j == M) {
            printf("Found pattern at index %d ", i - j);
            j = lps[j - 1];
        }

        // mismatch after j matches
        else if (i < N && pat[j] != txt[i]) {
            // Do not match lps[0..lps[j-1]] characters,
            // they will match anyway
            if (j != 0)
                j = lps[j - 1];
            else
                i = i + 1;
        }
    }
}

// Fills lps[] for given pattern pat[0..M-1]
void computeLPSArray(char* pat, int M, int* lps)
{
    // length of the previous longest prefix suffix
```

```
int len = 0;

lps[0] = 0; // lps[0] is always 0

// the loop calculates lps[i] for i = 1 to M-1
int i = 1;
while (i < M) {
    if (pat[i] == pat[len]) {
        len++;
        lps[i] = len;
        i++;
    }
    else // (pat[i] != pat[len])
    {
        // This is tricky. Consider the example.
        // AAACAAAA and i = 7. The idea is similar
        // to search step.
        if (len != 0) {
            len = lps[len - 1];

            // Also, note that we do not increment
            // i here
        }
        else // if (len == 0)
        {
            lps[i] = 0;
            i++;
        }
        }
    }
}

// Driver program to test above function
int main()
{
    char txt[] = "ABABDABACDABABCABAB";
    char pat[] = "ABABCABAB";
    KMPSearch(pat, txt);
    return 0;
}
```

Complexity Analysis:

Time Complexity: O(n)

5. Result:



```
Found pattern at index 10

...Program finished with exit code 0
Press ENTER to exit console.
```

Learning outcomes (What I have learnt):

1. Learn about finding pattern in a string.
2. Learn about time complexity of program.

3. Learnt to implement Knuth-Morris Pratt algorithm.