DAY 17:

ASSIGNMENT 1:

Q) Implementing the KMP Algorithm

Code the Knuth-Morris-Pratt (KMP) algorithm in Java for pattern searching which pre-processes the pattern to reduce the number of comparisons. Explain how this pre-processing improves the search time compared to the naive approach.

```
package com.patternsearching;

public class KMPAlgorithm {
void KMPSearch(String pat, String txt) {
        int M = pat.length();
        int N = txt.length();

        // create lps[] that will hold the longest prefix suffix values for pattern
        int lps[] = new int[M];
        int j = 0; // index for pat[]

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

        int i = 0; // index for txt[]
        while (i < N) {
            if (pat.charAt(j) == txt.charAt(i)) {
                j++;
                i++;
            }
            if (j == M) {
                System.out.println("Found pattern at index " + (i - j));
                j = lps[j - 1];
            }
            // mismatch after j matches
            else if (i < N && pat.charAt(j) != txt.charAt(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;
            }
        }
    }

    void computeLPSArray(String pat, int M, int lps[]) {
        // length of the previous longest prefix suffix
        int len = 0;
```

```
        int i = 1;
        lps[0] = 0; // lps[0] is always 0

        // the loop calculates lps[i] for i = 1 to M-1
        while (i < M) {
            if (pat.charAt(i) == pat.charAt(len)) {
                len++;
                lps[i] = len;
                i++;
            } else {
                // (pat[i] != pat[len])
                if (len != 0) {
                    len = lps[len - 1];
                } else {
                    lps[i] = len;
                    i++;
                }
            }
        }
    }

    public static void main(String[] args) {
        KMPAlgorithm kmp = new KMPAlgorithm();
        String txt = "ABABDABACDABABCABAB";
        String pat = "ABABCABAB";
        kmp.KMPSearch(pat, txt);
    }
}
```

Explanation of the Code

1. computeLPSArray Function:

- This function computes the LPS array for the given pattern.

- It iterates through the pattern and fills the LPS array based on the previous longest prefix suffix.

2. KMPSearch Function:

- This function performs the actual search.

- It uses the LPS array to avoid redundant comparisons.

- When characters match, both text and pattern indices are incremented.

- If a mismatch occurs after some matches, the pattern index is updated using the LPS array.

3. Main Function:

- Provides a simple test case to demonstrate the KMP search.

Why Preprocessing Improves Search Time

The preprocessing step of computing the LPS array allows the KMP algorithm to skip over parts of the text that have already been matched. This significantly reduces the number of comparisons needed in the worst-case scenario. While the naive approach might take \(O(n \cdot m)\) time in the worst case, the KMP algorithm runs in \(O(n + m)\) time, making it much more efficient for larger texts and patterns.