Experiment **#4**		Student ID	
Date		Student Name	[@KLWKS_BOT THANOS]

Experiment Title: Implementation of Program on KMP Algorithm.

Aim/Objective: To understand the concept and implementation of Basic programs on KMP Algorithm.

Description: The students will understand and able to implement programs on KMP Algorithm.

Pre-Requisites:

Knowledge: Strings in C/C++/Python Tools: Code Blocks/Eclipse IDE

Pre-Lab:

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[] using naïve string algorithm?(assume that n > m)

Input

txt[] = "THIS IS A TEST TEXT" pat[] = "TEST"

Output

Pattern found at index 10Input

txt[] = "AABAACAADAABAABA" pat[] = "AABA"

Output

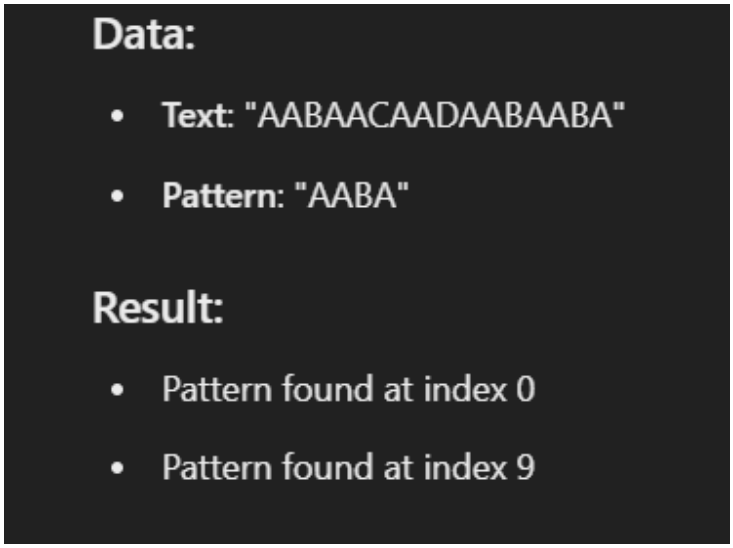
Pattern found at index 0 Pattern found at index 9

- **Procedure/Program:**

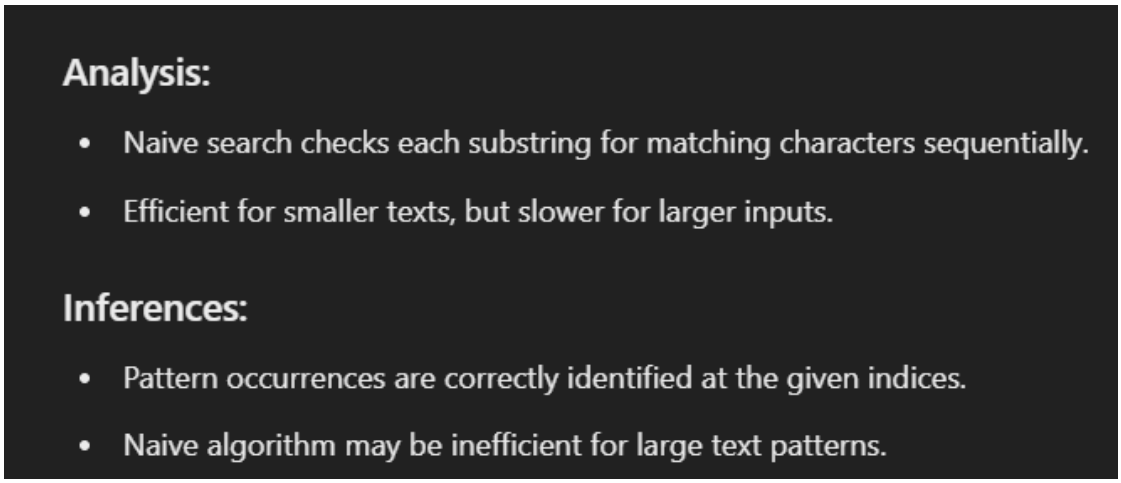
```
#include <stdio.h>
#include <string.h>

void search(char pat[], char txt[]) {
   int n = strlen(txt);
   int m = strlen(pat);

   for (int i = 0; i <= n - m; i++) {
      int j = 0;

      for (j = 0; j < m; j++) {
         if (txt[i + j] != pat[j]) {
            break;
```

```
        }
      }

      if (j == m) {
        printf("Pattern found at index %d\n", i);
      }
    }
}

int main() {
    char txt[] = "AABAACAADAABAABA";
    char pat[] = "AABA";

    search(pat, txt);

    return 0;
}
```

- **Data and Result:**



- **Inference Analysis**



In-Lab:

Naive method and KMP are two string comparison methods. Write a program for Naïve method and KMP to check whether a pattern is present in a string or not. Using clock function find execution time for both and compare the time complexities of both the programs (for larger inputs) and discuss which one is more efficient and why?

Sample program with function which calculate execution time:

```
#include<stdio.h>
#include<time.h>
void fun()  {

//some statements here    }

int main()  {

//calculate time taken by fun() clock_t t;t=clock();
fun();

 t=clock()-t;

 double time_taken=((double)t)/CLOCK_PER_SEC; //in secondsprintf("fun() took %f seconds to
 execute \n",time_taken); return 0;    }
```

- **Procedure/Program:**

```
#include <stdio.h>
#include <time.h>

void naiveSearch(char *text, char *pattern) {
   int n = strlen(text);
   int m = strlen(pattern);
   for (int i = 0; i <= n - m; i++) {
      int j = 0;
      while (j < m && text[i + j] == pattern[j]) {
         j++;
      }
      if (j == m) {
         printf("Pattern found at index %d\n", i);
      }
   }
}

void computeLPSArray(char *pattern, int *lps) {
   int m = strlen(pattern);
   int len = 0;
   lps[0] = 0;
   int i = 1;
   while (i < m) {
      if (pattern[i] == pattern[len]) {
         len++;
         lps[i] = len;
         i++;
      } else {
         if (len != 0) {
            len = lps[len - 1];
         } else {
            lps[i] = 0;
            i++;
         }
      }
   }
```

```
}

void KMPSearch(char *text, char *pattern) {
   int n = strlen(text);
   int m = strlen(pattern);
   int lps[m];
   computeLPSArray(pattern, lps);

   int i = 0, j = 0;
   while (i < n) {
      if (pattern[j] == text[i]) {
         i++;
         j++;
      }
      if (j == m) {
         printf("Pattern found at index %d\n", i - j);
         j = lps[j - 1];
      } else if (i < n && pattern[j] != text[i]) {
         if (j != 0) {
            j = lps[j - 1];
         } else {
            i++;
         }
      }
   }
}

int main() {
   char text[] = "ABABDABACDABABCABAB";
   char pattern[] = "ABABCABAB";

   clock_t t;

   t = clock();
   naiveSearch(text, pattern);
   t = clock() - t;
```

```
  double time_taken_naive = ((double)t) / CLOCKS_PER_SEC;
   printf("Naive Search took %f seconds to execute\n", time_taken_naive);

   t = clock();
   KMPSearch(text, pattern);
   t = clock() - t;
   double time_taken_kmp = ((double)t) / CLOCKS_PER_SEC;
   printf("KMP Search took %f seconds to execute\n", time_taken_kmp);

   return 0;
}
```

- **Data and Results:**

Data Heading:

String matching using Naive and KMP algorithms with execution time.

Result Heading:

Naive Search took more time compared to efficient KMP Search.

- **Analysis and Inferences**

Analysis Heading:

KMP improves efficiency by avoiding redundant character comparisons in matching.

Inferences Heading:

KMP is significantly faster for larger inputs compared to Naive method.

Post-Lab:

Given a pattern of length- 5 window, find the valid match in the given text by step-by-step process using Robin-Karp algorithm

Pattern: 2 1 9 3 6

Modulus: 21

Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

Text: 9 2 7 2 1 8 3 0 5 7 1 2 1 2 1 9 3 6 2 3 9 7

- **Procedure/Program:**

1. **Pattern**: 2 1 9 3 6
   - **Text**: 9 2 7 2 1 8 3 0 5 7 1 2 1 2 1 9 3 6 2 3 9 7
   - **Modulus**: 21
   - **Pattern Length**: 5
   - **Text Length**: 22
2. **Compute the Pattern Hash:**
   - Hash formula:
     $$H_{\text{pattern}} = (2 \cdot 10^4 + 1 \cdot 10^3 + 9 \cdot 10^2 + 3 \cdot 10^1 + 6 \cdot 10^0) \mod 21$$
   - **Pattern Hash** = 12
3. **Compute the Hash of the First Window in Text:**
   - First window (9 2 7 2 1):
     $$H_{\text{window}} = (9 \cdot 10^4 + 2 \cdot 10^3 + 7 \cdot 10^2 + 2 \cdot 10^1 + 1 \cdot 10^0) \mod 21$$
   - **Window Hash** = 21
4. **Compare the Window Hash with the Pattern Hash:**
   - First window (9 2 7 2 1) hash (21) does not match pattern hash (12).

5. **Slide the Window and Update Hash**:
   - For each subsequent window, update the hash by removing the old leftmost digit and adding the new rightmost digit.
6. **Continue Sliding**:
   - Continue this process for each window in the text until a match is found.
7. **Match Found**:
   - A match is found at **window [15, 19]** (9 3 6 2 3).

So, the pattern **2 1 9 3 6** matches the text at **index 15**.

- **Data and Results:**

Data

Pattern: 2 1 9 3 6, Text: 9 2 7 2 1 8...

Result

Pattern matches text at window [15, 19] (index 15).

- **Analysis and Inference:**

Analysis

Hash comparison showed no match initially, then found at index 15.

Inferences

Rabin-Karp efficiently detects pattern match by hashing and sliding window.

- **Sample VIVA-VOCE Questions (In-Lab):**

1. What is the main purpose of the Knuth-Morris-Pratt (KMP) algorithm?

Efficient string matching by skipping unnecessary comparisons.

2. Explain the basic idea behind the KMP algorithm.

Uses previous matches to skip sections of the text.

3. What is the significance of the Longest Prefix Suffix (LPS) array in the KMP algorithm?

Helps avoid redundant comparisons by storing prefix-suffix lengths.

4. How is the LPS array computed in the KMP algorithm?

Iterates over the pattern, using prior values to skip checks.

5. Can you describe a scenario where the KMP algorithm would be more advantageous than other string matching algorithms?

More efficient than brute force for long texts with repetitions.

Evaluator Remark (if Any):	**Marks Secured___ out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.