Experiment **#14**		Student ID	
Date		Student Name	[@KLWKS_BOT] THANOS

Experiment Title: To implement programs on String Matching Algorithms.

Aim/Objective: To understand the concept and implementation of programs on String Matching Algorithms.

Description: The students will understand the programs on Knuth-Morris-Pratt Algorithm, Rabin-Karp Algorithm, Boyer-Moore Algorithm, applying them to solve real-world problems.

Pre-Requisites:

Knowledge: Knuth-Morris-Pratt Algorithm, Rabin-Karp Algorithm, Boyer-Moore Algorithm.

Tools: Code Blocks/Eclipse IDE.

Pre-Lab:

Your test string S will have the following requirements:

- S must be of length 6
- First character: 1, 2 or 3
- Second character: 1, 2 or 0
- Third character: x, s or 0
- Fourth character: 3, 0, A or a
- Fifth character: x, s or u
- Sixth character: . or ,

Sample Input:

Test_strings = ["12x3x.", "31sA,", "20u0s.", "10xAa."]

Sample Output:

"12x3x.": Valid → True

"31sA,": Valid → True

"20u0s.": Valid → True

"10xAa.": Invalid → False

- **Procedure/Program:**

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

bool is_valid(const char *s) {
   return (s[0] == '1' || s[0] == '2' || s[0] == '3') &&
         (s[1] == '1' || s[1] == '2' || s[1] == '0') &&
         (s[2] == 'x' || s[2] == 's' || s[2] == '0') &&
         (s[3] == '3' || s[3] == '0' || s[3] == 'A' || s[3] == 'a') &&
         (s[4] == 'x' || s[4] == 's' || s[4] == 'u') &&
         (s[5] == '.' || s[5] == ',') &&
         s[6] == '\0';
}

int main() {
   const char *test_strings[] = {"12x3x.", "31sA,", "20u0s.", "10xAa."};
   for (int i = 0; i < 4; i++) {
      printf("\"%s\": Valid → %s\n", test_strings[i], is_valid(test_strings[i]) ? "True" : "False");
   }
   return 0;
}
```

- **Data and Results:**

Data

Test strings checked for validity based on given character constraints.

Result

Some strings met conditions, while others failed the validation rules.

- **Analysis and Inferences:**

Analysis

Validation ensures input format correctness for structured data processing.

Inferences

Strict pattern matching helps identify errors in predefined string formats.

2. A pangram is a string that contains every letter of the alphabet. Given a sentence determine whether it is a pangram in the English alphabet. Ignore case. Return either pangram or not pangram as appropriate.

Example-1:

Input :

S1= "the quick brown fox jumps over the lazy dog"

Output :

Pangram

Example-2:

Input :

S2 = "We promptly judged antique ivory buckles for the prize"

Output :

Not Pangram

- **Procedure/Program**:

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>



int isPangram(const char *str) {

   int alphabet[26] = {0};

   int index;



   for (int i = 0; str[i]; i++) {
```

```
      if (isalpha(str[i])) {
         index = tolower(str[i]) - 'a';
         alphabet[index] = 1;
      }
   }

   for (int i = 0; i < 26; i++) {
      if (alphabet[i] == 0) {
         return 0;
      }
   }
   return 1;
}
int main() {
   const char *S1 = "the quick brown fox jumps over the lazy dog";
   const char *S2 = "We promptly judged antique ivory buckles for the prize";
   printf("%s\n", isPangram(S1) ? "Pangram" : "Not Pangram");
   printf("%s\n", isPangram(S2) ? "Pangram" : "Not Pangram");

   return 0;
}
```

- **Data and Results:**

Data

Checks if a sentence contains all English alphabet letters.

Result

Both given sentences are identified as valid pangrams correctly.

- **Analysis and Inferences:**

Analysis

Each letter is tracked in an array to verify completeness.

Inferences

Pangrams ensure all alphabet letters appear, useful in testing.

In-Lab:

1. You are given a text TTT and a pattern PPP. Your task is to implement the Rabin-Karp algorithm to find the first occurrence of PPP in TTT. If PPP exists in TTT, return the starting index of the match (0-based indexing). Otherwise, return -1.

Example 1:

Input :

T = "ababcabcabababd"

P = "ababd"

Output:

10

Example 2:

Input :

T = "hello"

P = "world"

Output:

-1

- **Procedure/Program**:

```
#include <stdio.h>

#include <string.h>



#define d 256

#define q 101



int rabinKarp(char *T, char *P) {

   int M = strlen(P);

   int N = strlen(T);

   int i, j;

   int p = 0;

   int t = 0;

   int h = 1;



   for (i = 0; i < M - 1; i++)

      h = (h * d) % q;



   for (i = 0; i < M; i++) {

      p = (d * p + P[i]) % q;
```

```
        t = (d * t + T[i]) % q;
    }

    for (i = 0; i <= N - M; i++) {
        if (p == t) {
            for (j = 0; j < M; j++) {
                if (T[i + j] != P[j])
                    break;
            }
            if (j == M)
                return i;
        }

        if (i < N - M) {
            t = (d * (t - T[i] * h) + T[i + M]) % q;
            if (t < 0)
                t = t + q;
        }
    }
    return -1;
}
```

```
int main() {
    char T[] = "ababcabcabababd";
    char P[] = "ababd";
    int result = rabinKarp(T, P);
    printf("%d\n", result);

    char T2[] = "hello";
    char P2[] = "world";
    result = rabinKarp(T2, P2);
    printf("%d\n", result);

    return 0;
}
```

- **Data and Results:**

Data

Rabin-Karp algorithm searches for a pattern in given text.

Result

Pattern found at index 10 in first case, not found second.

- **Analysis and Inferences:**

Analysis

Uses hashing for efficient substring search in large texts.

Inferences

Hash collisions may occur, requiring additional character comparisons.

2. Design a program to find the length of the longest common substring between two input strings by using KMP Algorithm (Knuth-Morris-Pratt).

Example 1:

Input:

str1 = "zohoinnovations"

str2 = "innov"

Output:

Longest Common Substring Length: 5

- **Procedure/Program**:

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

void computeLPSArray(char* str, int* lps, int len) {
    int length = 0;
    lps[0] = 0;
    int i = 1;

    while (i < len) {
        if (str[i] == str[length]) {
            length++;
            lps[i] = length;
            i++;
        } else {
            if (length != 0) {
                length = lps[length - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}

int longestCommonSubstring(char* str1, char* str2) {
```

```
    int len1 = strlen(str1);
    int len2 = strlen(str2);
    int maxLength = 0;

    for (int i = 0; i < len1; i++) {
        int j = 0;
        while (j < len2 && str1[i + j] == str2[j]) {
            j++;
            if (j > maxLength) {
                maxLength = j;
            }
        }
    }
    return maxLength;
}

int main() {
    char str1[] = "zohoinnovations";
    char str2[] = "innov";

    int length = longestCommonSubstring(str1, str2);
    printf("Longest Common Substring Length: %d\n", length);

    return 0;
}
```

- **Data and Results:**

Data

Two strings are compared to find the longest common substring.

Result

The longest common substring length between them is five.

- **Analysis and Inferences:**

Analysis

A brute-force approach checks all substrings for maximum match.

Inferences

Efficient substring search is crucial in text processing applications.

Post-Lab:

2. Using the Boyer-Moore algorithm, write a program to count how many times a pattern appears in a text string.

Example:

Input:

Text = "INFO Starting process\nERROR Invalid configuration\nINFO Retrying process\nERROR Connection failed\nINFO Process complete\nERROR Disk full"

pattern = "ERROR"

Output :

The pattern 'ERROR' appears 3 times in the text.

- **Procedure/Program**:

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

#define NO_OF_CHARS 256
#define max(a, b) ((a) > (b) ? (a) : (b))

void badCharHeuristic(char *str, int size, int badchar[NO_OF_CHARS]) {
  for (int i = 0; i < NO_OF_CHARS; i++)
    badchar[i] = -1;

  for (int i = 0; i < size; i++)
    badchar[(int)str[i]] = i;
}
```

```
int search(char *text, char *pattern) {
   int m = strlen(pattern);
   int n = strlen(text);
   int badchar[NO_OF_CHARS];

   badCharHeuristic(pattern, m, badchar);

   int s = 0;
   int count = 0;
   while (s <= n - m) {
      int j = m - 1;

      while (j >= 0 && pattern[j] == text[s + j])
         j--;

      if (j < 0) {
         count++;
         s += (s + m < n) ? m - badchar[text[s + m]] : 1;
      } else {
         s += max(1, j - badchar[text[s + j]]);
      }
   }
   return count;
}
```

```
int main() {
   char text[] = "INFO Starting process\nERROR Invalid configuration\nINFO Retrying process\nERROR Connection failed\nINFO Process complete\nERROR Disk full";
   char pattern[] = "ERROR";

   int result = search(text, pattern);
   printf("The pattern '%s' appears %d times in the text.\n", pattern, result);

   return 0;
}
```

- **Data and Results:**

Data

Text contains multiple log messages, including "INFO" and "ERROR" entries.

Result

The pattern "ERROR" appears three times in the given text.

- **Analysis and Inferences :**

Analysis

Boyer-Moore algorithm efficiently finds occurrences in large text.

Inferences

Log analysis helps identify frequent errors for debugging purposes.

3. Find DNA Pattern Occurrences :You are given a DNA sequence (text) and a DNA pattern (substring) consisting of the characters A, C, G, and T. Write a program to find all starting indices of the occurrences of the DNA pattern in the DNA sequence.

Example 1:

Input:

DNA Sequence: "ACGTACGTGACG"

DNA Pattern: "ACG"

Output:

Pattern found at indices: [0, 4, 9]

- **Procedure/Program**:

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

void findPatternOccurrences(const char* text, const char* pattern) {
  int textLength = strlen(text);
  int patternLength = strlen(pattern);
  int found = 0;

  for (int i = 0; i <= textLength - patternLength; i++) {
    if (strncmp(&text[i], pattern, patternLength) == 0) {
      printf("%d ", i);
      found = 1;
    }
  }

  if (!found) {
    printf("Pattern not found");
  }
}
```

```
int main() {
   const char* dnaSequence = "ACGTACGTGACG";
   const char* dnaPattern = "ACG";

   printf("Pattern found at indices: [");
   findPatternOccurrences(dnaSequence, dnaPattern);
   printf("]\n");

   return 0;
}
```

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

1. What is a string matching algorithm?

- It is an algorithm used to find occurrences of a pattern in a given text.

2. What are the primary objectives of string matching algorithms?

- To efficiently locate patterns in text while minimizing time complexity.

3. What is the difference between a string and a pattern in the context of string matching?

- A **string** is the main text, while a **pattern** is the substring we are searching for.

4. Explain the Naive String Matching Algorithm. Why is it considered inefficient for large inputs?

- It checks the pattern at every position in the text, leading to **O(n × m)** time complexity, making it slow.

5. What are the common applications of string matching algorithms in real-world scenarios?

- Used in search engines, plagiarism detection, DNA sequencing, and network security.

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.