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:**

```
public class Main {
   public static boolean isValid(String s) {
      return (s.charAt(0) == '1' || s.charAt(0) == '2' || s.charAt(0) == '3') &&
            (s.charAt(1) == '1' || s.charAt(1) == '2' || s.charAt(1) == '0') &&
            (s.charAt(2) == 'x' || s.charAt(2) == 's' || s.charAt(2) == '0') &&
            (s.charAt(3) == '3' || s.charAt(3) == '0' || s.charAt(3) == 'A' || s.charAt(3) == 'a') &&
            (s.charAt(4) == 'x' || s.charAt(4) == 's' || s.charAt(4) == 'u') &&
            (s.charAt(5) == '.' || s.charAt(5) == ',') &&
            s.length() == 6;
   }

   public static void main(String[] args) {
      String[] testStrings = {"12x3x.", "31sA,", "20u0s.", "10xAa."};
      for (String testString : testStrings) {
         System.out.printf("\"%s\": Valid → %s%n", testString, isValid(testString) ? "True" :
"False");
      }
   }
}
```

- **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**:

```
public class Main {

public static boolean isPangram(String str) {

  boolean[] alphabet = new boolean[26];

  int index;


  for (int i = 0; i < str.length(); i++) {

    if (Character.isLetter(str.charAt(i))) {

      index = Character.toLowerCase(str.charAt(i)) - 'a';

      alphabet[index] = true;
```

```
        }
    }

    for (boolean b : alphabet) {
        if (!b) {
            return false;
        }
    }
    return true;
}

public static void main(String[] args) {
    String S1 = "the quick brown fox jumps over the lazy dog";
    String S2 = "We promptly judged antique ivory buckles for the prize";
    System.out.println(isPangram(S1) ? "Pangram" : "Not Pangram");
    System.out.println(isPangram(S2) ? "Pangram" : "Not Pangram");
}
}
```

- **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**:

```
 public class Main {
static final int d = 256;
static final int q = 101;

public static int rabinKarp(String T, String P) {
   int M = P.length();
   int N = T.length();
   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.charAt(i)) % q;
      t = (d * t + T.charAt(i)) % q;
   }

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

      if (i < N - M) {
         t = (d * (t - T.charAt(i) * h) + T.charAt(i + M)) % q;
         if (t < 0)
            t = t + q;
```

```
        }
      }
      return -1;
    }

    public static void main(String[] args) {
      String T = "ababcabcabababd";
      String P = "ababd";
      int result = rabinKarp(T, P);
      System.out.println(result);

      String T2 = "hello";
      String P2 = "world";
      result = rabinKarp(T2, P2);
      System.out.println(result);
    }
}
```

- **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**:

```
public class Main {
  public static int longestCommonSubstring(String str1, String str2) {
    int len1 = str1.length();
    int len2 = str2.length();
    int maxLength = 0;

    for (int i = 0; i < len1; i++) {
      int j = 0;
      while (j < len2 && (i + j) < len1 && str1.charAt(i + j) == str2.charAt(j)) {
        j++;
        if (j > maxLength) {
          maxLength = j;
        }
      }
    }
```

```
        return maxLength;
    }

    public static void main(String[] args) {
        String str1 = "zohoinnovations";
        String str2 = "innov";

        int length = longestCommonSubstring(str1, str2);
        System.out.println("Longest Common Substring Length: " + length);
    }
}
```

- **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**:

```
public class Main {
    static final int NO_OF_CHARS = 256;

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

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

    static int search(String text, String pattern) {
        int m = pattern.length();
```

```
    int n = text.length();
    int[] badchar = new int[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.charAt(j) == text.charAt(s + j))
        j--;

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

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

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

- **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**:

```
public class Main {
  public static void findPatternOccurrences(String text, String pattern) {
    int textLength = text.length();
    int patternLength = pattern.length();
    boolean found = false;

    for (int i = 0; i <= textLength - patternLength; i++) {
      if (text.substring(i, i + patternLength).equals(pattern)) {
        System.out.print(i + " ");
        found = true;
      }
    }

    if (!found) {
      System.out.print("Pattern not found");
    }
  }

  public static void main(String[] args) {
    String dnaSequence = "ACGTACGTGACG";
    String dnaPattern = "ACG";
```

```
        System.out.print("Pattern found at indices: [");
        findPatternOccurrences(dnaSequence, dnaPattern);
        System.out.println("]");
    }
}
```

- **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 **$O(n \times m)$** 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.