Data Structure and Operations: Day-11 (String, Naive Pattern, KMP, Rabin-Karp, Boyer-Moore):

Task 1: String Operations

Write a method that takes two strings, concatenates them, reverses the result, and then extracts the middle substring of the given length. Ensure your method handles edge cases, such as an empty string or a substring length larger than the concatenated string.

```java
package wipro.com.assignment08;
public class StringManipulation {

    // concatenate two strings,reverse the result.
        public static String concatenateReverseExtract(String str1,String str2, int
      length) {
        // Concatenate the two strings
        String concatenated = str1 + str2;

        // Handle edge case where either string is empty
        if (concatenated.isEmpty()) {
            return "";
        }

        // Reverse the concatenated string
        String reversed = new StringBuilder(concatenated).reverse().toString();


        if (length > reversed.length()) {
            length = reversed.length();
        }

        // Find the starting index for the middle substring
        int start = (reversed.length() - length) / 2;

        // Extract and return the middle substring
        return reversed.substring(start, start + length);
    }

    public static void main(String[] args)  {
        // Test cases
        System.out.println(concatenateReverseExtract("hello", "world", 5)); //
Output: "dlrow"
        System.out.println(concatenateReverseExtract("java", "program", 3)); //
Output: "arg"
        System.out.println(concatenateReverseExtract("", "empty", 4));      //
Output: "ytpm"
        System.out.println(concatenateReverseExtract("short", "", 2));      //
Output: "tr"
```

```
        System.out.println(concatenateReverseExtract("edge", "case", 20));  // 
Output: "esacedge"
    }
}
```

Output:

```
rowol
orp
ytpm
ro
esacegde
```

Task 2: Naive Pattern Search

Implement the naive pattern searching algorithm to find all occurrences of a pattern within a given text string. Count the number of comparisons made during the search to evaluate the efficiency of the algorithm.

```
package wipro.com.assignment08;
      public class NaivePatternSearch {

          // Function to perform naive pattern search
          public static void naivePatternSearch(String text, String pattern) {
              int textLength = text.length();
              int patternLength = pattern.length();
              int comparisonCount = 0;

              // Loop through the text to find the pattern
              for (int i = 0; i <= textLength - patternLength; i++) {
                  int j;

                  // For current index i, check for pattern match
                  for (j = 0; j < patternLength; j++) {
                      comparisonCount++;
                      if (text.charAt(i + j) != pattern.charAt(j)) {
                          break;
                      }
                  }

                  // If pattern is found
                  if (j == patternLength) {
                      System.out.println("Pattern found at index " + i);
                  }
              }

              // Print the total number of comparisons
              System.out.println("Total number of comparisons: " + comparisonCount);
          }
```

```
        public static void main(String[] args) {
            String text = "ABABDABATTAAEEVVCDABABCABAB";
            String pattern = "TTAAEEVV";

            System.out.println("Text: " + text);
            System.out.println("Pattern: " + pattern);

            naivePatternSearch(text, pattern);
        }
    }
```

Output:

```
Text: ABABDABATTAAEEVVCDABABCABAB
Pattern: TTAAEEVV
Pattern found at index 8
Total number of comparisons: 28
```

Task 3: Implementing the KMP Algorithm

Code the Knuth-Morris-Pratt (KMP) algorithm in C# 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 wipro.com.assignment08;

public class KMPAlgorithm {

    // Preprocess the pattern to create the LPS array
    private static int[] computeLPSArray(String pattern) {
        int length = pattern.length();
        int[] lps = new int[length];
        int len = 0;  // length of the previous longest prefix suffix
        int i = 1;

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

        // Compute the LPS array
        while (i < length) {
            if (pattern.charAt(i) == pattern.charAt(len)) {
                len++;
                lps[i] = len;
                i++;
            } else {
                if (len != 0) {
                    len = lps[len - 1];
                } else {
                    lps[i] = 0;
                    i++;
                }
            }
        }
        return lps;
    }

    // KMP search algorithm
    public static void KMPSearch(String text, String pattern) {
        int m = text.length();
        int n = pattern.length();

        // Preprocess the pattern to create the LPS array
        int[] lps = computeLPSArray(pattern);

        int i = 0;  // index for text
        int j = 0;  // index for pattern

        while (i < m) {
            if (pattern.charAt(j) == text.charAt(i)) {
                i++;
                j++;
            }

            if (j == n) {
                System.out.println("Pattern found at index " + (i - j));
                j = lps[j - 1];
            } else if (i < m && pattern.charAt(j) != text.charAt(i)) {
                if (j != 0) {
                    j = lps[j - 1];
                } else {
                    i++;
                }
            }
        }
    }

    public static void main(String[] args)
   {
        String text = "ABABDAVISHALABCABAB";
        String pattern = "VISHAL";

        System.out.println("Text: " + text);
        System.out.println("Pattern: " + pattern);

        KMPSearch(text, pattern);
    }
}
```

Output:

```
Text: ABABDAVISHALABCABAB
Pattern: VISHAL
Pattern found at index 6
```

Task 4: Rabin-Karp Substring Search

Implement the Rabin-Karp algorithm for substring search using a rolling hash. Discuss the impact of hash collisions on the algorithm's performance and how to handle them.

```
package wipro.com.assignment08;
import java.util.*;

public class RabinKarp  {
    private final static int d = 256; // number of characters in the input alphabet
    private final static int q = 101; // a prime number

    // Rabin-Karp algorithm for pattern searching
    public static void search(String pattern, String text) {
        int m = pattern.length();
        int n = text.length();
        int i, j;
        int p = 0; // hash value for pattern
        int t = 0; // hash value for text
        int h = 1;

        // The value of h would be "pow(d, m-1) % q"
        for (i = 0; i < m - 1; i++)
            h = (h * d) % q;

        // Calculate the hash value of the pattern and first window of the text
        for (i = 0; i < m; i++) {
            p = (d * p + pattern.charAt(i)) % q;
            t = (d * t + text.charAt(i)) % q;
        }

        // Slide the pattern over the text one by one
        for (i = 0; i <= n - m; i++) {

            // Check the hash values of current window of text and pattern
            if (p == t) {
                // Check for characters one by one
                for (j = 0; j < m; j++) {
                    if (text.charAt(i + j) != pattern.charAt(j))
                        break;
                }

                // If p == t and pattern[0...m-1] = text[i, i+1, ...i+m-1]
                if (j == m)
                    System.out.println("Pattern found at index " + i);
            }

            // Calculate hash value for the next window of text: Remove leading
digit, add trailing digit
            if (i < n - m) {
                t = (d * (t - text.charAt(i) * h) + text.charAt(i + m)) % q;
                // We might get negative value of t, converting it to positive
                if (t < 0)
                    t = (t + q);
            }
        }
    }

    public static void main(String[] args)   {
        String text = "I AM SOFTWARE ENGINEER";
        String pattern = "SOFTWARE";

        System.out.println("Text: " + text);
        System.out.println("Pattern: " + pattern);

        search(pattern, text);
    }
}
```

Output:

```
Text: I AM SOFTWARE ENGINEER
Pattern: SOFTWARE
Pattern found at index 5
```

Task 5: Boyer-Moore Algorithm Application

Use the Boyer-Moore algorithm to write a function that finds the last occurrence of a substring in a given string and returns its index. Explain why this algorithm can outperform others in certain scenarios.

```
package wipro.com.assignment08;

public class BoyerMoore
{

    // The Boyer-Moore algorithm for pattern searching
    public static int boyerMooreSearch(String text, String pattern) {
        int[] badChar = preprocessBadChar(pattern);
        int[] goodSuffix = preprocessGoodSuffix(pattern);

        int n = text.length();
        int m = pattern.length();
        int i = 0;

        // Start from the end of the text and move backwards
        while (i <= n - m) {
            int j = m - 1;

            // Keep reducing index j of pattern while characters of pattern and text
are matching
```

```
            while (j >= 0 && pattern.charAt(j) == text.charAt(i + j))
                j--;

            // If the pattern is present at the current shift, return the index
            if (j < 0) {
                return i;
            } else {
                // Shift the pattern so that the bad character in text aligns with
the last occurrence of it in the pattern
                i += Math.max(goodSuffix[j], j - badChar[text.charAt(i + j)]);
            }
        }
        return -1;
    }

    // Preprocess the pattern to create the Bad Character Heuristic array
    private static int[] preprocessBadChar(String pattern) {
        int[] badChar = new int[256];
        int m = pattern.length();

        // Initialize all occurrences as -1
        for (int i = 0; i < 256; i++) {
            badChar[i] = -1;
        }

        // Fill the actual value of the last occurrence
        for (int i = 0; i < m; i++) {
            badChar[pattern.charAt(i)] = i;
        }

        return badChar;
    }

    // Preprocess the pattern to create the Good Suffix Heuristic array
    private static int[] preprocessGoodSuffix(String pattern)
   {
        int m = pattern.length();
        int[] goodSuffix = new int[m];

        int[] suffix = new int[m];
        suffix[m - 1] = m;
        int g = m - 1, f = m - 1;

        for (int i = m - 2; i >= 0; i--) {
            if (i > g && suffix[i + m - 1 - f] < i - g) {
                suffix[i] = suffix[i + m - 1 - f];
            } else {
                if (i < g) {
                    g = i;
                }
                f = i;
                while (g >= 0 && pattern.charAt(g) == pattern.charAt(g + m - 1 - f))
{
                    g--;
                }
                suffix[i] = f - g;
            }
        }

        for (int i = 0; i < m; i++) {
            goodSuffix[i] = m;
        }
        int j = 0;
        for (int i = m - 1; i >= 0; i--) {
            if (suffix[i] == i + 1) {
                for (; j < m - 1 - i; j++) {
                    if (goodSuffix[j] == m) {
                        goodSuffix[j] = m - 1 - i;
                    }
                }
            }
        }
        for (int i = 0; i <= m - 2; i++) {
            goodSuffix[m - 1 - suffix[i]] = m - 1 - i;
        }

        return goodSuffix;
    }

    public static void main(String[] args) {
        String text = "HERE IS A SIMPLE EXAMPLE";
        String pattern = "EXAMPLE";

        System.out.println("Text: " + text);
        System.out.println("Pattern: " + pattern);

        int index = boyerMooreSearch(text, pattern);
        System.out.println("Last occurrence of the pattern is at index: " + index);
    }
}
```

Output:

```
Text: HER IS A SIMPLE EXAMPLE
Pattern: EXAMPLE
Last occurrence of the pattern is at index: 16
```