Assignment-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.

```
package com.wipro.day11;

public class StringOperations {

    public static String concatenateReverseAndExtractMiddle(String str1, String str2, int length) {
        String concatenated = str1 + str2;

        String reversed = new StringBuilder(concatenated).reverse().toString();

        if (length > reversed.length()) {
            return "Substring length is larger than the concatenated string length.";
        }
        if (length == 0 || reversed.length() == 0) {
            return "";
        }

        int start = (reversed.length() - length) / 2;
        return reversed.substring(start, start + length);
    }

    public static void main(String[] args) {
        // Test cases
        System.out.println(concatenateReverseAndExtractMiddle("hello", "world", 5));
        System.out.println(concatenateReverseAndExtractMiddle("java", "program", 3));
        System.out.println(concatenateReverseAndExtractMiddle("abc", "def", 7));
        System.out.println(concatenateReverseAndExtractMiddle("", "def", 2));
        System.out.println(concatenateReverseAndExtractMiddle("abc", "", 2));
    }
}
```

OUTPUT:

```
<terminated> StringOperations [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe  (03-Jun-2024, 8:25:34 pm – 8:25:34 pm) [pid: 5712]
rowol
orp
Substring length is larger than the concatenated string length.
fe
cb
```

Assignment-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 com.wipro.day11;

public class NaivePatternSearch {
    public static int naivePatternSearch(String text, String pattern) {
        int textLength = text.length();
        int patternLength = pattern.length();
        int comparisonCount = 0;

        for (int i = 0; i <= textLength - patternLength; i++) {
            int j;

            for (j = 0; j < patternLength; j++) {
                comparisonCount++;
                if (text.charAt(i + j) != pattern.charAt(j)) {
                    break;
                }
            }
            if (j == patternLength) {
                System.out.println("Pattern found at index " + i);
            }
        }
        return comparisonCount;
    }
    public static void main(String[] args) {
        String text = "AABAACAADAABAABA";
        String pattern = "AABA";

        int comparisons = naivePatternSearch(text, pattern);
        System.out.println("Total comparisons made: " + comparisons);
    }
}
```

OUTPUT:

```
<terminated> NaivePatternSearch [Java Application] C:\Program Files\Java
Pattern found at index 0
Pattern found at index 9
Pattern found at index 12
Total comparisons made: 30
```

Assignment-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 com.wipro.day11;

public class KMPAlgorithm {
    public static void KMPSearch(String pattern,
String text) {
        int M = pattern.length();
        int N = text.length();
        int[] lps = new int[M];
        computeLPSArray(pattern, M, lps);

        int i = 0;
        int j = 0;
        while (i < N) {
            if (pattern.charAt(j) == text.charAt(i))
{
                j++;
                i++;
            }

            if (j == M) {
                System.out.println("Found pattern at
index " + (i - j));
                j = lps[j - 1];
            } else if (i < N && pattern.charAt(j) !=
text.charAt(i)) {
                if (j != 0)
                    j = lps[j - 1];
                else
                    i = i + 1;
            }
        }
    }
```

```
    private static void computeLPSArray(String
pattern, int M, int[] lps) {
        int length = 0;
        int i = 1;
        lps[0] = 0;

        while (i < M) {
            if (pattern.charAt(i) ==
pattern.charAt(length)) {
                length++;
                lps[i] = length;
                i++;
            } else {

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

    public static void main(String[] args) {
        String text = "ABABDABACDABABCABAB";
        String pattern = "ABABCABAB";
        KMPSearch(pattern, text);
    }
}
```

Assignment-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.

Solution:

```java
package com.wipro.day11;

public class RabinKarpAlgorithm {
    public final static int d = 256;

    static void search(String pattern, String text,
int q) {
        int M = pattern.length();
        int N = text.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 + pattern.charAt(i)) % q;
            t = (d * t + text.charAt(i)) % q;
        }

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

                if (j == M)
                    System.out.println("Pattern
found at index " + i);
            }
            if (i < N - M) {
                t = (d * (t - text.charAt(i) * h) +
text.charAt(i + 1)) % q;

                if (t < 0)
                    t = (t + q);
            }
        }
    }
```

```
    public static void main(String[] args) {
        String text = "i am subha";
        String pattern = "subha";
        int q = 101;
        search(pattern, text, q);
    }
}
```

Assignment-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.

Solution:

```
package com.wipro.day11;

public class BoyerMooreAlgorithm {
    private final int ALPHABET_SIZE = 256;

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

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

    void search(String text, String pattern) {
        char[] txt = text.toCharArray();
        char[] pat = pattern.toCharArray();
        int m = pat.length;
        int n = txt.length;

        int[] badchar = new int[ALPHABET_SIZE];

        badCharHeuristic(pat, m, badchar);
```

```
        int s = 0; // s is shift of the pattern with
respect to text
        while (s <= (n - m)) {
            int j = m - 1;

            while (j >= 0 && pat[j] == txt[s + j])
                j--;

            if (j < 0) {
                System.out.println("Pattern found at
index " + s);

                s += (s + m < n) ? m - badchar[txt[s
+ m]] : 1;

            } else {
                s += Math.max(1, j - badchar[txt[s +
j]]);
            }
        }
    }

    public static void main(String[] args) {
        BoyerMooreAlgorithm bm = new
BoyerMooreAlgorithm();
        String text = "ABAAABCD";
        String pattern = "ABC";
        bm.search(text, pattern);
    }
}
```

OUTPUT: