ASSIGNMENT: Day 11

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.

ANS:

```
public class StringOperations {


  public static String performStringOperations(String str1, String str2, int length) {


    String concatenated = str1 + str2;

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


    int n = reversed.length();

    if (length > n) {

      length = n;

    }

    int start = (n - length) / 2;


    return reversed.substring(start, start + length);

  }


  public static void main(String[] args) {


    System.out.println(performStringOperations("hello", "world", 5));  // Output: "dlrow"

    System.out.println(performStringOperations("abc", "def", 4));     // Output: "fedc"
```

```
        System.out.println(performStringOperations("", "xyz", 2));        // Output: "yz"
        System.out.println(performStringOperations("abc", "def", 10));    // Output: "fedcba"
    }
}
//code_by_RUBY
```

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.

ANS:

```
public class NaivePatternSearch {

    public static void naivePatternSearch(String text, String pattern) {
        int textLength = text.length();
        int patternLength = pattern.length();
        int comparisons = 0;

        for (int i = 0; i <= textLength - patternLength; i++) {
            int j;
            for (j = 0; j < patternLength; j++) {
                comparisons++;
                if (text.charAt(i + j) != pattern.charAt(j)) {
                    break;
                }
            }
            if (j == patternLength) {
                System.out.println("Pattern found at index " + i);
            }
        }
        System.out.println("Total comparisons made: " + comparisons);
    }

    public static void main(String[] args) {

        String text = "AABAACAADAABAAABAA";
        String pattern = "AABA";
        naivePatternSearch(text, pattern);

    }
}
//code_by_RUBY
```

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.

ANS:

```
public class KMP {

    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] = length;

                i++;

            }

        }

    }

}


public static void main(String[] args) {

    String text = "ABABDABACDABABCABAB";

    String pattern = "ABABCABAB";

    KMPSearch(pattern, text);

}
```

```
}

//code_by_RUBY
```

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.

ANS:

```
public class RabinKarp {


    public final static int d = 256;

    public final static int q = 101;


    public static void search(String pattern, String text) {

        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 + M)) % q;

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

    public static void main(String[] args) {
        String text = "GEEKS FOR GEEKS";
        String pattern = "GEEK";
        search(pattern, text);
    }

}

//code_by_RUBY
```

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.

ANS:

```
public class BoyerMoore {


    private final int NO_OF_CHARS = 256;


    private void badCharHeuristic(char[] 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[i]] = i;

    }


    public int search(char[] text, char[] pattern) {

        int m = pattern.length;

        int n = text.length;
    int[] badChar = new int[NO_OF_CHARS];

    badCharHeuristic(pattern, m, badChar);


    int s = 0;

    int lastOccurrence = -1;


    while (s <= (n - m)) {

        int j = m - 1;


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

            j--;


        if (j < 0) {

            lastOccurrence = s;

            s += (s + m < n) ? m - badChar[text[s + m]] : 1;

        } else

            s += Math.max(1, j - badChar[text[s + j]]);

    }


    return lastOccurrence;

}


public static void main(String[] args) {

    BoyerMoore bm = new BoyerMoore();

    String text = "ABAAABCD";
```

```
        String pattern = "ABC";

        int result = bm.search(text.toCharArray(), pattern.toCharArray());



        if (result == -1)

            System.out.println("Pattern not found in the text.");

        else

            System.out.println("Last occurrence of the pattern found at index " + result);

    }

}

//code_by_RUBY
```