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.

```
public static String extractMiddleSubstring(String str1, String str2, int length) {
  if (str1 == null || str2 == null || length <= 0) {
    return "";
  }

  String combinedStr = str1 + str2;
  int combinedLength = combinedStr.length();

  if (combinedLength < length) {
    return "";
  }

  int startIndex = combinedLength / 2 - length / 2;

  return new StringBuilder(combinedStr).reverse().substring(startIndex, startIndex + length).toString();
}
```

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.

```
public class NaivePatternSearch {

  public static void search(String text, String pattern) {
    int comparisons = 0;

    if (text == null || pattern == null || pattern.isEmpty()) {
      return;
    }

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

    for (int i = 0; i <= n - m; i++) {
      int j = 0;
      while (j < m && text.charAt(i + j) == pattern.charAt(j)) {
        comparisons++;
        j++;
      }

      if (j == m) {
        System.out.println("Pattern found at index: " + i);
      }
    }

    System.out.println("Total comparisons: " + comparisons);
  }
}
```

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.

```
public class KMPSearch {


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


    int[] lps = computeLPSArray(pattern);


    int i = 0;
    int j = 0;


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


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


        if (j > 0) {
          j = lps[j - 1];
        } else {
          i++;
```

```
        }
      }
    }
  }

  private static int[] computeLPSArray(String pattern) {
    int[] lps = new int[pattern.length()];
    int len = 0;
    int i = 1;

    while (i < pattern.length()) {
      if (pattern.charAt(i) == pattern.charAt(len)) {
        lps[i] = len + 1;
        len++;
        i++;
      } else {
        if (len > 0) {
          len = lps[len - 1];
        } else {
          lps[i] = 0;
          i++;
        }
      }
    }
    return lps;
  }
}
```

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.

```
public class RabinKarpSearch {

  public static void search(String text, String pattern, int d, int q) {
    int n = text.length();
    int m = pattern.length();
    int p = 0;
    int t = 0;
    int h = 1;

    for (int i = 0; i < m - 1; i++) {
      h = (h * d) % q;
    }

    for (int i = 0; i < m; i++) {
      p = (p * d + (int) pattern.charAt(i)) % q;
      t = (t * d + (int) text.charAt(i)) % q;
    }

    int i = 0;
    while (i <= n - m) {
      if (p == t) {
        boolean match = true;
        for (int j = 0; j < m; j++) {
          if (text.charAt(i + j) != pattern.charAt(j)) {
            match = false;
            break;
```

```
        }
      }
      if (match) {
        System.out.println("Pattern found at index: " + i);
      }
    }

    if (i < n - m) {
      t = (d * (t - (int) text.charAt(i) * h) + (int) text.charAt(i + m)) % q;
      if (t < 0) {
        t = (t + q);
      }
    }
    i++;
  }
 }
}
```

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.

```
public class BoyerMoore {

  public static int searchLastOccurrence(String text, String pattern) {
    if (text == null || pattern == null || pattern.isEmpty()) {
      return -1;
    }
```

```
        int n = text.length();
        int m = pattern.length();

        int[] badCharTable = computeBadCharacterTable(pattern);

        int i = n - 1;
        while (i >= 0) {
            int j = m - 1;

            while (j >= 0 && (text.charAt(i) != pattern.charAt(j) || !isPrefixSuffixMatch(text, i, pattern, j))) {
                int shift = badCharTable[text.charAt(i)];
                if (shift == -1) {
                    shift = m;
                }
                i -= shift;
                break;
            }
            if (j == -1) {
                return i;
            }
            i--;
        }
        return -1;
    }

    private static int[] computeBadCharacterTable(String pattern) {
        int[] badCharTable = new int[256];

        Arrays.fill(badCharTable, -1);
```

```
        for (int i = 0; i < pattern.length(); i++) {
            badCharTable[pattern.charAt(i)] = i;
        }

        return badCharTable;
    }

    private static boolean isPrefixSuffixMatch(String text, int textIndex, String pattern, int patternIndex) {
        int k = patternIndex;
        while (k >= 0 && text.charAt(textIndex--) == pattern.charAt(k--)) ;
        return k == -1;
    }
}
```