Day 11

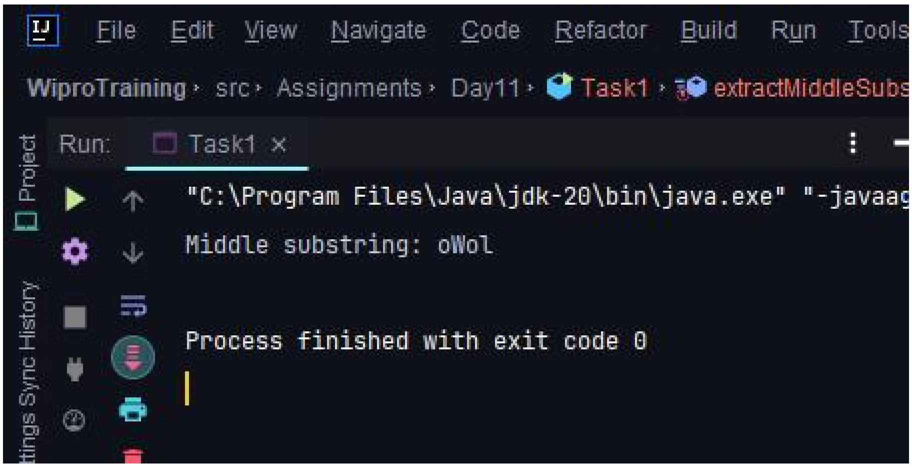
Task 1: String Operations
Write a method that takes two strings, concatenates them, reverses the result, and then extracts the middle substring of the given length. Ensure your method handles edge cases, such as an empty string or a substring length larger than the concatenated string.

Program:

```
package Assignments.Day11;

public class Task1 {

   public static String extractMiddleSubstring(String str1, String str2, int length)
{
      // Concatenate the two input strings
      String concatenated = str1 + str2;

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

      // Calculate the middle index
      int middleIndex = reversed.length() / 2;

      // Handle edge case: empty string or invalid length
      if (reversed.isEmpty() || length <= 0 || length > reversed.length()) {
         return "Invalid input or empty result.";
      }

      // Extract the middle substring
      int startIndex = middleIndex - length / 2;
      int endIndex = startIndex + length;
      return reversed.substring(startIndex, endIndex);
   }

   public static void main(String[] args) {
      String str1 = "Hello";
      String str2 = "World";
      int desiredLength = 4;
      // HelloWorld 10
      //dlr oWol leH
```

```
        String result = extractMiddleSubstring(str1, str2, desiredLength);
        System.out.println("Middle substring: " + result);
    }
}
```

Output:



Task 2: Naive Pattern Search

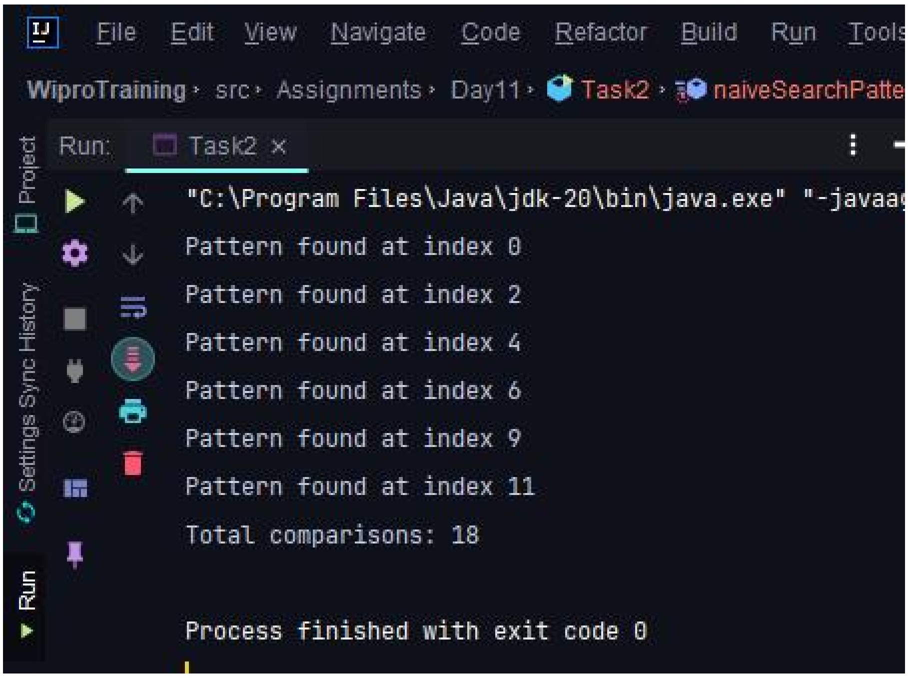
Implement the naive pattern searching algorithm to find all occurrences of a pattern within a given text string. Count the number of comparisons made during the search to evaluate the efficiency of the algorithm.

Program:

```
package Assignments.Day11;

public class Task2 {

    public static void naiveSearchPattern(String text, String pattern) {
        int n = text.length();
        int m = pattern.length();
        int comparisons = 0;

        for (int i = 0; i <= n - m; i++) {
            int j;
            for (j = 0; j < m; j++) {
                comparisons++; // Count each character comparison
                if (text.charAt(i + j) != pattern.charAt(j)) {
                    break;
                }
            }
```

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

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

    public static void main(String[] args) {
        String text = "ABABABABCABAB"; // 0 AB 2 AB 4 AB 6 AB C 9 AB 11 AB
        String pattern = "AB";

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

Output:



Task 3: Implementing the KMP Algorithm

Code the Knuth-Morris-Pratt (KMP) algorithm in java 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.

Program:

```
package Assignments.Day11;

public class Task3 {

    // Compute LPS array
    private static int[] computeLPS(String pattern) {
        int m = pattern.length();
        int[] lps = new int[m];
        int len = 0; // Length of the previous longest prefix suffix

        for (int i = 1; i < m; ) {
            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 pattern search
    public static void searchPatternKMP(String text, String pattern) {
        int n = text.length();
        int m = pattern.length();
        int[] lps = computeLPS(pattern);

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

        while (i < n) {
            if (pattern.charAt(j) == text.charAt(i)) {
```

```
                i++;
                j++;
            }

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

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

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

Output:

Knuth-Morris-Pratt (KMP) Algorithm:

1. **Preprocessing (Compute LPS Array):**

   - The Longest Prefix Suffix (LPS) array stores the length of the longest proper prefix that is also a suffix for each prefix of the pattern.
   - It helps us skip unnecessary character comparisons when a mismatch occurs.
   - We build the LPS array in a single pass over the pattern.

2. **Pattern Matching:**

   We compare characters of the text and pattern:

   - If characters match, we move both pointers forward.
   - If characters don’t match:
     - We use the LPS array to determine how many characters in the pattern can be skipped.
     - We adjust the pattern pointer accordingly.

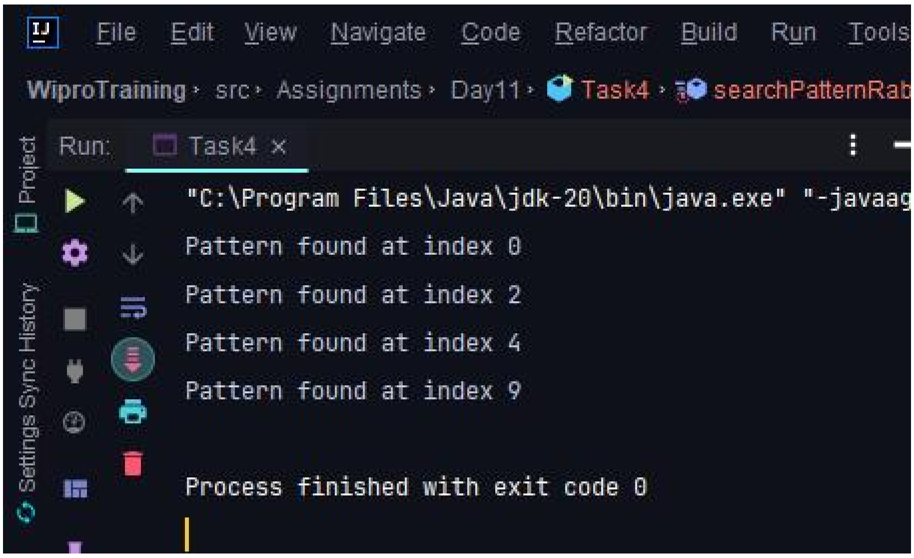
Task 4: Rabin-Karp Substring Search
Implement the Rabin-Karp algorithm for substring search using a rolling hash. Discuss the impact of hash collisions on the algorithm's performance and how to handle them.

Program:

```
package Assignments.Day11;

public class Task4{

   private static final int PRIME = 101; // Prime number for hash computation

   public static void searchPatternRabinKarpAlgorithm(String text, String pattern) {
       int n = text.length();
       int m = pattern.length();
       long patternHash = computeHash(pattern);
       long textHash = computeHash(text.substring(0, m));
```

```
        for (int i = 0; i <= n - m; i++) {
            if (patternHash == textHash) {
                if (text.substring(i, i + m).equals(pattern)) {
                    System.out.println("Pattern found at index " + i);
                }
            }
            if (i < n - m) {
                textHash = recomputeHash(textHash, text.charAt(i), text.charAt(i +
m), m);
            }
        }
    }

    private static long computeHash(String str) {
        long hash = 0;
        for (char ch : str.toCharArray()) {
            hash = (hash * PRIME + ch) % Integer.MAX_VALUE;
        }
        return hash;
    }

    private static long recomputeHash(long oldHash, char oldChar, char
newChar, int m) {
        long newHash = (oldHash - oldChar * pow(PRIME, m - 1)) %
Integer.MAX_VALUE;
        newHash = (newHash * PRIME + newChar) % Integer.MAX_VALUE;
        return (newHash + Integer.MAX_VALUE) % Integer.MAX_VALUE;
    }

    private static long pow(int base, int exp) {
        long result = 1;
        for (int i = 0; i < exp; i++) {
            result = (result * base) % Integer.MAX_VALUE;
        }
        return result;
    }

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

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

Output:



Impact of Hash Collisions:

- Hash collisions can lead to false positives (i.e., matching hash values but different substrings).
- Handling collisions with character-by-character comparison ensures correctness.

Note: Remember that the Rabin-Karp algorithm is sensitive to the choice of prime number and hash function. Adjust the PRIME value as needed for better performance.

Task 5: Boyer-Moore Algorithm Application

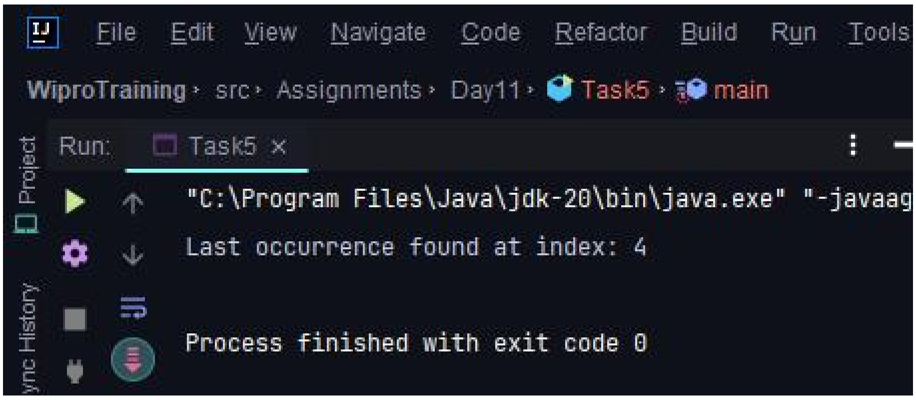
Use the Boyer-Moore algorithm to write a function that finds the last occurrence of a substring in a given string and returns its index. Explain why this algorithm can outperform others in certain scenarios.

Program:

```
package Assignments.Day11;

public class Task5 {

    public static int BoyerMooreLastOccurrence(String text, String pattern) {
        int n = text.length();
        int m = pattern.length();
        int[] lastOccurrence = new int[256]; // Initialize with -1
```

```java
        // Precompute last occurrence of each character in the pattern
        for (int i = 0; i < m; i++) {
            lastOccurrence[pattern.charAt(i)] = i;
        }

        int i = m - 1; // Start from the end of the pattern
        int j = m - 1; // Start matching from the end of the pattern

        while (i < n) {
            if (pattern.charAt(j) == text.charAt(i)) {
                if (j == 0) {
                    // Pattern found at index i
                    return i;
                }
                i--;
                j--;
            } else {
                // Apply bad character shift
                i += m - Math.min(j, 1 + lastOccurrence[text.charAt(i)]);
                j = m - 1;
            }
        }

        return -1; // Pattern not found
    }

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

        int lastIndex = BoyerMooreLastOccurrence(text, pattern);
        if (lastIndex != -1) {
            System.out.println("Last occurrence found at index: " + lastIndex);
        } else {
            System.out.println("Pattern not found in the text.");
        }
    }
}
```

Output:



Boyer-Moore Can Be Efficient:

- Boyer-Moore performs fewer character comparisons by skipping ahead based on precomputed information.

- It is particularly efficient when:

  1. The pattern has a large alphabet (many distinct characters).

  2. The text contains long sequences of the same character (e.g., DNA sequences).

  3. The pattern has a repeating structure (e.g., periodic patterns).