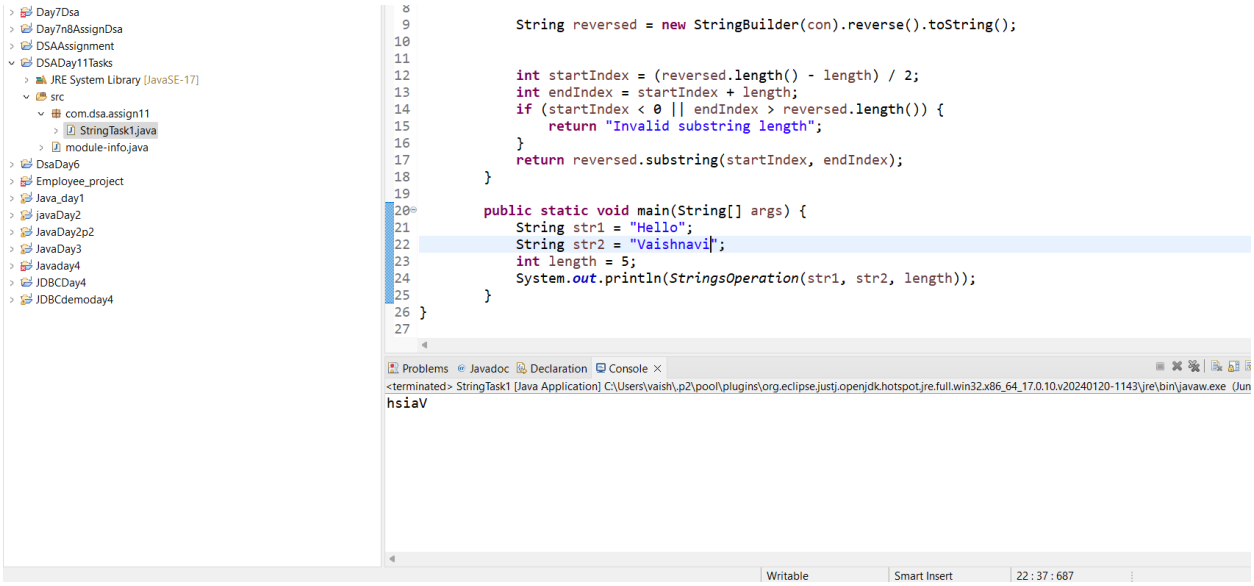
Name: Vaishnavi Vinod Ingole

Mail: vaishnavingole54@gmail.com

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.

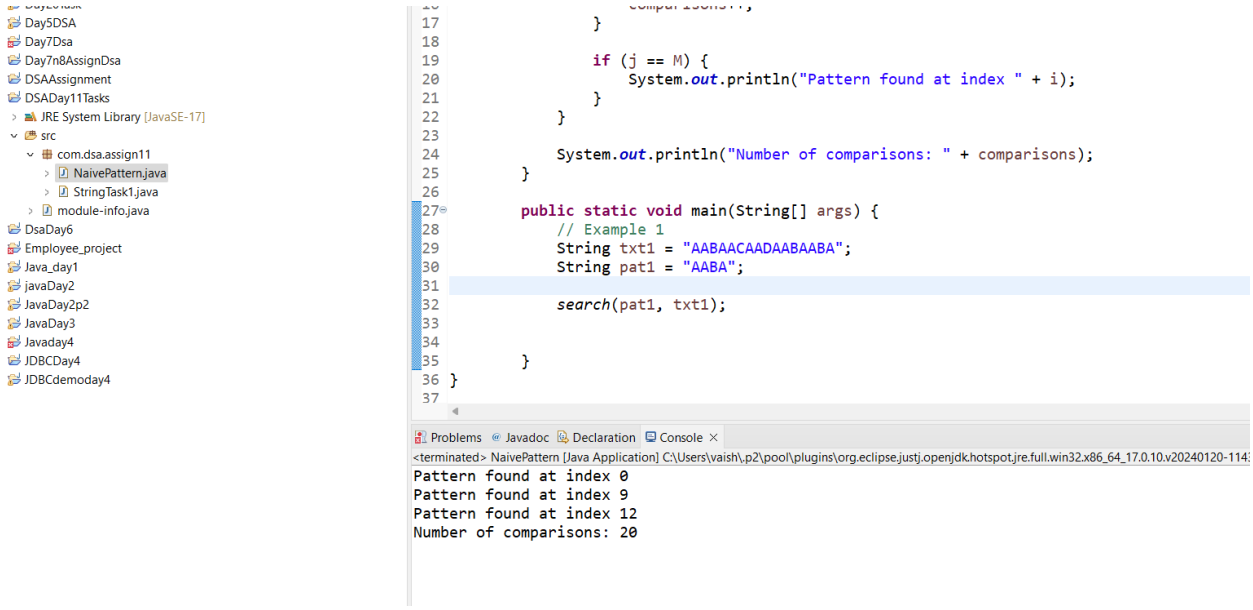
```
package com.dsa.assign11;

public class StringTask1 {
	public static String StringsOperation(String str1, String str2,
int length) {

		String con = str1.concat(str2);


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


		int startIndex = (reversed.length() - length) / 2;
		int endIndex = startIndex + length;
		if (startIndex < 0 || endIndex > reversed.length()) {
		    return "Invalid substring length";
		}
		return reversed.substring(startIndex, endIndex);
	}

	public static void main(String[] args) {
		String str1 = "Hello";
		String str2 = "Vaishnavi";
		int length = 5;
		System.out.println(StringsOperation(str1, str2, length));
	}
}
```



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 com.dsa.assign11;

public class NaivePattern {
	public static void search(String pat, String txt) {
		int M = pat.length();
		int N = txt.length();
		int comparisons = 0;


		for (int i = 0; i <= N - M; i++) {
		    int j = 0;


		    while (j < M && txt.charAt(i + j) == pat.charAt(j)) {
		        j++;
		        comparisons++;
		    }

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

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

        public static void main(String[] args) {
            // Example 1
            String txt1 = "AABAACAADAABAABA";
            String pat1 = "AABA";

            search(pat1, txt1);


        }
}
```



ask 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.dsa.assign11;

public class KMP {
	public static void search(String pat, String txt) {
```

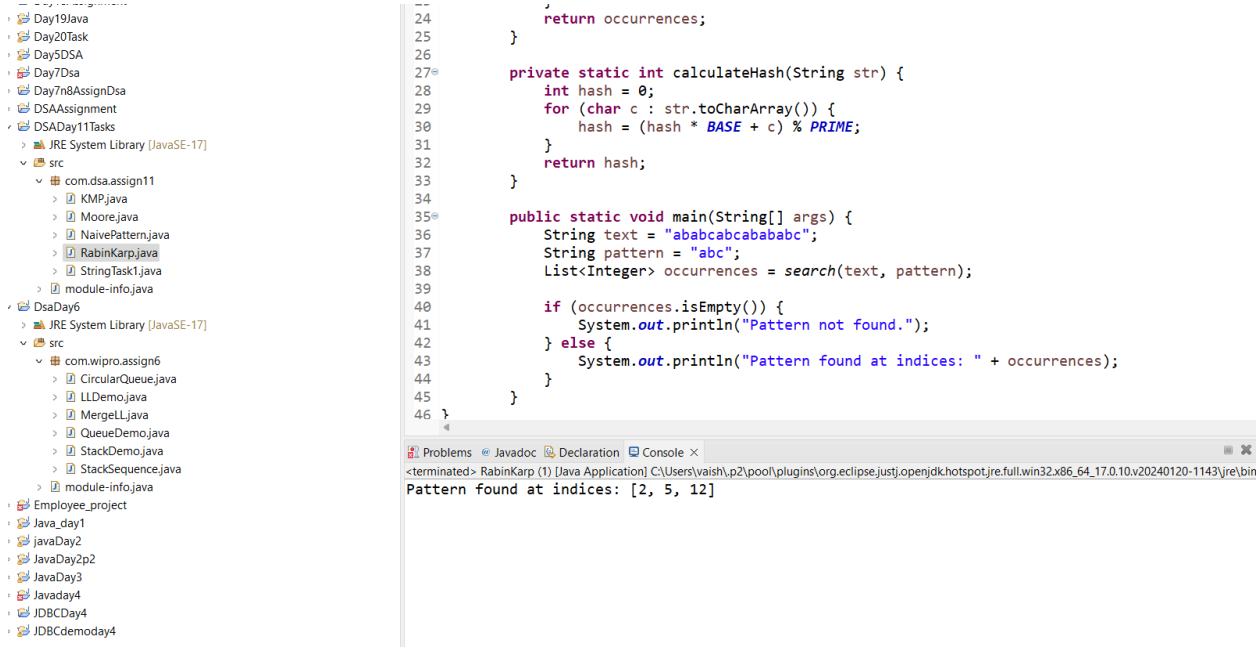
```
            int M = pat.length();
            int N = txt.length();
            int[] lps = new int[M];
            int j = 0;
            int i = 0;
            int comparisons = 0;


            computeLPSArray(pat, M, lps);

            while (i < N) {
                comparisons++;
                if (pat.charAt(j) == txt.charAt(i)) {
                    j++;
                    i++;
                }

                if (j == M) {
                    System.out.println("Pattern found at index " + (i
- j));
                    j = lps[j - 1];
                } else if (i < N && pat.charAt(j)!= txt.charAt(i)) {
                    if (j!= 0)
                        j = lps[j - 1];
                    else
                        i = i + 1;
                }
            }

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

        public static void computeLPSArray(String pat, int M, int[]
lps) {
            int length = 0;
            lps[0] = 0;

            int i = 1;
            while (i < M) {
                if (pat.charAt(i) == pat.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 txt2 = "agGd";
        String pat2 = "gG";

        search(pat2, txt2);
    }
}
```

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 com.dsa.assign11;

import java.util.ArrayList;
import java.util.List;
```

```java
public class RabinKarp {
        private static final int PRIME = 101;
         private static final int BASE = 256;

         public static List<Integer> search(String text, String
pattern) {
             List<Integer> occurrences = new ArrayList<>();
             int n = text.length();
             int m = pattern.length();
             int patternHash = calculateHash(pattern);

             for (int i = 0; i <= n - m; i++) {
                 String substring = text.substring(i, i + m);
                 int substringHash = calculateHash(substring);

                 if (substringHash == patternHash &&
substring.equals(pattern)) {
                     occurrences.add(i);
                 }
             }
             return occurrences;
         }

         private static int calculateHash(String str) {
             int hash = 0;
             for (char c : str.toCharArray()) {
                 hash = (hash * BASE + c) % PRIME;
             }
             return hash;
         }

         public static void main(String[] args) {
             String text = "ababcabcababababc";
             String pattern = "abc";
             List<Integer> occurrences = search(text, pattern);

             if (occurrences.isEmpty()) {
                 System.out.println("Pattern not found.");
             } else {
                 System.out.println("Pattern found at indices: " +
occurrences);
             }
         }
}
```



Impact of Hash Collisions

1. **False Positives**: If there is a hash collision, the algorithm might think it has found a match when it hasn't. This means that even when the hash values match, the algorithm needs to do an additional check by comparing the actual strings to confirm the match.
2. **Performance**: It is generally efficient, the occurrence of hash collisions can degrade performance because each collision requires additional string comparison checks. The more collisions, the more comparisons, and the slower the algorithm.
3. **Correctness**: Hash collisions do not affect the correctness of the Rabin-Karp algorithm because it always performs a final string comparison when the hash values match. However, if there were no final string comparison step, collisions could lead to incorrect matches.

Task 5: Boyer-Moore Algorithm Application

Use theBoyer-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.

Boyer-Moore's combination of bad character and good suffix heuristics allows it to skip large portions of the text, making it exceptionally fast in practice for many types of input. This makes it superior to other algorithms for many real-world applications, especially with large texts and longer patterns.

```
package com.dsa.assign11;
```

```java
public class Moore {
     public static int findLastOccurrence(String text, String pattern)
{
        int n = text.length();
        int m = pattern.length();
        int[] lastOccurrence = new int[256];


        for (int i = 0; i < 256; i++) {
            lastOccurrence[i] = -1;
        }
        for (int i = 0; i < m; i++) {
            lastOccurrence[pattern.charAt(i)] = i;
        }


        int i = m - 1;
        int j = m - 1;
        while (j < n) {
            if (text.charAt(j) == pattern.charAt(i)) {
                if (i == 0) {
                    return j;
                }
                i--;
                j--;
            } else {

                int      shift      =      Math.max(1,      i      -
lastOccurrence[text.charAt(j)]);
                j += shift;
                i = m - 1;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        String text = "Vaikshnaviiivaishnaviik";
        String pattern = "ik";
        int lastIndex = findLastOccurrence(text, pattern);

        if (lastIndex != -1) {
            System.out.println("Last  occurrence  found  at  index:  "  +
lastIndex);
        } else {
```

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