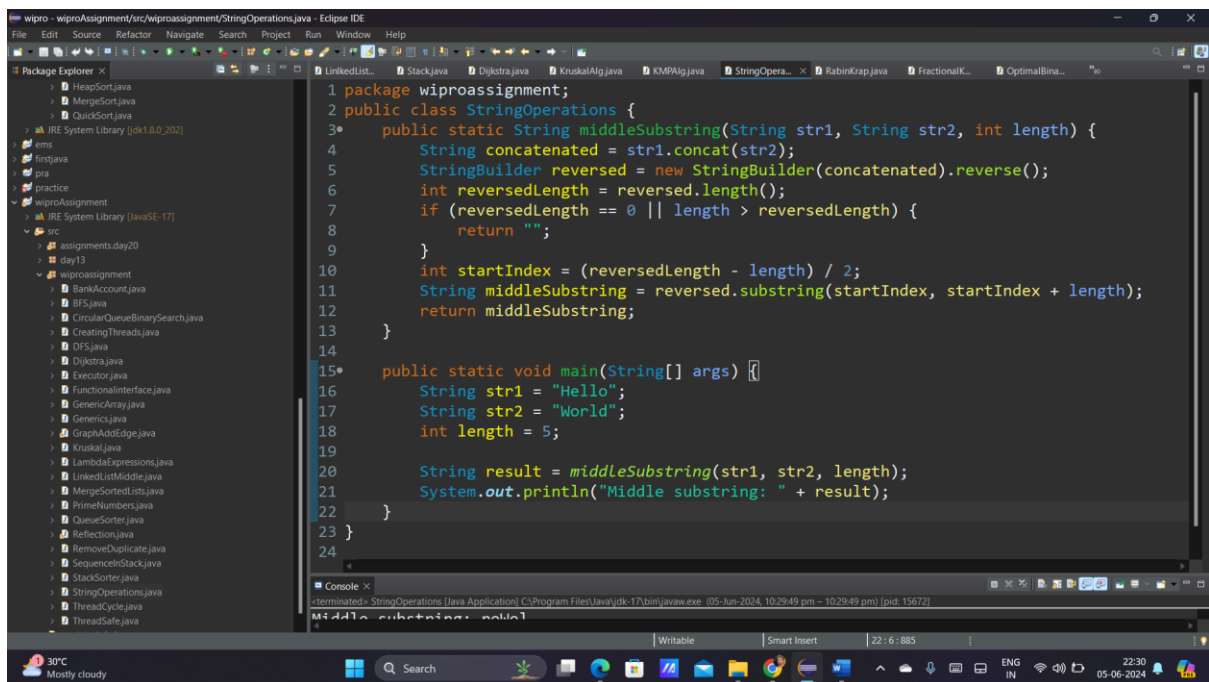


Assignment-Day 11

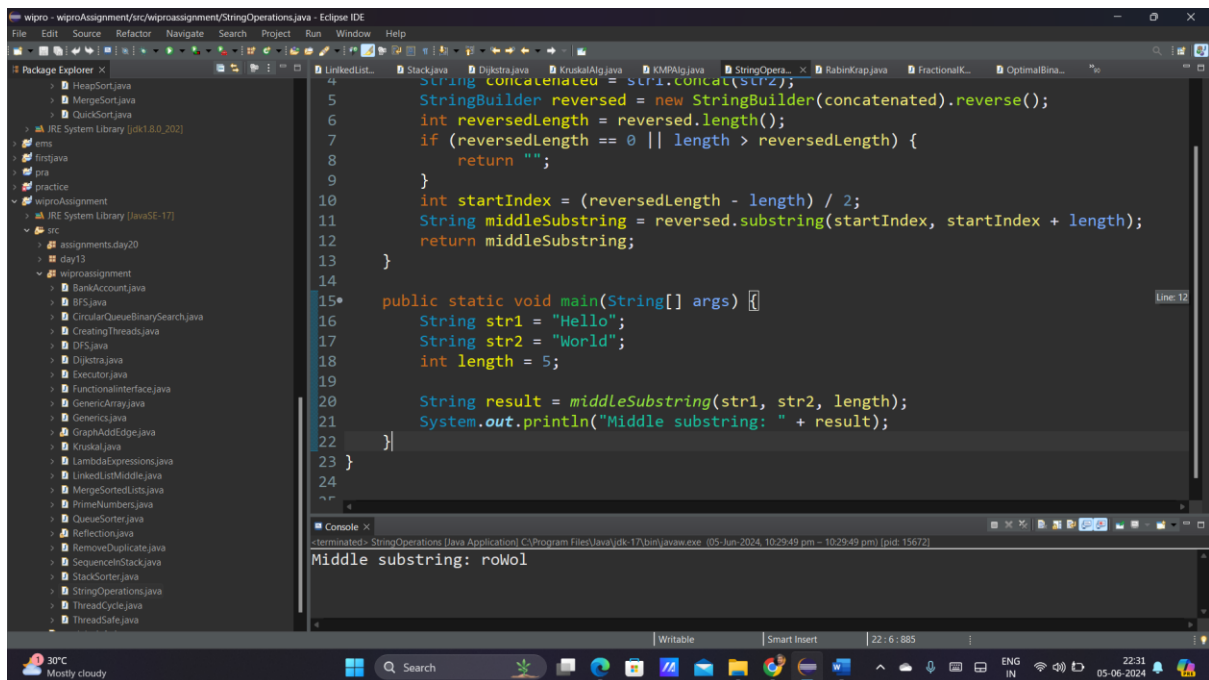
Task 1: String Operations

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.



```
1 package wiproassignment;
2 public class StringOperations {
3     public static String middleSubstring(String str1, String str2, int length) {
4         String concatenated = str1.concat(str2);
5         StringBuilder reversed = new StringBuilder(concatenated).reverse();
6         int reversedLength = reversed.length();
7         if (reversedLength == 0 || length > reversedLength) {
8             return "";
9         }
10        int startIndex = (reversedLength - length) / 2;
11        String middleSubstring = reversed.substring(startIndex, startIndex + length);
12        return middleSubstring;
13    }
14
15    public static void main(String[] args) {
16        String str1 = "Hello";
17        String str2 = "World";
18        int length = 5;
19
20        String result = middleSubstring(str1, str2, length);
21        System.out.println("Middle substring: " + result);
22    }
23 }
24
```

Console Output: Middle substring: nola

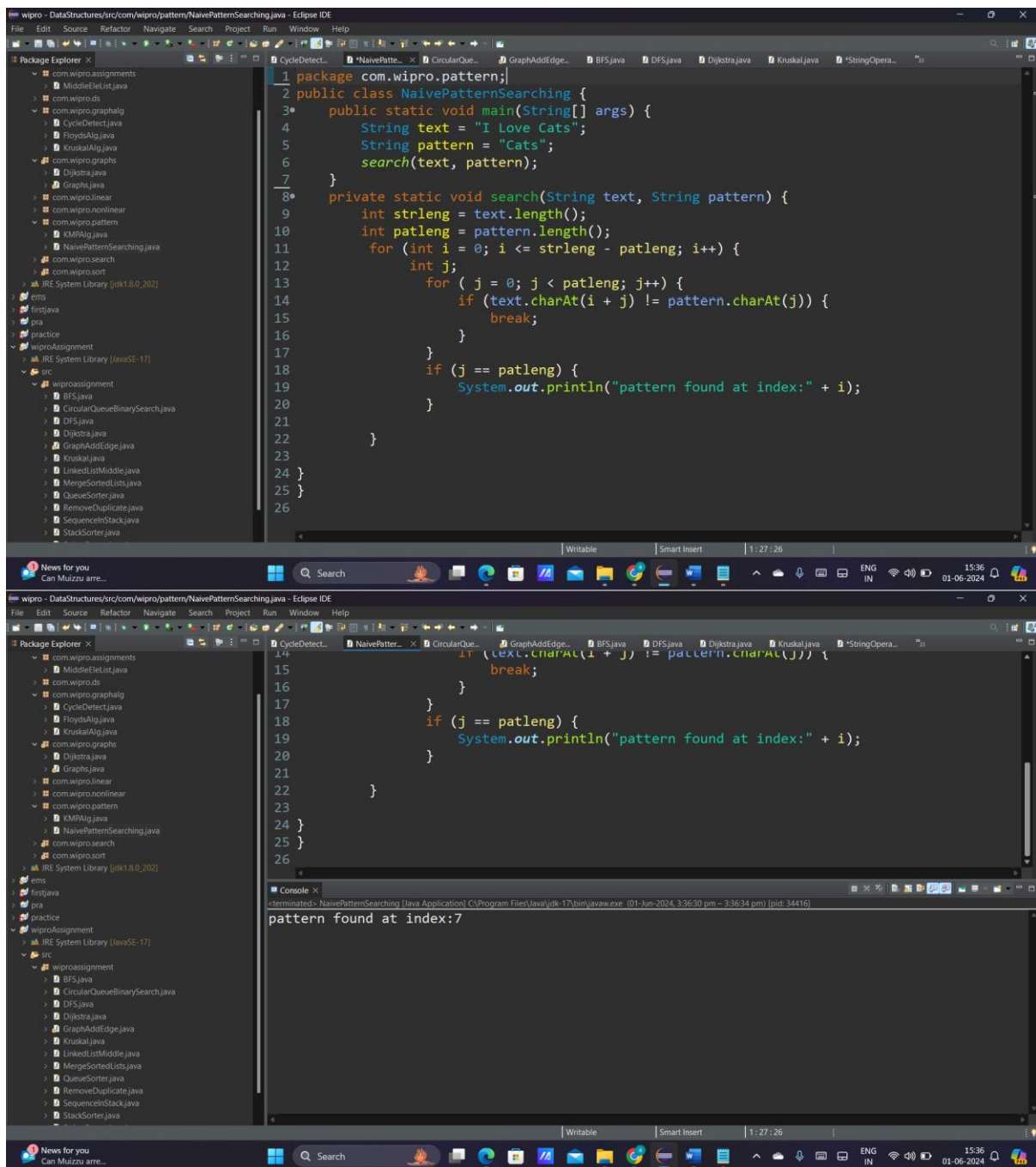


```
4      String concatenated = str1.concat(str2);
5      StringBuilder reversed = new StringBuilder(concatenated).reverse();
6      int reversedLength = reversed.length();
7      if (reversedLength == 0 || length > reversedLength) {
8          return "";
9      }
10     int startIndex = (reversedLength - length) / 2;
11     String middleSubstring = reversed.substring(startIndex, startIndex + length);
12     return middleSubstring;
13 }
14
15 public static void main(String[] args) {
16     String str1 = "Hello";
17     String str2 = "World";
18     int length = 5;
19
20     String result = middleSubstring(str1, str2, length);
21     System.out.println("Middle substring: " + result);
22 }
23 }
24 }
```

Console Output: Middle substring: roWoL

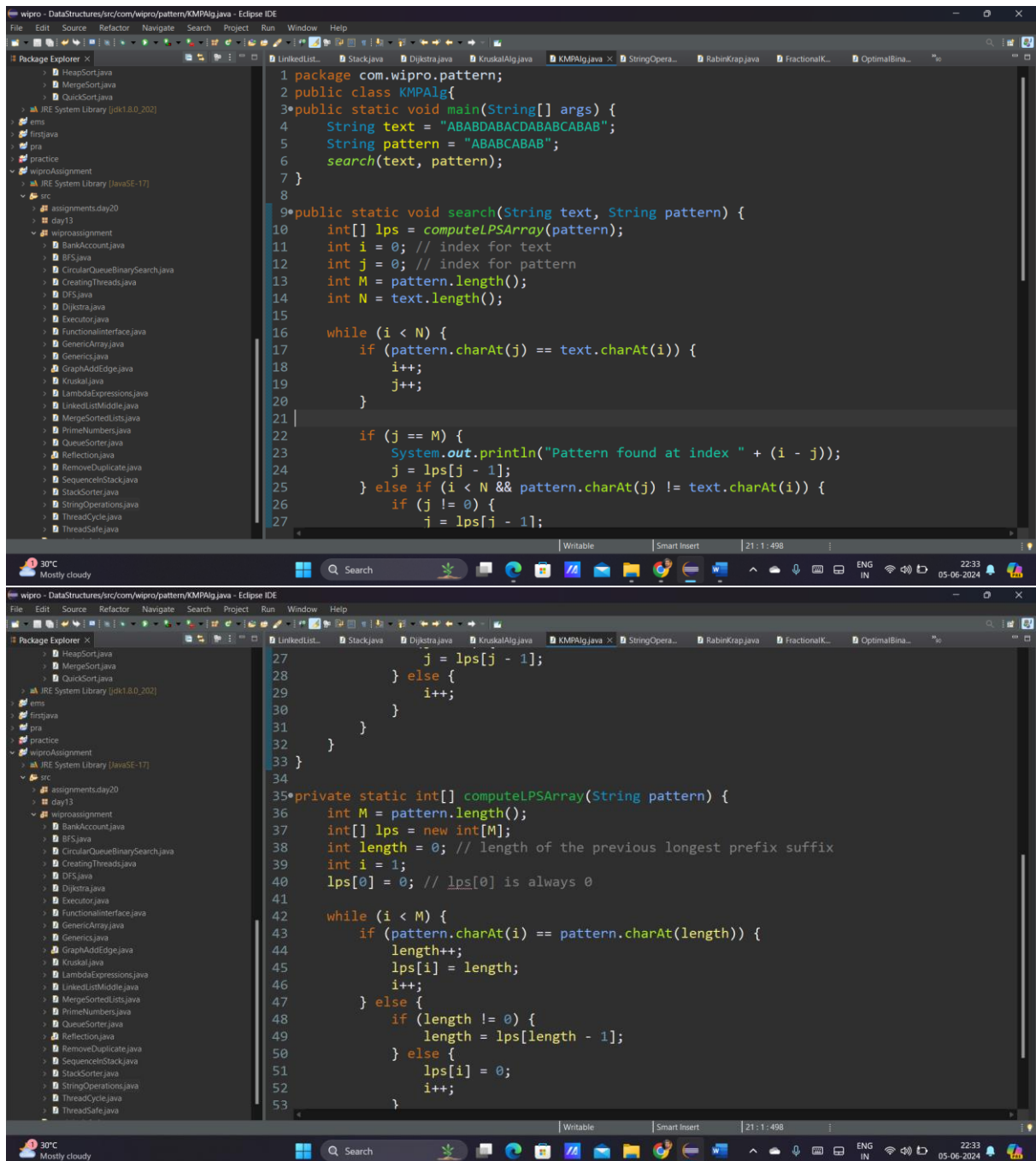
Task 2: Naive Pattern Search

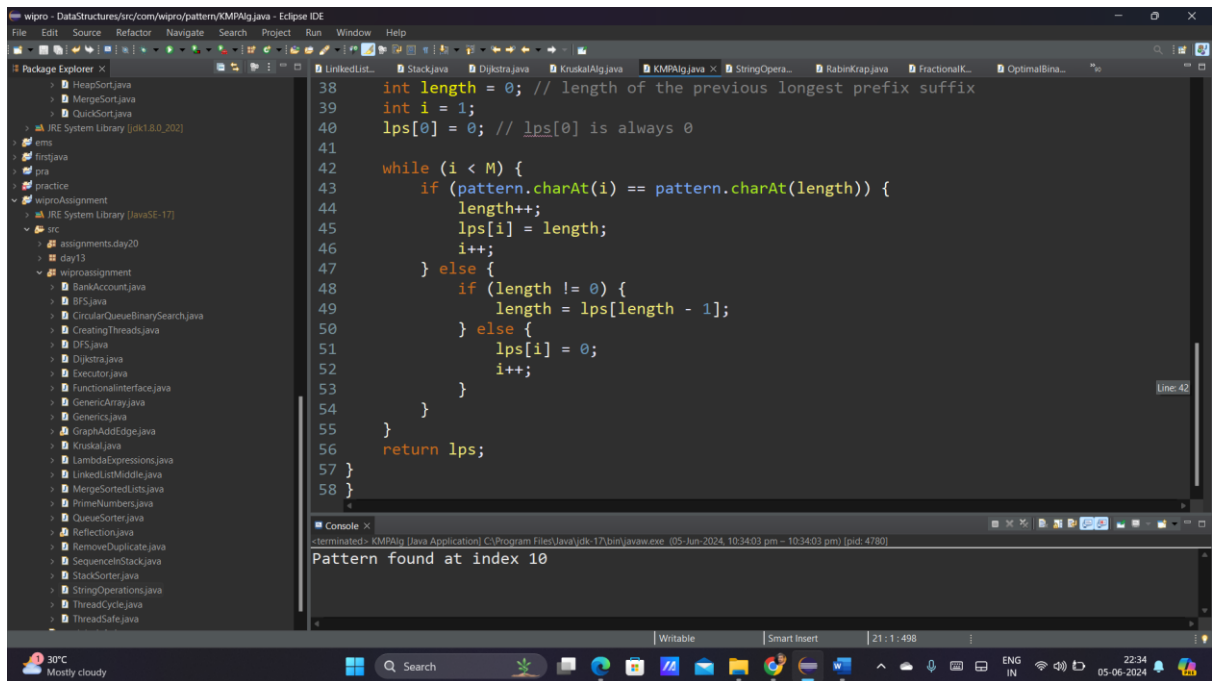
Implement the naive pattern searching algorithm to find all occurrences of a pattern within a giventext string. Count the number of comparisons made during the search to evaluate the efficiency ofthe algorithm.



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.



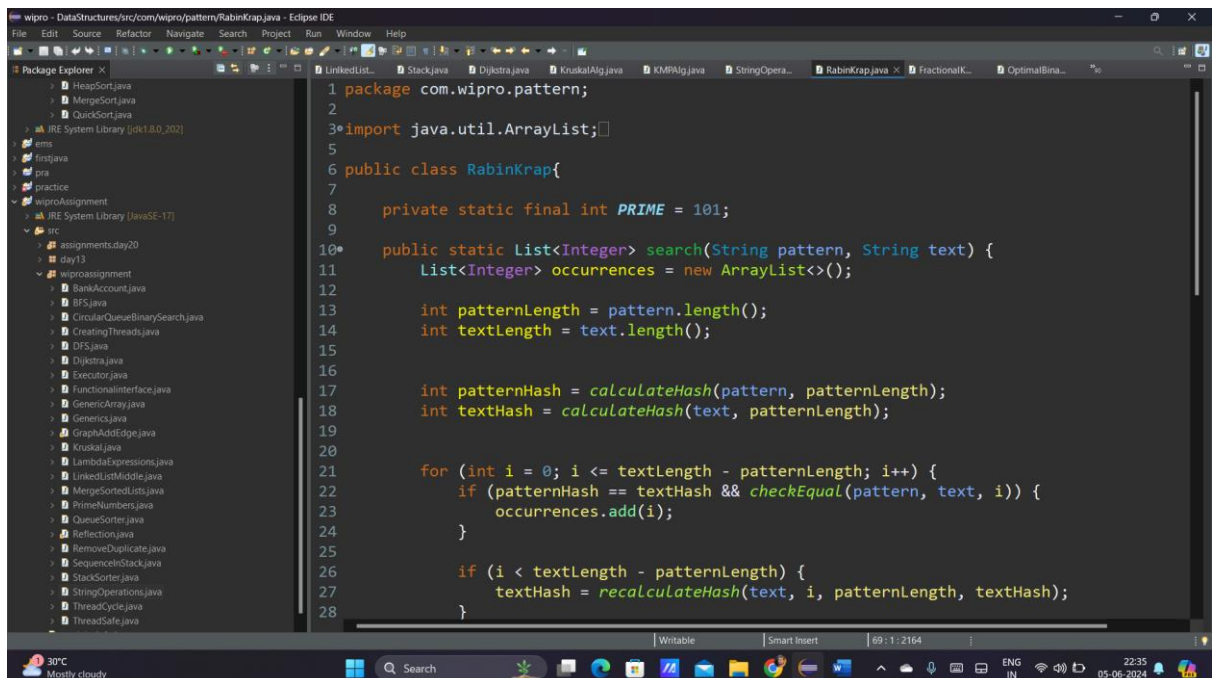


```
38 int length = 0; // length of the previous longest prefix suffix
39 int i = 1;
40 lps[0] = 0; // lps[0] is always 0
41
42 while (i < M) {
43     if (pattern.charAt(i) == pattern.charAt(length)) {
44         length++;
45         lps[i] = length;
46         i++;
47     } else {
48         if (length != 0) {
49             length = lps[length - 1];
50         } else {
51             lps[i] = 0;
52             i++;
53         }
54     }
55 }
56 return lps;
57 }
58 }
```

Console Output:
-terminated> KMPAlg [Java Application] C:\Program Files\Java\jdk-17\bin\java.exe (05-Jun-2024, 10:34:03 pm - 10:34:03 pm) (pid: 4780)
Pattern found at index 10

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.



```
1 package com.wipro.pattern;
2
3 import java.util.ArrayList;
4
5
6 public class RabinKrap{
7
8     private static final int PRIME = 101;
9
10    public static List<Integer> search(String pattern, String text) {
11        List<Integer> occurrences = new ArrayList<>();
12
13        int patternLength = pattern.length();
14        int textLength = text.length();
15
16        int patternHash = calculateHash(pattern, patternLength);
17        int textHash = calculateHash(text, patternLength);
18
19        for (int i = 0; i <= textLength - patternLength; i++) {
20            if (patternHash == textHash && checkEqual(pattern, text, i)) {
21                occurrences.add(i);
22            }
23
24            if (i < textLength - patternLength) {
25                textHash = recalculateHash(text, i, patternLength, textHash);
26            }
27        }
28    }
```

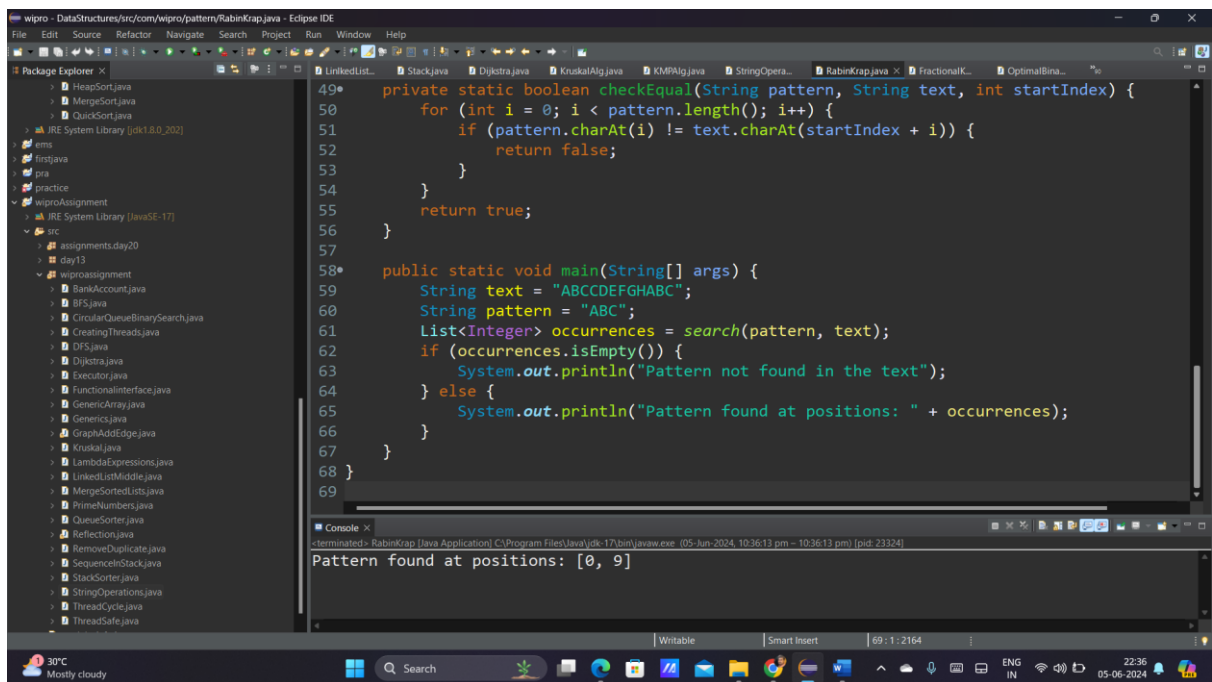
The image displays two screenshots of an Eclipse IDE window, showing the implementation of the Rabin-Karp algorithm in Java. The top screenshot shows the 'calculateHash' and 'recalculateHash' methods, while the bottom screenshot shows the 'checkEqual' method and the 'main' method.

Top Screenshot:

```
28     }
29     }
30
31     return occurrences;
32 }
33
34* private static int calculateHash(String str, int length) {
35     int hash = 0;
36     for (int i = 0; i < length; i++) {
37         hash += str.charAt(i) * Math.pow(PRIME, i);
38     }
39     return hash;
40 }
41
42* private static int recalculateHash(String str, int oldIndex, int patternLength, int oldHash) {
43     int newHash = oldHash - str.charAt(oldIndex);
44     newHash /= PRIME;
45     newHash += str.charAt(oldIndex + patternLength) * Math.pow(PRIME, patternLength - 1);
46     return newHash;
47 }
48
49* private static boolean checkEqual(String pattern, String text, int startIndex) {
50     for (int i = 0; i < pattern.length(); i++) {
51         if (pattern.charAt(i) != text.charAt(startIndex + i)) {
52             return false;
53         }
54     }
55     return true;
56 }
```

Bottom Screenshot:

```
43     int newHash = oldHash - str.charAt(oldIndex);
44     newHash /= PRIME;
45     newHash += str.charAt(oldIndex + patternLength) * Math.pow(PRIME, patternLength - 1);
46     return newHash;
47 }
48
49* private static boolean checkEqual(String pattern, String text, int startIndex) {
50     for (int i = 0; i < pattern.length(); i++) {
51         if (pattern.charAt(i) != text.charAt(startIndex + i)) {
52             return false;
53         }
54     }
55     return true;
56 }
57
58* public static void main(String[] args) {
59     String text = "ABCCDEFGHABC";
60     String pattern = "ABC";
61     List<Integer> occurrences = search(pattern, text);
62     if (occurrences.isEmpty()) {
63         System.out.println("Pattern not found in the text");
64     } else {
65         System.out.println("Pattern found at positions: " + occurrences);
66     }
67 }
68 }
69 }
```

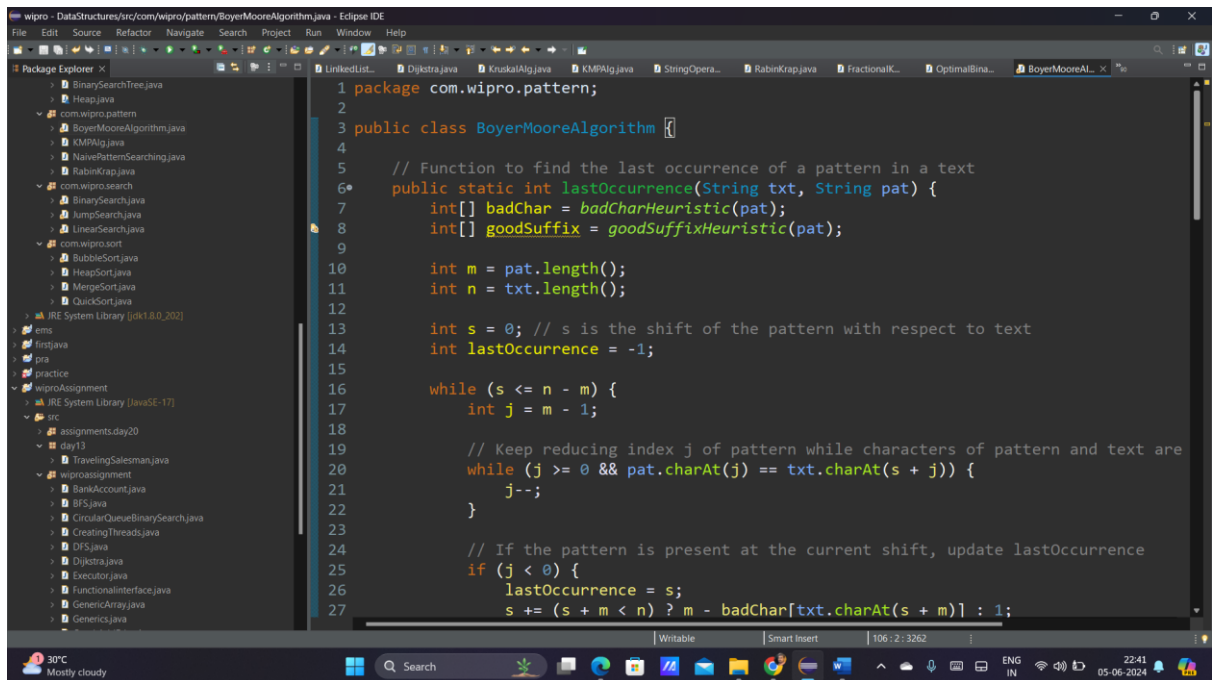
The screenshot shows the Eclipse IDE with the file `RabinKrap.java` open. The code implements a Rabin-Karp algorithm for string matching. It defines a `checkEqual` method to compare a pattern with a text substring and a `main` method that uses the algorithm to find occurrences of the pattern "ABC" in the text "ABCCDEFGHABC". The console output shows "Pattern found at positions: [0, 9]".

```
49 private static boolean checkEqual(String pattern, String text, int startIndex) {
50     for (int i = 0; i < pattern.length(); i++) {
51         if (pattern.charAt(i) != text.charAt(startIndex + i)) {
52             return false;
53         }
54     }
55     return true;
56 }
57
58 public static void main(String[] args) {
59     String text = "ABCCDEFGHABC";
60     String pattern = "ABC";
61     List<Integer> occurrences = search(pattern, text);
62     if (occurrences.isEmpty()) {
63         System.out.println("Pattern not found in the text");
64     } else {
65         System.out.println("Pattern found at positions: " + occurrences);
66     }
67 }
68 }
69 }
```

Console Output: Pattern found at positions: [0, 9]

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.



The screenshot shows the Eclipse IDE with the file `BoyerMooreAlgorithm.java` open. The code implements the Boyer-Moore algorithm for finding the last occurrence of a pattern in a text. It defines a `lastOccurrence` method that uses the `badChar` and `goodSuffix` heuristics to shift the pattern and find the last occurrence. The code is partially visible, showing the package declaration, class definition, and the start of the `lastOccurrence` method.

```
1 package com.wipro.pattern;
2
3 public class BoyerMooreAlgorithm {
4
5     // Function to find the last occurrence of a pattern in a text
6     public static int lastOccurrence(String txt, String pat) {
7         int[] badChar = badCharHeuristic(pat);
8         int[] goodSuffix = goodSuffixHeuristic(pat);
9
10        int m = pat.length();
11        int n = txt.length();
12
13        int s = 0; // s is the shift of the pattern with respect to text
14        int lastOccurrence = -1;
15
16        while (s <= n - m) {
17            int j = m - 1;
18
19            // Keep reducing index j of pattern while characters of pattern and text are
20            while (j >= 0 && pat.charAt(j) == txt.charAt(s + j)) {
21                j--;
22            }
23
24            // If the pattern is present at the current shift, update lastOccurrence
25            if (j < 0) {
26                lastOccurrence = s;
27                s += (s + m < n) ? m - badChar[txt.charAt(s + m)] : 1;
```