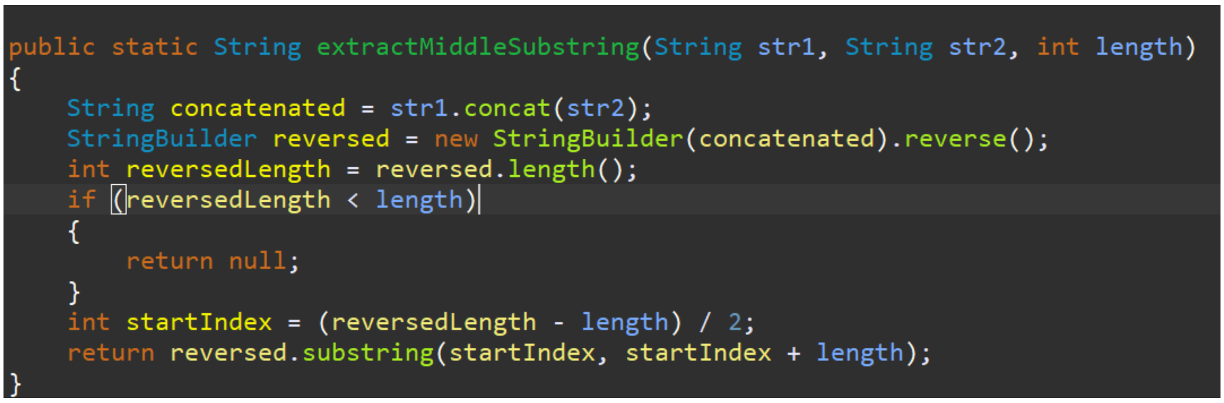
Task 1: String Operations

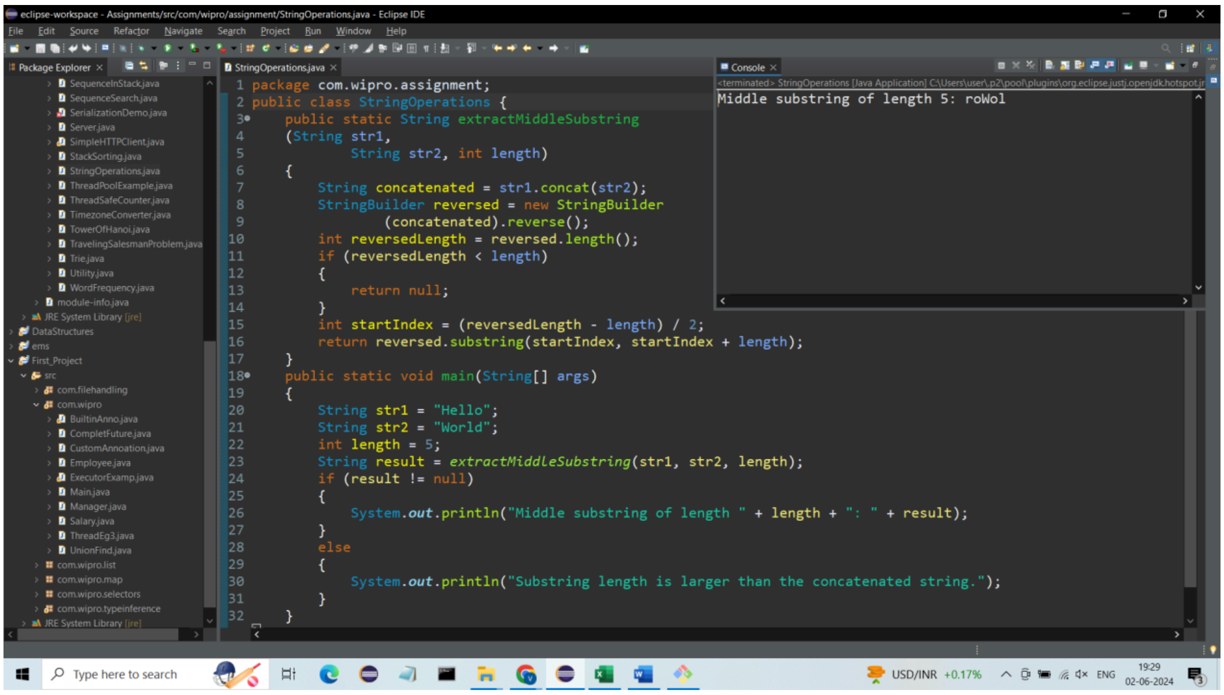
Write a method that takes two strings, concatenates them, reverses the result, and then extracts the middle substring of the given length. Ensure your method handles edge cases, such as an empty string or a substring length larger than the concatenated string.

Method: -



```java
public static String extractMiddleSubstring(String str1, String str2, int length)
{
    String concatenated = str1.concat(str2);
    StringBuilder reversed = new StringBuilder(concatenated).reverse();
    int reversedLength = reversed.length();
    if (reversedLength < length)
    {
        return null;
    }
    int startIndex = (reversedLength - length) / 2;
    return reversed.substring(startIndex, startIndex + length);
}
```

Output: -



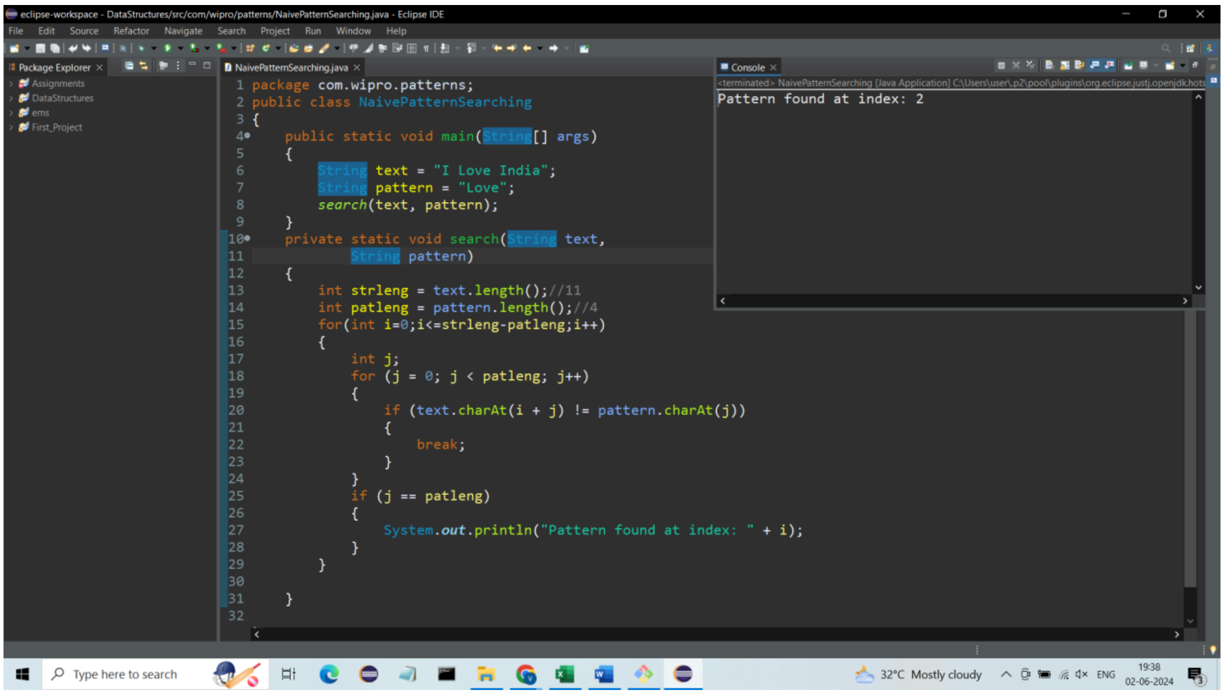
eclipse-workspace - Assignments/src/com/wipro/assignment/StringOperations.java - Eclipse IDE

```java
package com.wipro.assignment;
public class StringOperations {
    public static String extractMiddleSubstring
    (String str1,
            String str2, int length)
    {
        String concatenated = str1.concat(str2);
        StringBuilder reversed = new StringBuilder
                (concatenated).reverse();
        int reversedLength = reversed.length();
        if (reversedLength < length)
        {
            return null;
        }
        int startIndex = (reversedLength - length) / 2;
        return reversed.substring(startIndex, startIndex + length);
    }
    public static void main(String[] args)
    {
        String str1 = "Hello";
        String str2 = "World";
        int length = 5;
        String result = extractMiddleSubstring(str1, str2, length);
        if (result != null)
        {
            System.out.println("Middle substring of length " + length + ": " + result);
        }
        else
        {
            System.out.println("Substring length is larger than the concatenated string.");
        }
    }
```

```
<terminated> StringOperations [Java Application] C:\Users\user\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jr
Middle substring of length 5: roWol
```

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.



```java
package com.wipro.patterns;
public class NaivePatternSearching
{
    public static void main(String[] args)
    {
        String text = "I Love India";
        String pattern = "Love";
        search(text, pattern);
    }
    private static void search(String text,
            String pattern)
    {
        int strleng = text.length();//11
        int patleng = pattern.length();//4
        for(int i=0;i<=strleng-patleng;i++)
        {
            int j;
            for (j = 0; j < patleng; j++)
            {
                if (text.charAt(i + j) != pattern.charAt(j))
                {
                    break;
                }
            }
            if (j == patleng)
            {
                System.out.println("Pattern found at index: " + i);
            }
        }

    }
```

```
Pattern found at index: 2
```

Here's an explanation of how the algorithm works:

1. The search function takes two inputs: the text string and the pattern string.

2. It initializes a count variable to keep track of the number of comparisons made during the search.

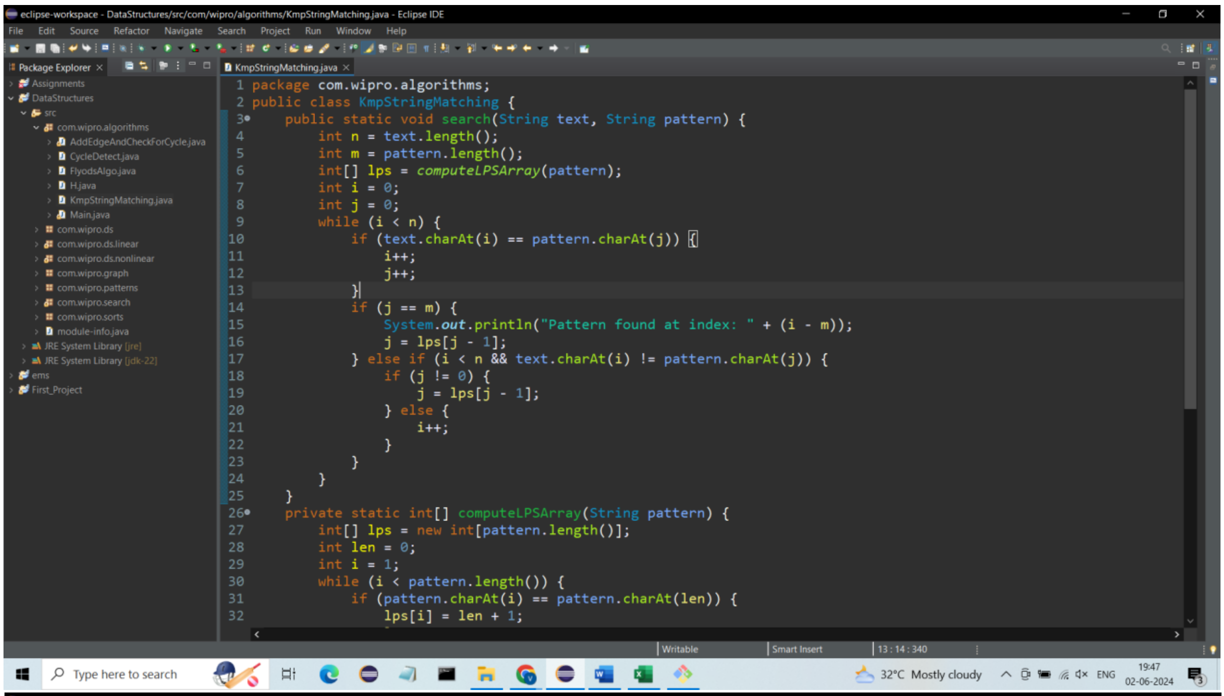
3. It iterates through the text string, starting from the first character, and checks if the pattern matches at each position.

4. For each position, it iterates through the pattern string and compares each character with the corresponding character in the text string.
5. If all characters match, it prints the index where the pattern is found.
6. The count variable is incremented for each comparison made.
7. Finally, the function returns the total number of comparisons made.

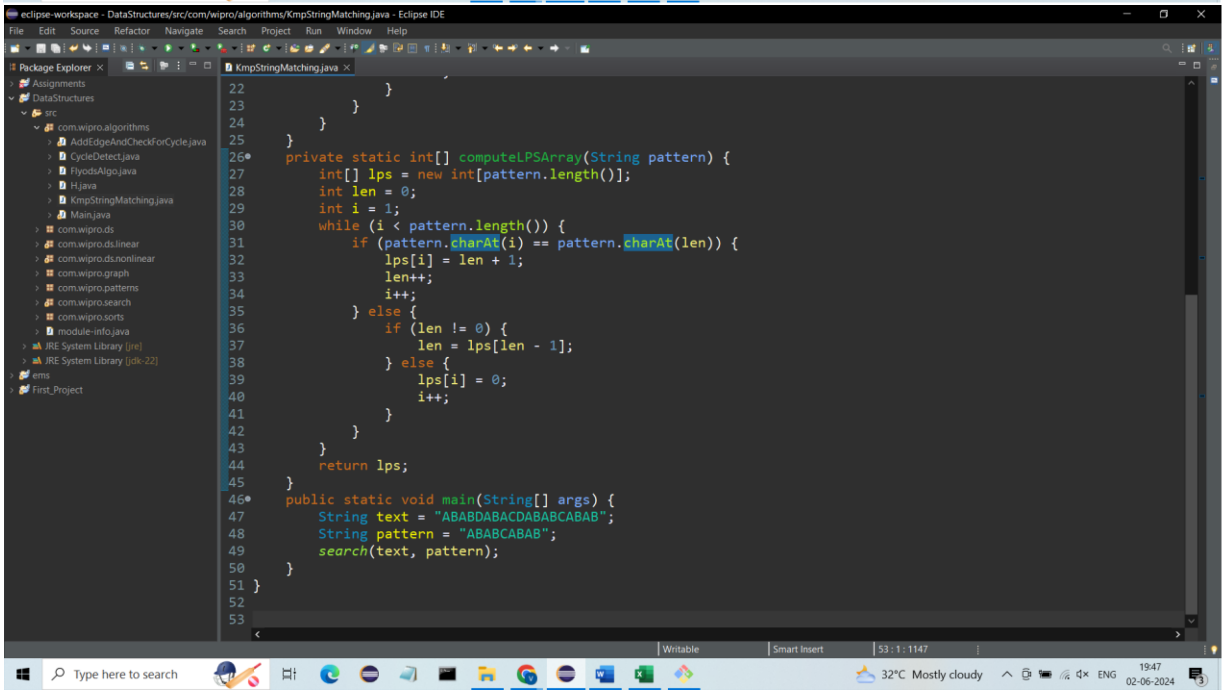
Task 3: Implementing the KMP Algorithm

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.

Code: -

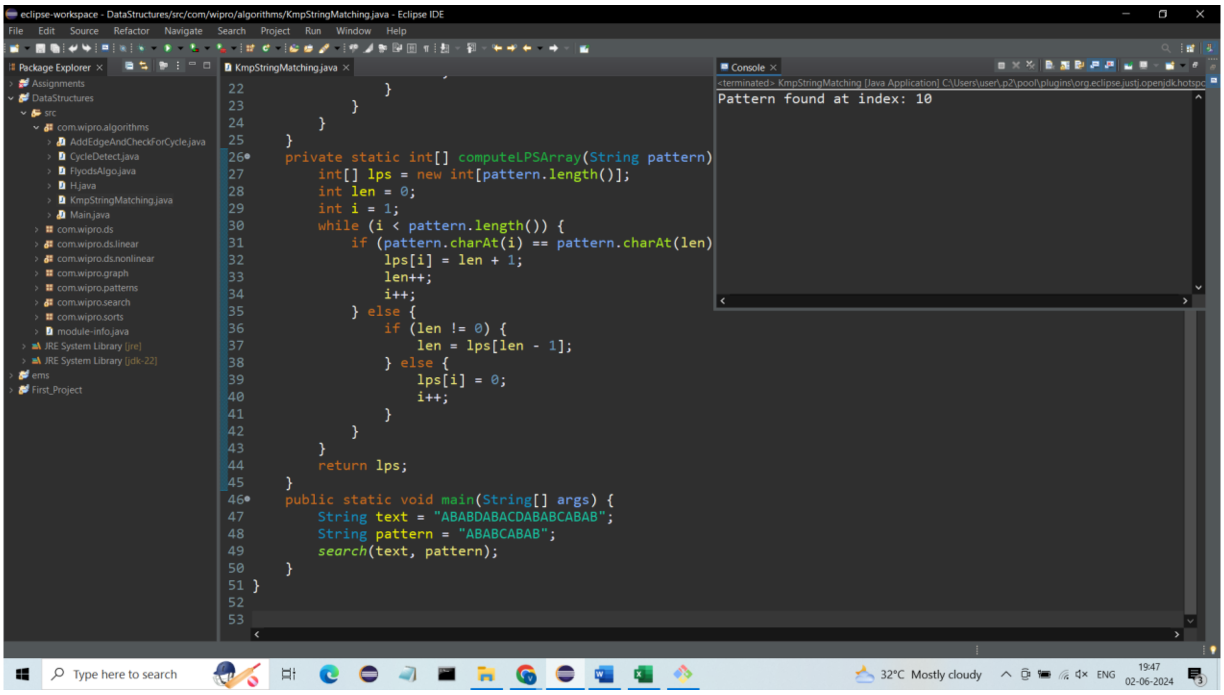

eclipse-workspace - DataStructures/src/com/wipro/algorithms/KmpStringMatching.java - Eclipse IDE
KmpStringMatching.java

```java
package com.wipro.algorithms;
public class KmpStringMatching {
    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;
```


eclipse-workspace - DataStructures/src/com/wipro/algorithms/KmpStringMatching.java - Eclipse IDE
KmpStringMatching.java

```java
                }
            }
        }
    }
    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;
    }
    public static void main(String[] args) {
        String text = "ABABDABACDABABCABAB";
        String pattern = "ABABCABAB";
        search(text, pattern);
    }
}
```

Output: -



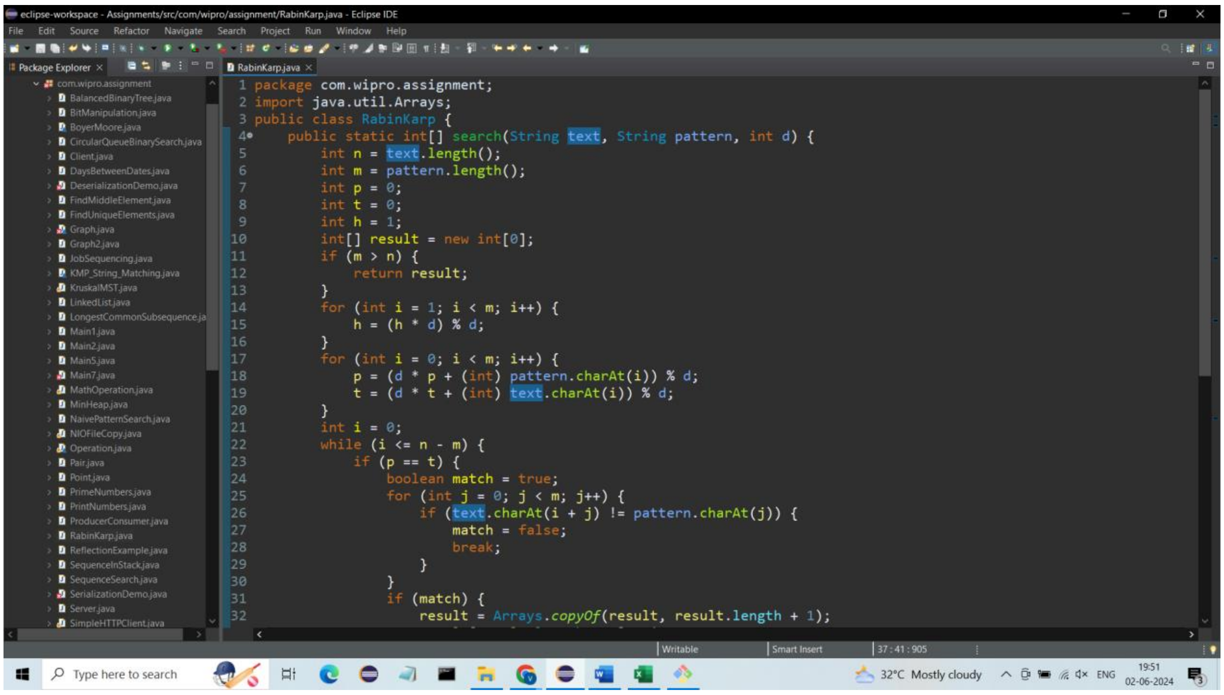
```java
                }
            }
        }
    }
    private static int[] computeLPSArray(String pattern)
        int[] lps = new int[pattern.length()];
        int len = 0;
        int i = 1;
        while (i < pattern.length()) {
            if (pattern.charAt(i) == pattern.charAt(len)
                lps[i] = len + 1;
                len++;
                i++;
            } else {
                if (len != 0) {
                    len = lps[len - 1];
                } else {
                    lps[i] = 0;
                    i++;
                }
            }
        }
        return lps;
    }
    public static void main(String[] args) {
        String text = "ABABDABACDABABCABAB";
        String pattern = "ABABCABAB";
        search(text, pattern);
    }
}
```

```
<terminated> KmpStringMatching [Java Application] C:\Users\user\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspo
Pattern found at index: 10
```

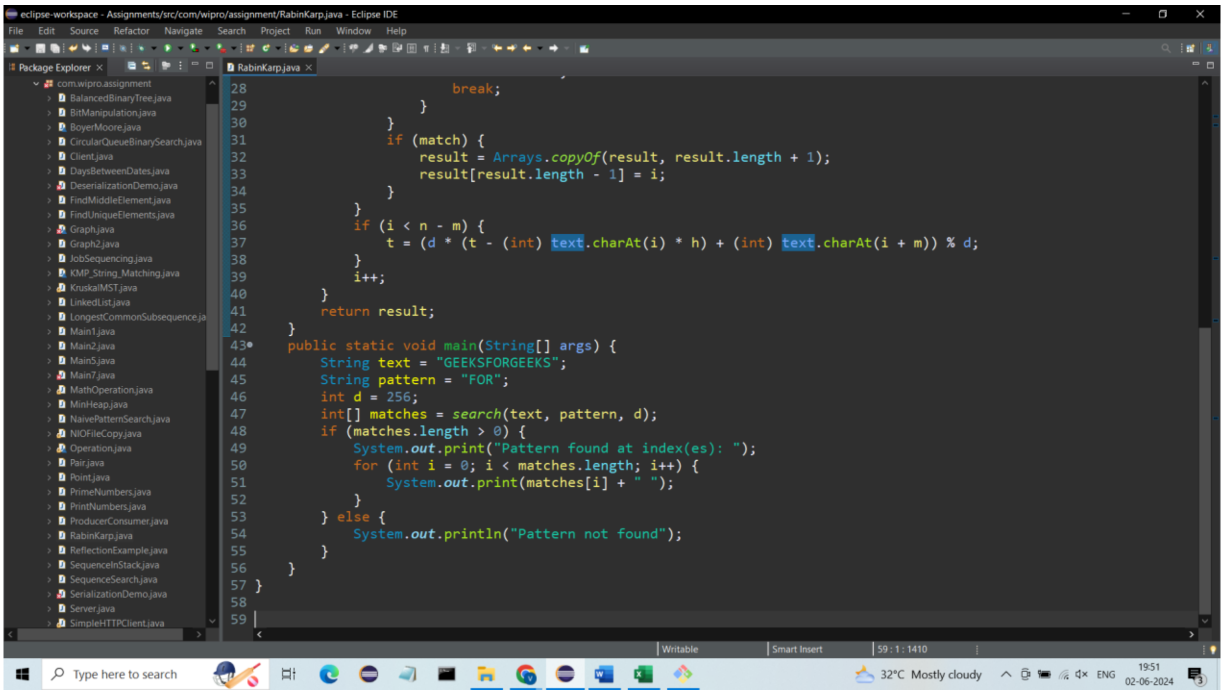
Task 04- : Rabin-Karp Substring Search

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.

Code: -

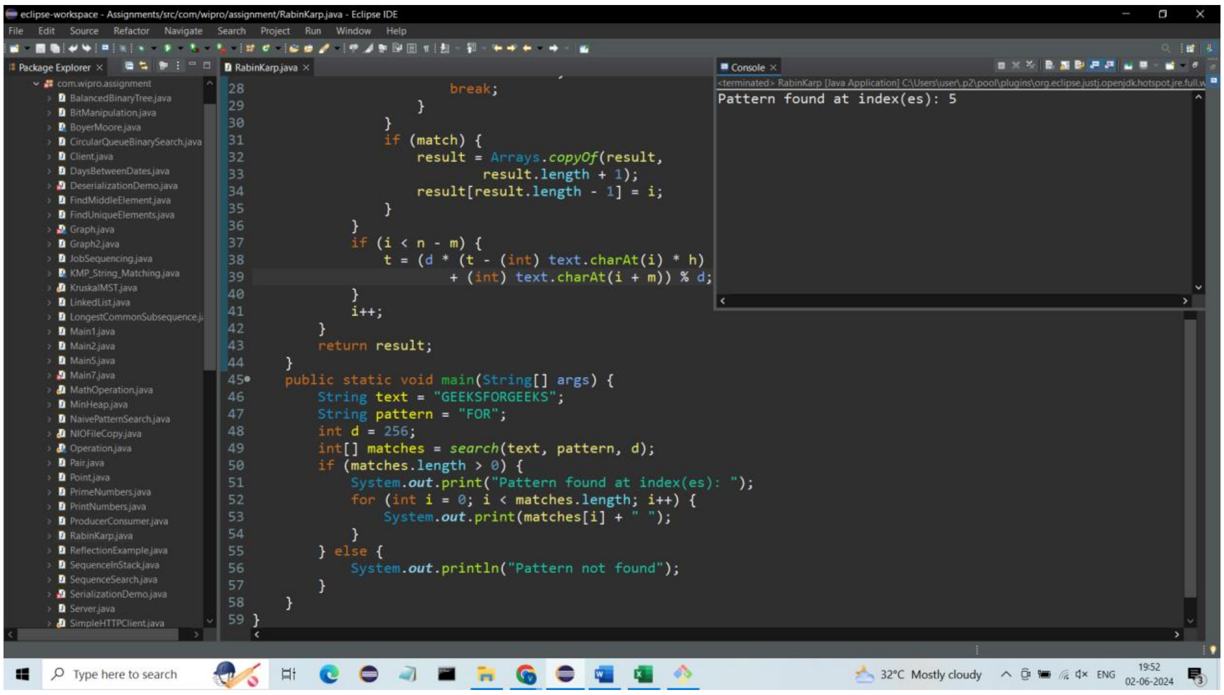


```java
package com.wipro.assignment;
import java.util.Arrays;
public class RabinKarp {
    public static int[] search(String text, String pattern, int d) {
        int n = text.length();
        int m = pattern.length();
        int p = 0;
        int t = 0;
        int h = 1;
        int[] result = new int[0];
        if (m > n) {
            return result;
        }
        for (int i = 1; i < m; i++) {
            h = (h * d) % d;
        }
        for (int i = 0; i < m; i++) {
            p = (d * p + (int) pattern.charAt(i)) % d;
            t = (d * t + (int) text.charAt(i)) % d;
        }
        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) {
                    result = Arrays.copyOf(result, result.length + 1);
```



```java
                        break;
                    }
                }
                if (match) {
                    result = Arrays.copyOf(result, result.length + 1);
                    result[result.length - 1] = i;
                }
            }
            if (i < n - m) {
                t = (d * (t - (int) text.charAt(i) * h) + (int) text.charAt(i + m)) % d;
            }
            i++;
        }
        return result;
    }
    public static void main(String[] args) {
        String text = "GEEKSFORGEEKS";
        String pattern = "FOR";
        int d = 256;
        int[] matches = search(text, pattern, d);
        if (matches.length > 0) {
            System.out.print("Pattern found at index(es): ");
            for (int i = 0; i < matches.length; i++) {
                System.out.print(matches[i] + " ");
            }
        } else {
            System.out.println("Pattern not found");
        }
    }
}
```

Output: -



```java
                    break;
                }
            }
            if (match) {
                result = Arrays.copyOf(result,
                        result.length + 1);
                result[result.length - 1] = i;
            }
        }
        if (i < n - m) {
            t = (d * (t - (int) text.charAt(i) * h)
                    + (int) text.charAt(i + m)) % d;
        }
        i++;
    }
    return result;
}
public static void main(String[] args) {
    String text = "GEEKSFORGEEKS";
    String pattern = "FOR";
    int d = 256;
    int[] matches = search(text, pattern, d);
    if (matches.length > 0) {
        System.out.print("Pattern found at index(es): ");
        for (int i = 0; i < matches.length; i++) {
            System.out.print(matches[i] + " ");
        }
    } else {
        System.out.println("Pattern not found");
    }
}
}
```

```
Pattern found at index(es): 5
```

Impact of Hash Collisions:

- Hash collisions occur when different strings map to the same hash value.
- Rabin-Karp relies on the hash values for potential matches, but hash collisions can lead to false positives (potential matches that are not actual matches).
- To handle collisions, the algorithm performs a character-by-character comparison after finding a potential match based on hash values. This additional step ensures that only actual matches are reported.

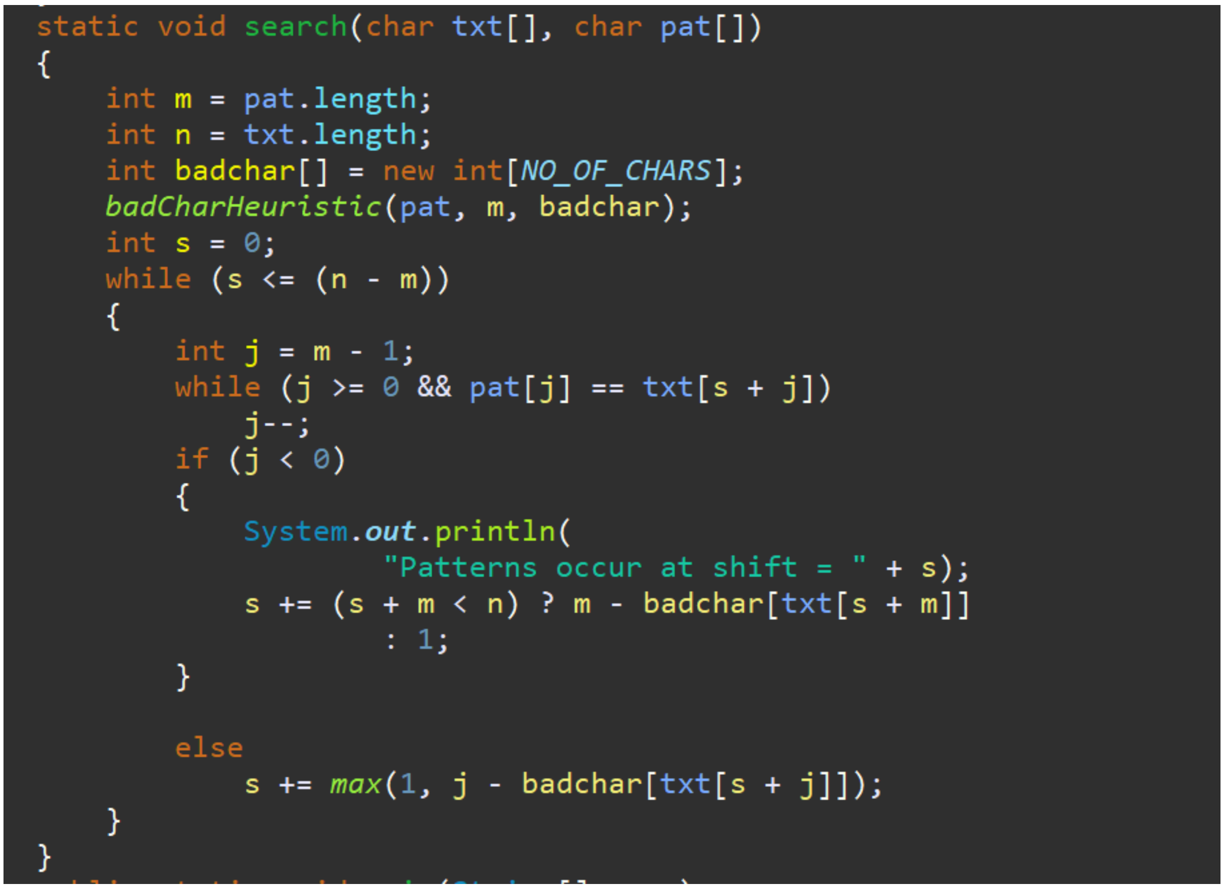
How to Mitigate Collisions:

- Using a larger prime number (q) for modular arithmetic can reduce the probability of collisions.
- Employing additional hash functions and comparing the results can improve collision detection.
- If using a good hash function and a large prime number q, the impact of collisions is usually minimal.

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.

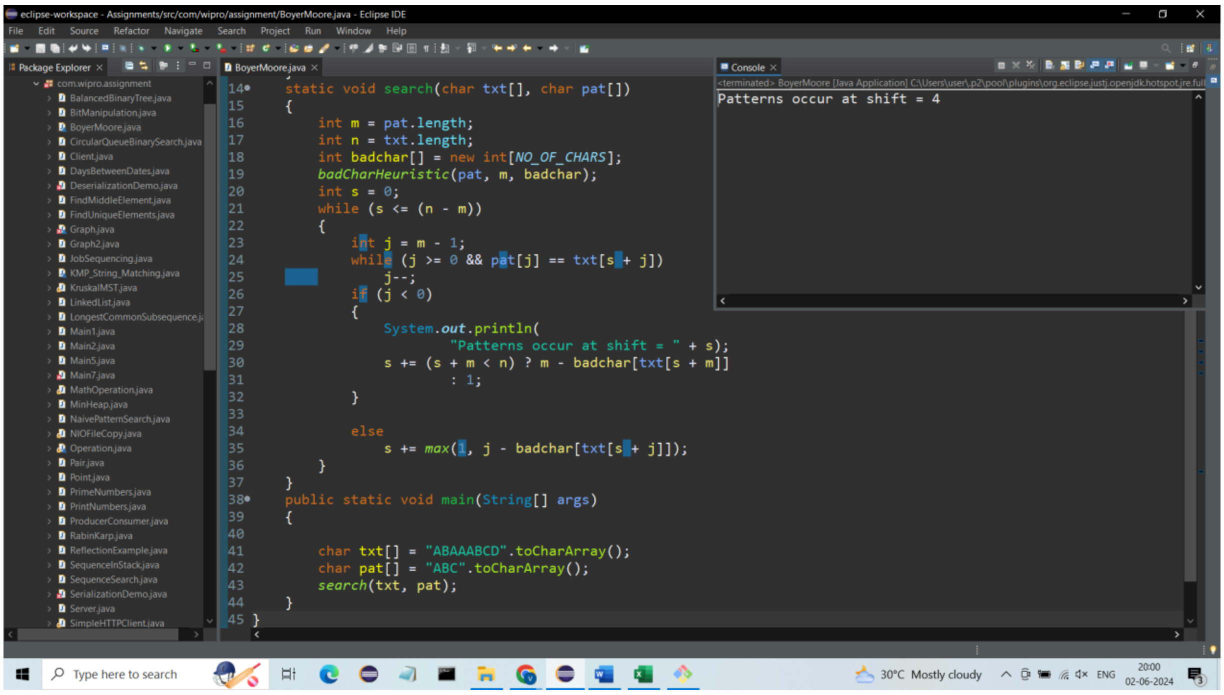
Function: -



```java
static void search(char txt[], char pat[])
{
    int m = pat.length;
    int n = txt.length;
    int badchar[] = new int[NO_OF_CHARS];
    badCharHeuristic(pat, m, badchar);
    int s = 0;
    while (s <= (n - m))
    {
        int j = m - 1;
        while (j >= 0 && pat[j] == txt[s + j])
            j--;
        if (j < 0)
        {
            System.out.println(
                    "Patterns occur at shift = " + s);
            s += (s + m < n) ? m - badchar[txt[s + m]]
                    : 1;
        }

        else
            s += max(1, j - badchar[txt[s + j]]);
    }
}
```

Output: -



eclipse-workspace - Assignments/src/com/wipro/assignment/BoyerMoore.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
BoyerMoore.java
Console
Type here to search
30°C Mostly cloudy
ENG
20:00
02-06-2024

```java
static void search(char txt[], char pat[])
{

    int m = pat.length;
    int n = txt.length;
    int badchar[] = new int[NO_OF_CHARS];
    badCharHeuristic(pat, m, badchar);
    int s = 0;
    while (s <= (n - m))
    {
        int j = m - 1;
        while (j >= 0 && pat[j] == txt[s + j])
            j--;
        if (j < 0)
        {
            System.out.println(
                    "Patterns occur at shift = " + s);
            s += (s + m < n) ? m - badchar[txt[s + m]]
                    : 1;
        }

        else
            s += max(1, j - badchar[txt[s + j]]);
    }
}
public static void main(String[] args)
{

    char txt[] = "ABAAABCD".toCharArray();
    char pat[] = "ABC".toCharArray();
    search(txt, pat);
}
}
```

```
<terminated> BoyerMoore [Java Application] C:\Users\user\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full
Patterns occur at shift = 4
```

The Boyer-Moore algorithm can outperform other string searching algorithms like the naive search and KMP in certain scenarios due to the following reasons:

1.Right-to-Left Character Comparison:

Boyer-Moore searches for the pattern from the right end (end of the text) towards the left. This allows it to potentially shift the entire pattern in one go if there's a mismatch at the rightmost character.
In contrast, the naive and KMP algorithms compare characters from the left, requiring them to move the pattern one position at a time for each mismatch, even if the mismatch occurs at the beginning of the pattern.

2.Bad Character Rule:

The bad character rule pre-computes the last occurrence of each character in the pattern. Upon a mismatch, the algorithm can directly shift the pattern based on the bad character rule, potentially skipping several comparisons.
This is more efficient than the naive approach, which needs to compare characters from the beginning again.

KMP also requires some comparisons before shifting based on the LPS table.

3.Good Suffix Rule (Optional):

The good suffix rule (optional but recommended) helps determine a larger shift value based on suffixes of the pattern. If a mismatch occurs, the good suffix rule can identify a potential suffix that already exists in the pattern and shift the pattern by the length of that suffix. This can further reduce unnecessary comparisons compared to the bad character rule alone.

Scenarios Where Boyer-Moore Shines:

- The Boyer-Moore algorithm is particularly effective when the pattern does not appear frequently in the text, or when there are many mismatches at the right end of the pattern.
- In such cases, the right-to-left comparison, bad character rule, and good suffix rule can lead to significant reductions in character comparisons compared to the naive search or KMP.