**NAME :-** Jangili Ravali

**EMAIL :-** jangiliravali9@gmail.com

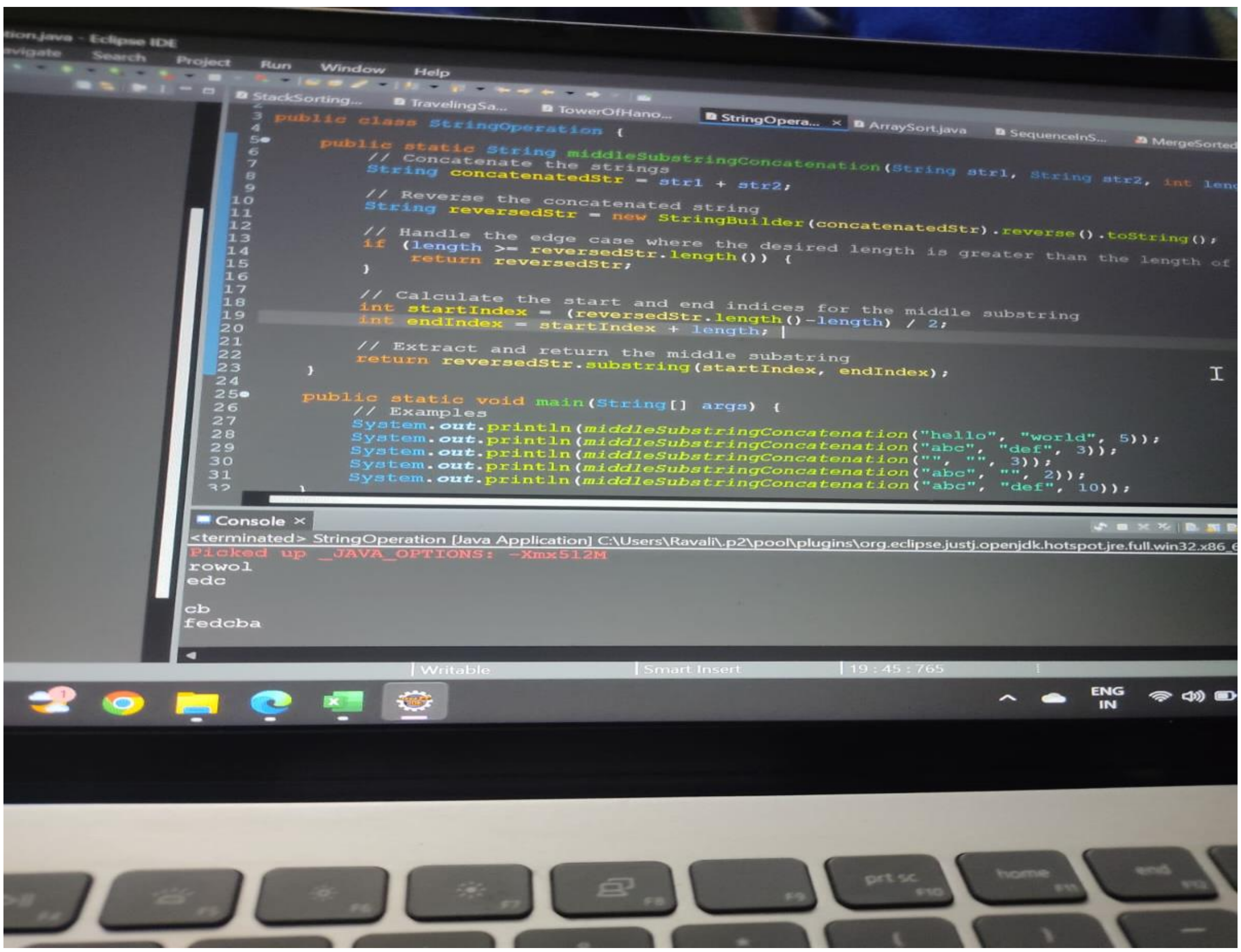
---

## 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.**



Explanation :-

**1.**Concatenation: The two input strings are concatenated using the + operator.

**2.**Reversal: The concatenated string is reversed using StringBuilder.

**3.** Edge Case Handling: If the desired substring length is greater than or equal to the length of the reversed string, the entire reversed string is returned.

**4.**Middle Substring Extraction:

**5.** The start index for the middle substring is calculated as (reversedStr.length() – length) / 2.

**6.** The end index is startIndex + length.

**7.** The middle substring is extracted using substring

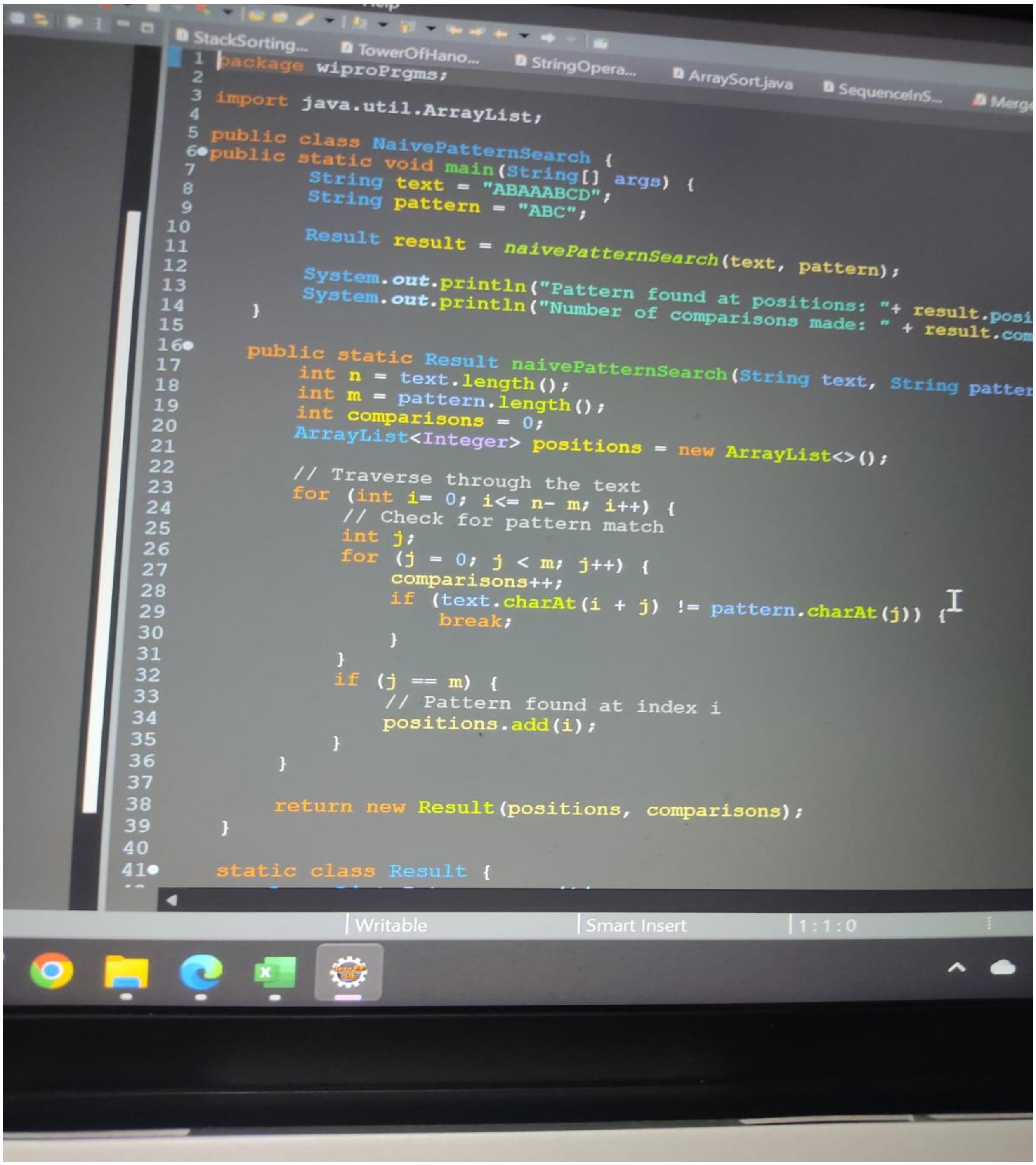
Edge Cases Handled :-

- Empty strings: If both strings are empty, the result is an empty string.

- Length larger than the concatenated string: If the specified length is greater than the length of the concatenated string, the entire reversed string is returned.

**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 wiproPrgms;

import java.util.ArrayList;

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

        Result result = naivePatternSearch(text, pattern);

        System.out.println("Pattern found at positions: "+ result.posi
        System.out.println("Number of comparisons made: " + result.com
    }

    public static Result naivePatternSearch(String text, String patter
        int n = text.length();
        int m = pattern.length();
        int comparisons = 0;
        ArrayList<Integer> positions = new ArrayList<>();

        // Traverse through the text
        for (int i= 0; i<= n- m; i++) {
            // Check for pattern match
            int j;
            for (j = 0; j < m; j++) {
                comparisons++;
                if (text.charAt(i + j) != pattern.charAt(j)) {
                    break;
                }
            }
            if (j == m) {
                // Pattern found at index i
                positions.add(i);
            }
        }

        return new Result(positions, comparisons);
    }

    static class Result {
```

```java
            // Check for pattern match
            int j;
            for (j = 0; j < m; j++) {
                comparisons++;
                if (text.charAt(i + j) != pattern.charAt(j)) {
                    break;
                }
            }
            if (j == m) {
                // Pattern found at index i
                positions.add(i);
            }
        }

        return new Result(positions, comparisons);
    }

    static class Result {
        ArrayList<Integer> positions;
        int comparisons;

        Result(ArrayList<Integer> positions, int comparisons) {
            this.positions = positions;
            this.comparisons = comparisons;
        }
    }
}
```

Console

```
<terminated> NaivePatternSearch [Java Application] C:\Users\Ravali\.p2\pool\plugins\org.eclipse.justj.open
Picked up _JAVA_OPTIONS: -Xmx512M
Pattern found at positions: [4]
Number of comparisons made: 12
```

Explanation

**1.** Initialization:

- n: Length of the text.
- m: Length of the pattern.
- comparisons: Counter to track the number of comparisons made.
- positions: List to store the starting indices where the pattern is found.

**2.**Outer Loop:

- The outer loop runs from 0 to n - m, ensuring the pattern can fit into the remaining part of the text.

**3.**Inner Loop:

- The inner loop checks each character of the pattern against the corresponding character in the text.
- If any character doesn't match, it breaks out of the inner loop.
- If the inner loop completes without a break, it means the pattern is found at the current index i.

**4.** Comparison Counting:

● The comparisons counter is incremented each time a character comparison is made.

**5.** Result Class:

● A nested Result class is used to store the positions where the pattern is found and the total number of comparisons made.

**6.** Output:

● The function prints the positions where the pattern occurs in the text and the total number of character comparisons performed during the search.

---

**Task 3: Implementing the KMP Algorithm**

**Code the Knuth-Morris-Pratt (KMP) algorithm in C# for pattern searching which preprocesses the pattern to reduce the number of comparisons. Explain how this preprocessing improves the search time compared to the naive approach.**

The Knuth-Morris-Pratt (KMP) algorithm improves the search time for pattern matching by preprocessing the pattern to build a partial match table (also known as the "prefix" table or "lps" array). This table is used to skip unnecessary comparisons in the search phase, thus reducing the overall number of comparisons.

Here's the step-by-step implementation of the KMP algorithm in C# along with an

explanation of how the preprocessing improves search time. KMP Algorithm in C#

using System;

```
public class KMPAlgorithm

{

   // Function to build the partial match table (lps array)     private

static int[] ComputeLPSArray(string pattern)

   {

       int length = 0;  // length of the previous longest prefix suffix

int i = 1;

       int M = pattern.Length;       int[]

lps = new int[M];         lps[0] = 0;  //

lps[0] is always 0


       // Loop calculates lps[i] for i = 1 to M-1

while (i < M)

       {

          if (pattern[i] == pattern[length])

          {

             length++;

lps[i] = length;

i++;

          }

else

          {

             if (length != 0)

             {

                length = lps[length - 1];
```

```
            }
else
            {
lps[i] = 0;
i++;
            }
          }
      }
      return lps;
  }

  // Function that implements KMP algorithm for pattern searching     public
static void KMPSearch(string text, string pattern)
  {
      int N = text.Length;
int M = pattern.Length;

      int[] lps = ComputeLPSArray(pattern);

      int i = 0;  // index for text
int j = 0;  // index for pattern
while (i < N)
      {
          if (pattern[j] == text[i])
```

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

            if (j == M)
            {
                Console.WriteLine("Found pattern at index " + (i - j));
                j = lps[j - 1];
            }
            else if (i < N && pattern[j] != text[i])
            {               if (j
!= 0)               {
j = lps[j - 1];
                }
else
{
i++;
                }
            }
        }
    }

    public static void Main()
    {
```

```
        string text = "ABABDABACDABABCABAB";
string pattern = "ABABCABAB";
        KMPSearch(text, pattern);
    }
}
```

Explanation of Preprocessing

(Partial Match Table)

The KMP algorithm preprocesses the pattern to build the lps (Longest Prefix which is also Suffix) array. This array is crucial for reducing the number of comparisons:

1. Building the LPS Array:

- The lps array for a given pattern contains values that tell us the longest proper prefix which is also a suffix for the pattern substring ending at each position.

- This preprocessing is done in O(M) time, where M is the length of the pattern.

2. Using the LPS Array:

- During the search phase, if there is a mismatch after j matches, instead of restarting the search from the beginning of the pattern, we use the lps array to skip some characters in the pattern itself.

- Specifically, if there is a mismatch at pattern[j], the next comparison should be with pattern[lps[j-1]] instead of pattern[0].

- This allows the algorithm to avoid redundant comparisons and ensures that every character in the text is compared at most once, leading to a search time of O(N), where N is the length of the text.

Improvement Over Naive Approach

The naive approach to pattern matching may require O(N*M) comparisons in the worst case because it attempts to match the pattern at every position in the text, restarting the comparison from the beginning of the pattern each time a mismatch occurs.

In contrast, the KMP algorithm preprocesses the pattern to avoid unnecessary comparisons. The lps array helps to skip over portions of the pattern that have already been matched, ensuring that the overall time complexity of the search is linear, O(N + M).

This preprocessing step significantly improves the efficiency of the pattern search, especially when dealing with large texts and patterns.

---

**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.**

```java
package wiproPrgms;

import java.util.List;

public class RabinKarp {
    public final static int d = 256;

    public static List<Integer> rabinKarp(String pattern, String text, int q) {
        List<Integer> result = new ArrayList<>();
        int m = pattern.length();
        int n = text.length();
        int i, j;
        int p = 0; // hash value for pattern
        int t = 0; // hash value for text
        int h = 1;

        // The value of h would be "pow(d, M-1)%q"
        for (i= 0; i < n-m ; i++) {
            h = (h * d) % q;
        }

        // Calculate the hash value of pattern and first window of text
        for (i = 0; i <m; i++) {
            p = (d * p + pattern.charAt(i)) % q;
            t= (d * t + text.charAt(i)) % q;
        }

        // Slide the pattern over text one by one
        for (i = 0; i<= n; i++) {
            // Check the hash values of current window of text and pattern
            if (p == t) {
                // Check for characters one by one
                for (j = 0; j < m; j++) {
                    if (text.charAt(i + j) != pattern.charAt(j)) {
                        break;
                    }
                }

                // if p == t and pattern[0…m-1] = text[I, i+1, …i+m-1]
                if (j == m) {
                    result.add(i);
```

```java
            if (j == m) {
                result.add(i);
            }
        }

        // Calculate hash value for next window of text
        if (i< n -m) {
            t = (d * (t- text.charAt(i) * h) + text.charAt(i + m)) % q;

            // We might get a negative value of t, converting it to positive
            if (t < 0) {
                t = (t + q);
            }
        }
    }

    return result;
}

public static void main(String[] args) {
    String text = "abracadabra";
    String pattern = "abra";
    int q = 101; // A prime number
    List<Integer> result = rabinKarp(pattern, text, q);

    System.out.println("Pattern found at indices: " + result);
}
}
```

Console

```
<terminated> RabinKarp [Java Application] C:\Users\Ravali\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_1
Picked up _JAVA_OPTIONS: -Xmx512M
Pattern found at indices: [0]
```

Explanation

**1.** Initialization:

- d is the number of characters in the input alphabet (here, 256 for ASCII).

- q is a prime number used for modulus operation to reduce the hash value size.

- p and t store the hash values for the pattern and the current text window, respectively.

- h is used for rolling hash calculations.

**2.**Preprocessing:

- The value of h is calculated as pow(d, M-1) % q.

- Initial hash values for the pattern and the first window of text are computed.

**3.**Pattern Search:

- The algorithm slides the pattern over the text and compares hash values.

- If hash values match, it performs character-by-character comparison to avoid false positives.

- Hash value for the next window of text is computed using the rolling hash formula.

**4.**Handling Hash Collisions:

- Collisions are handled by character comparison when hash values match.

- Using a large prime number q reduces the likelihood of collisions

Hash Collisions and Handling

In the Rabin-Karp algorithm, hash collisions can cause unnecessary character comparisons, slightly affecting performance. The use of a prime modulus q and an appropriate base d minimizes collisions. When collisions occur, character-by-character comparison ensures correctness.

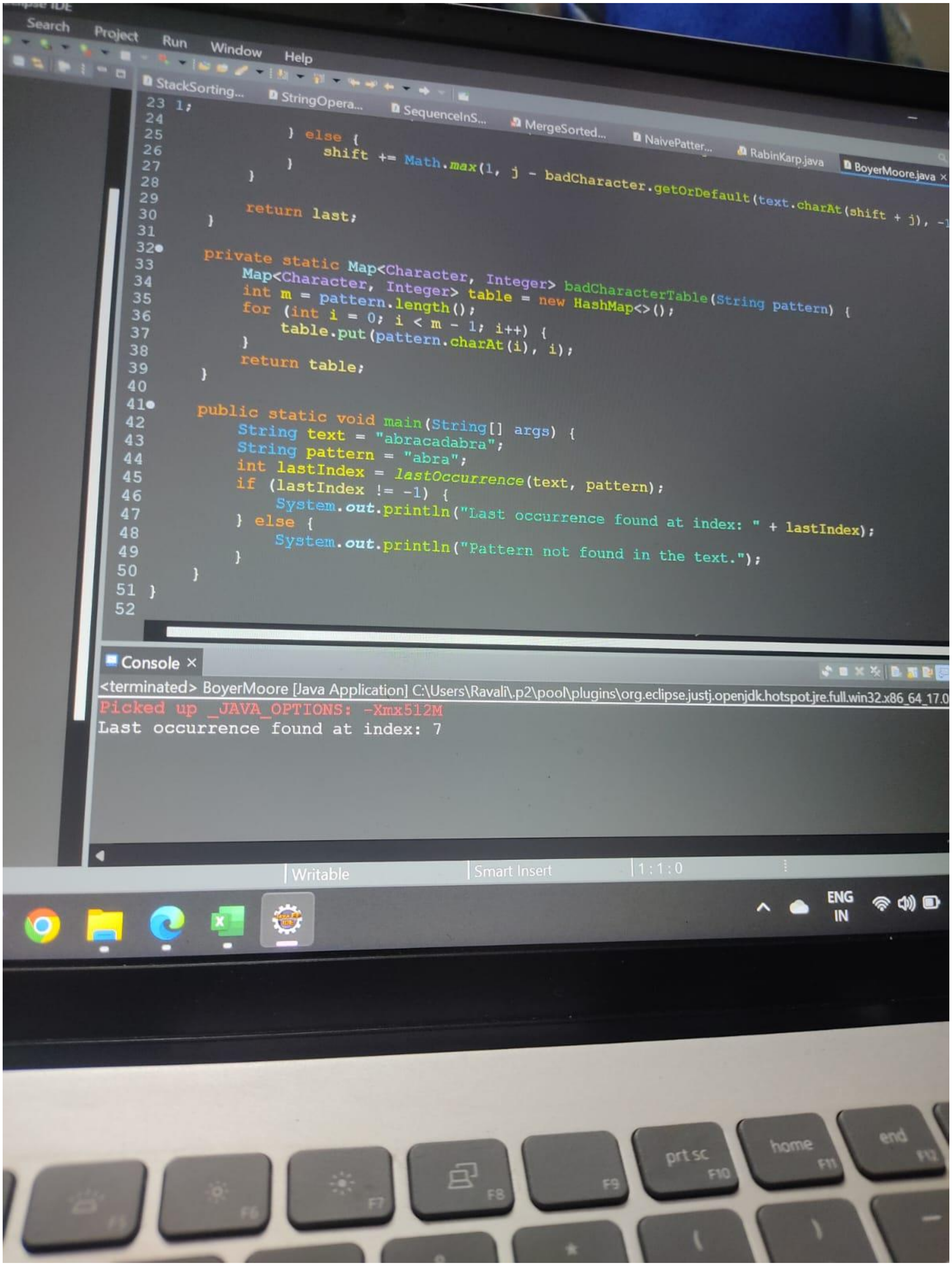
This implementation in Java efficiently searches for the pattern in the text, handling collisions appropriately to maintain performance.

---

**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.**

```java
package wiproPrgms;

import java.util.HashMap;
public class BoyerMoore {
    public static int lastOccurrence(String text, String pattern) {
        Map<Character, Integer> badCharacter = badCharacterTable(pattern);
        int n = text.length();
        int m = pattern.length();
        int shift = 0;
        int last = -1;

        while (shift <= n - m) {
            int j = m - 1;

            while (j >= 0 && pattern.charAt(j) == text.charAt(shift + j)) {
                j--;
            }

            if (j < 0) {
                last = shift;
                shift += (shift + m < n) ? m - badCharacter.getOrDefault(text.charAt(shift + m), -1)
1;
            } else {
                shift += Math.max(1, j - badCharacter.getOrDefault(text.charAt(shift + j), -1));
            }
        }

        return last;
    }

    private static Map<Character, Integer> badCharacterTable(String pattern) {
        Map<Character, Integer> table = new HashMap<>();
        int m = pattern.length();
        for (int i = 0; i < m - 1; i++) {
            table.put(pattern.charAt(i), i);
        }
        return table;
    }

    public static void main(String[] args) {
        String text = "abracadabra";
```


Search
Project
Run
Window
Help
StackSorting...
StringOpera...
SequenceInS...
MergeSorted...
NaivePatter...
RabinKarp.java
BoyerMoore.java
1;
} else {
shift += Math.max(1, j - badCharacter.getOrDefault(text.charAt(shift + j), -1
}
}
return last;
}
private static Map<Character, Integer> badCharacterTable(String pattern) {
Map<Character, Integer> table = new HashMap<>();
int m = pattern.length();
for (int i = 0; i < m - 1; i++) {
table.put(pattern.charAt(i), i);
}
return table;
}
public static void main(String[] args) {
String text = "abracadabra";
String pattern = "abra";
int lastIndex = lastOccurrence(text, pattern);
if (lastIndex != -1) {
System.out.println("Last occurrence found at index: " + lastIndex);
} else {
System.out.println("Pattern not found in the text.");
}
}
}
Console
<terminated> BoyerMoore [Java Application] C:\Users\Ravali\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0
Picked up _JAVA_OPTIONS: -Xmx512M
Last occurrence found at index: 7
Writable
Smart Insert
1 : 1 : 0
ENG
IN