Day 11: Manisha Assignment

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

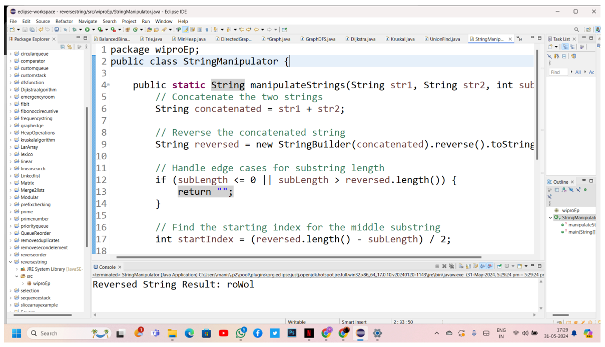
1. **StringManipulator Class:**
   - manipulateStrings Method:
     - Concatenate: Combines `str1` and `str2`.
     - Reverse: Reverses the concatenated string.
     - Edge Cases: Checks if the requested substring length is valid.
     - Middle Substring: Calculates the starting index for the middle substring and extracts it.

2. **Main Method:**
   - Test Data: Defines sample strings `str1` and `str2`, and the desired substring length `subLength`.
   - Method Call: Calls `manipulateStrings` with the test data and prints the result.

Running the Program in Eclipse

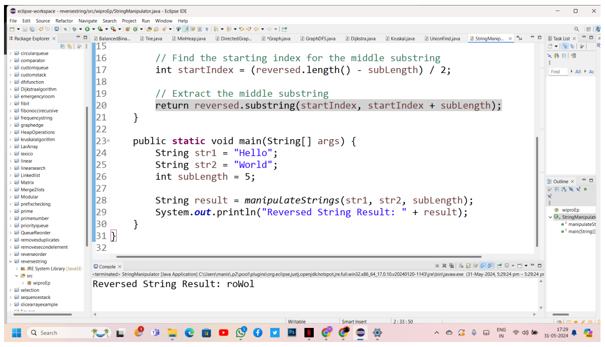
1. **Create a New Java Project**: In Eclipse, go to `File -> New -> Java Project`. Name the project, e.g., `StringManipulationExample`.
2. **Create a New Java Class**: Right-click on the `src` folder in your project, then go to `New -> Class`. Name the class `StringManipulator` and include the `public static void main(String[] args)` method.
3. **Copy and Paste the Code:** Copy the provided code and paste it into your `StringManipulator` class.
4. **Run the Program**: Right-click on your `StringManipulator` class in the Eclipse project explorer, then select `Run As -> Java Application`.

Here output as follows:



```java
package wiproEp;
public class StringManipulator {

    public static String manipulateStrings(String str1, String str2, int sub
        // Concatenate the two strings
        String concatenated = str1 + str2;

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

        // Handle edge cases for substring length
        if (subLength <= 0 || subLength > reversed.length()) {
            return "";
        }

        // Find the starting index for the middle substring
        int startIndex = (reversed.length() - subLength) / 2;

```

```
<terminated> StringManipulator [Java Application] C:\Users\manis\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe (31-May-2024, 5:29:24 pm – 5:29:24 pr
Reversed String Result: roWol
```

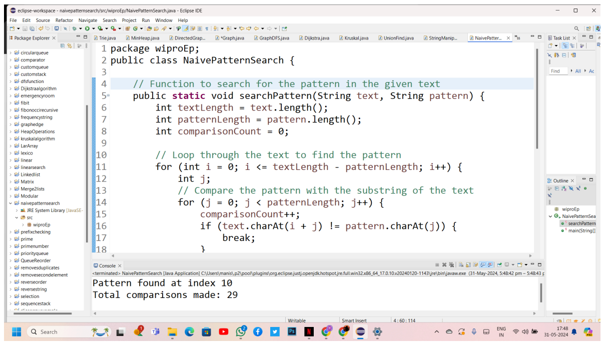
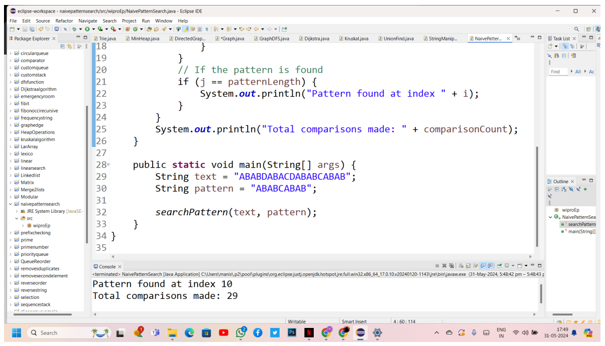


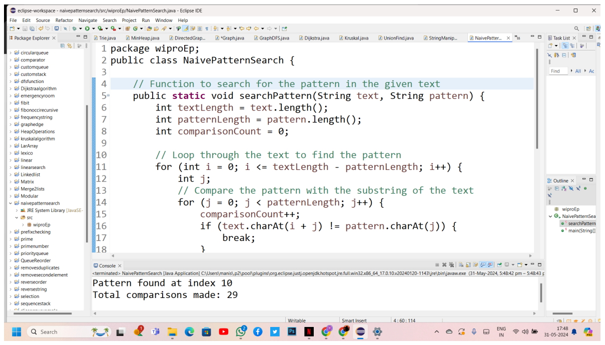
```java

        // Find the starting index for the middle substring
        int startIndex = (reversed.length() - subLength) / 2;

        // Extract the middle substring
        return reversed.substring(startIndex, startIndex + subLength);
    }

    public static void main(String[] args) {
        String str1 = "Hello";
        String str2 = "World";
        int subLength = 5;

        String result = manipulateStrings(str1, str2, subLength);
        System.out.println("Reversed String Result: " + result);
    }
}

```

```
<terminated> StringManipulator [Java Application] C:\Users\manis\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe (31-May-2024, 5:29:24 pm – 5:29:24 pr
Reversed String Result: roWol
```

Task 2: Naive Pattern Search

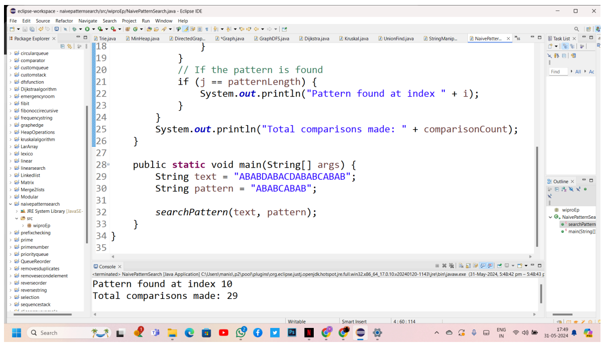
Implement the naive pattern searching algorithm to find all occurrences of a pattern within a given text string. Count the number of comparisons made during the search to evaluate the efficiency of the algorithm.

Explanation	**Here Output as follows:**
1. **NaivePatternSearch Class:** - searchPattern Method: - Takes two parameters: `text` (the string to be searched) and `pattern` (the pattern to search for). - Initializes `textLength` and `patternLength` to store the lengths of the text and the pattern, respectively. - Initializes `comparisonCount` to zero to count the number of comparisons made. - Uses a nested loop to compare the pattern with substrings of the text: - The outer loop iterates through the text from index `0` to `textLength - patternLength`. - The inner loop compares each character of the pattern with the corresponding character in the text and increments the `comparisonCount` for each comparison. - If the entire pattern matches the substring of the text, it prints the starting index. - After the outer loop completes, it prints the total number of comparisons made. 2. **Main Method:** - Defines the sample `text` and `pattern`. - Calls the `searchPattern` method with the sample data.	 



```
package wiproEp;
public class NaivePatternSearch {

    // Function to search for the pattern in the given text
    public static void searchPattern(String text, String pattern) {
        int textLength = text.length();
        int patternLength = pattern.length();
        int comparisonCount = 0;

        // Loop through the text to find the pattern
        for (int i = 0; i <= textLength - patternLength; i++) {
            int j;
            // Compare the pattern with the substring of the text
            for (j = 0; j < patternLength; j++) {
                comparisonCount++;
                if (text.charAt(i + j) != pattern.charAt(j)) {
                    break;
                }
```

```
<terminated> NaivePatternSearch [Java Application] C:\Users\manis\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe (31-May-2024, 5:48:42 pm – 5:48:43 p
Pattern found at index 10
Total comparisons made: 29
```



```
                }
            }
            // If the pattern is found
            if (j == patternLength) {
                System.out.println("Pattern found at index " + i);
            }
        }
        System.out.println("Total comparisons made: " + comparisonCount);
    }

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

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

```
<terminated> NaivePatternSearch [Java Application] C:\Users\manis\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe (31-May-2024, 5:48:42 pm – 5:48:43 p
Pattern found at index 10
Total comparisons made: 29
```

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.

Importance of KMP Algorithm

1. Efficiency: The KMP algorithm improves the efficiency of pattern searching by reducing the number of comparisons. Unlike the naive approach, which may require backtracking in the text, KMP uses the information from the pattern itself to avoid unnecessary comparisons.

2. LPS Array: The pre-processing step in KMP, where the longest prefix suffix (LPS) array is created

3. Applications: The KMP algorithm is widely used in various applications, including text editors, search engines, and bioinformatics, where efficient pattern matching is crucial.

Explanation

1. KMPAlgorithm Class:

- KMPSearch Method:

- Takes `pattern` and `text` as inputs.

- Sets lengths for both.

- Creates an `lps` array to track longest prefix suffix values.

- Fills `lps` array using `computeLPSArray`.

- Uses two indices (`i` for text and `j` for pattern).

- If characters match, increments both indices.

- If `j` reaches pattern length, prints the match index and resets `j` using `lps`.

- If characters don't match and `j` is non-zero, resets `j` using `lps`; otherwise, increments `i`.

- computeLPSArray Method:

- Takes `pattern`, its length, and the `lps` array.

- Initializes `length` to 0 and starts `i` at 1.

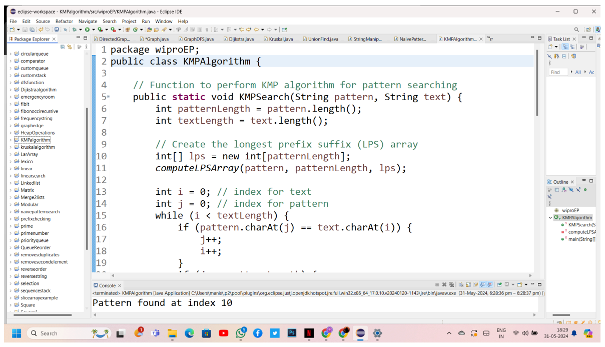
- Matches characters, updates `lps`.

- If no match and `length` is non-zero, resets `length` using `lps`; otherwise, sets `lps[i]` to 0.

2. Main Method:

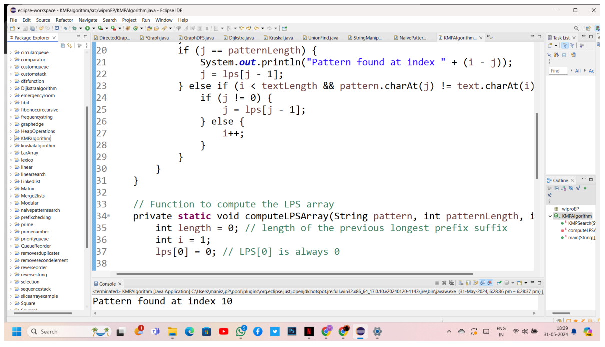
- Defines sample `text` and `pattern`.

- Calls `KMPSearch` with the sample data.



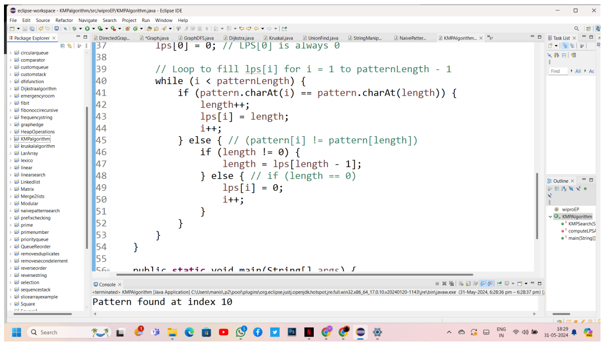
```java
package wiproEP;
public class KMPAlgorithm {

    // Function to perform KMP algorithm for pattern searching
    public static void KMPSearch(String pattern, String text) {
        int patternLength = pattern.length();
        int textLength = text.length();

        // Create the longest prefix suffix (LPS) array
        int[] lps = new int[patternLength];
        computeLPSArray(pattern, patternLength, lps);

        int i = 0; // index for text
        int j = 0; // index for pattern
        while (i < textLength) {
            if (pattern.charAt(j) == text.charAt(i)) {
                j++;
                i++;
            }
```

```
Pattern found at index 10
```



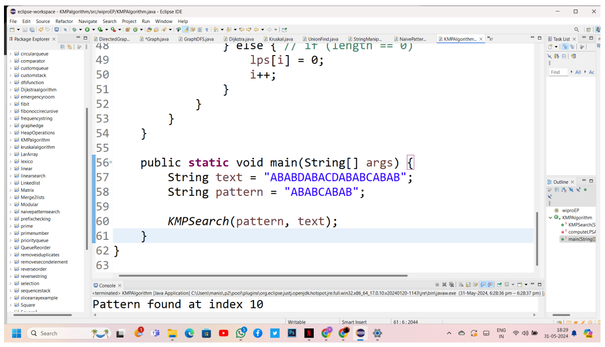
```java
            if (j == patternLength) {
                System.out.println("Pattern found at index " + (i - j));
                j = lps[j - 1];
            } else if (i < textLength && pattern.charAt(j) != text.charAt(i)
                if (j != 0) {
                    j = lps[j - 1];
                } else {
                    i++;
                }
            }
        }
    }

    // Function to compute the LPS array
    private static void computeLPSArray(String pattern, int patternLength, i
        int length = 0; // length of the previous longest prefix suffix
        int i = 1;
        lps[0] = 0; // LPS[0] is always 0

```

```
Pattern found at index 10
```



```java
        lps[0] = 0; // LPS[0] is always 0

        // Loop to fill lps[i] for i = 1 to patternLength - 1
        while (i < patternLength) {
            if (pattern.charAt(i) == pattern.charAt(length)) {
                length++;
                lps[i] = length;
                i++;
            } else { // (pattern[i] != pattern[length])
                if (length != 0) {
                    length = lps[length - 1];
                } else { // if (length == 0)
                    lps[i] = 0;
                    i++;
                }
            }
        }
    }

```

```
Pattern found at index 10
```



```java
                } else { // if (length == 0)
                    lps[i] = 0;
                    i++;
                }
            }
        }
    }

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

        KMPSearch(pattern, text);
    }
}

```

```
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.

Explanation:

1. **RabinKarp Class:**
   - `search` Method:
     - Takes `pattern`, `text`, and a prime number `q`.
     - Calculates hash values for pattern and the first window of text.
     - Slides the pattern over the text to compare hash values.
     - If hash values match, it checks each character for a complete match.
     - Uses a rolling hash to update the hash value for the next window of text.
   - `main` Method:
     - Defines sample `text` and `pattern`.
     - Calls the `search` method with the sample data and a prime number.

Output:

Pattern found at index 0

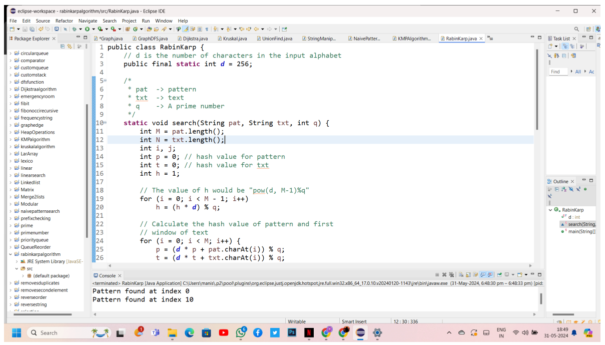
Pattern found at index 10

Importance of Rabin-Karp Algorithm

The Rabin-Karp algorithm is efficient for multiple pattern searches in a single text. It uses hash values to quickly filter out non-matching substrings, reducing unnecessary character comparisons.

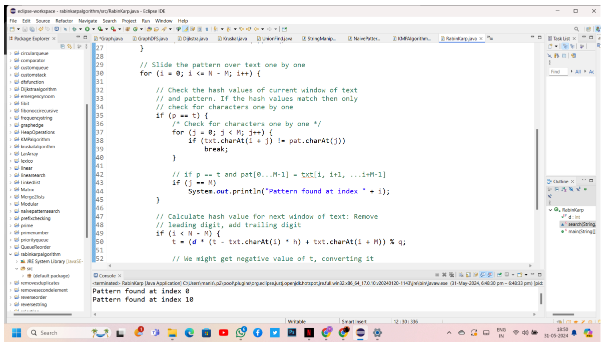
Impact of Hash Collisions

Hash collisions occur when different substrings produce the same hash value. Collisions lead to extra character comparisons, slightly reducing performance. Using a good hash function and a large prime number reduces collision probability. If a collision is detected, the algorithm compares the actual substring to confirm a match.



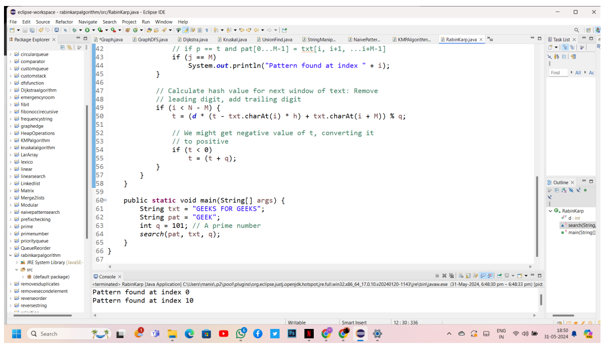
```java
public class RabinKarp {
    // d is the number of characters in the input alphabet
    public final static int d = 256;

    /*
     * pat  -> pattern
     * txt  -> text
     * q    -> A prime number
     */
    static void search(String pat, String txt, int q) {
        int M = pat.length();
        int N = txt.length();
        int i, j;
        int p = 0; // hash value for pattern
        int t = 0; // hash value for txt
        int h = 1;

        // The value of h would be "pow(d, M-1)%q"
        for (i = 0; i < M - 1; 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 + pat.charAt(i)) % q;
            t = (d * t + txt.charAt(i)) % q;
```

```
<terminated> RabinKarp [Java Application] C:\Users\manis\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe
Pattern found at index 0
Pattern found at index 10
```



```java
        }

        // Slide the pattern over text one by one
        for (i = 0; i <= N - M; i++) {

            // Check the hash values of current window of text
            // and pattern. If the hash values match then only
            // check for characters one by one
            if (p == t) {
                /* Check for characters one by one */
                for (j = 0; j < M; j++) {
                    if (txt.charAt(i + j) != pat.charAt(j))
                        break;
                }

                // if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
                if (j == M)
                    System.out.println("Pattern found at index " + i);
            }

            // Calculate hash value for next window of text: Remove
            // leading digit, add trailing digit
            if (i < N - M) {
                t = (d * (t - txt.charAt(i) * h) + txt.charAt(i + M)) % q;

                // We might get negative value of t, converting it
```

```
<terminated> RabinKarp [Java Application] C:\Users\manis\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe
Pattern found at index 0
Pattern found at index 10
```



```java
                // if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
                if (j == M)
                    System.out.println("Pattern found at index " + i);
            }

            // Calculate hash value for next window of text: Remove
            // leading digit, add trailing digit
            if (i < N - M) {
                t = (d * (t - txt.charAt(i) * h) + txt.charAt(i + M)) % q;

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

    public static void main(String[] args) {
        String txt = "GEEKS FOR GEEKS";
        String pat = "GEEK";
        int q = 101; // A prime number
        search(pat, txt, q);
    }
}
```

```
<terminated> RabinKarp [Java Application] C:\Users\manis\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe
Pattern found at index 0
Pattern found at index 10
```

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.

Importance of Boyer-Moore Algorithm

The Boyer-Moore algorithm can outperform other string matching algorithms like the naive algorithm.

1. **Efficiency with Larger Alphabets:** The bad character heuristic can skip more characters when the alphabet size is large, reducing the number of comparisons.
2. **Pattern Length:** For longer patterns, the good suffix heuristic can lead to larger skips, making the search faster.
3. **Preprocessing Time:** The preprocessing time of Boyer-Moore is linear relative to the length of the pattern, making it efficient for repeated searches with the same pattern.

Example

Let's break down the example provided:

- Text: `**"ABAAABCDABC"**`
- Pattern: `**"ABC"**`
- The algorithm will:
  1. Create a bad character table for the pattern `**"ABC"**`.
  2. Start comparing from the end of the text.
  3. Use the bad character table to skip sections of the text that cannot contain the pattern.
  4. Find the last occurrence of the pattern `"ABC"` in the text at index 8.

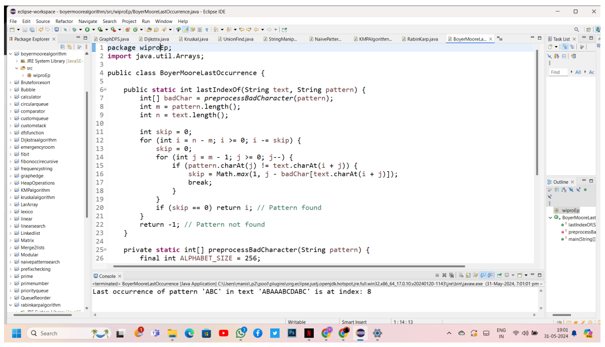
The output confirms that the last occurrence of `"ABC"` in `**"ABAAABCDABC"**` is at index 8.

Output

Last occurrence of pattern 'ABC' in text **'ABAAABCDABC'** is at index: 8

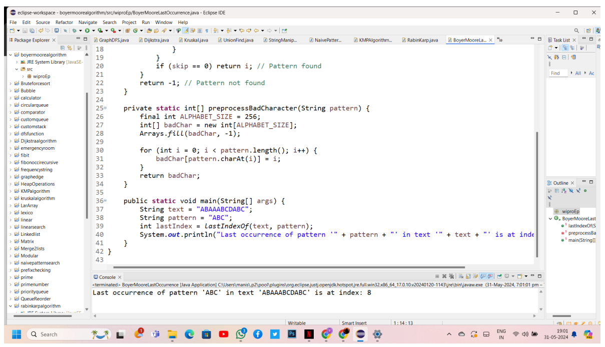
This demonstrates how the Boyer-Moore algorithm efficiently finds the last occurrence of a pattern in a given text by skipping unnecessary comparisons.

Here Output as follows



```java
package wiproEp;
import java.util.Arrays;

public class BoyerMooreLastOccurrence {

    public static int lastIndexOf(String text, String pattern) {
        int[] badChar = preprocessBadCharacter(pattern);
        int m = pattern.length();
        int n = text.length();

        int skip = 0;
        for (int i = n - m; i >= 0; i -= skip) {
            skip = 0;
            for (int j = m - 1; j >= 0; j--) {
                if (pattern.charAt(j) != text.charAt(i + j)) {
                    skip = Math.max(1, j - badChar[text.charAt(i + j)]);
                    break;
                }
            }
            if (skip == 0) return i; // Pattern found
        }
        return -1; // Pattern not found
    }

    private static int[] preprocessBadCharacter(String pattern) {
        final int ALPHABET_SIZE = 256;
```

```
<terminated> BoyerMooreLastOccurrence [Java Application] C:\Users\manis\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe (31-May-2024, 7:01:01 pm –
Last occurrence of pattern 'ABC' in text 'ABAAABCDABC' is at index: 8
```



```java
            }
        }
        if (skip == 0) return i; // Pattern found
    }
    return -1; // Pattern not found
}

    private static int[] preprocessBadCharacter(String pattern) {
        final int ALPHABET_SIZE = 256;
        int[] badChar = new int[ALPHABET_SIZE];
        Arrays.fill(badChar, -1);

        for (int i = 0; i < pattern.length(); i++) {
            badChar[pattern.charAt(i)] = i;
        }
        return badChar;
    }

    public static void main(String[] args) {
        String text = "ABAAABCDABC";
        String pattern = "ABC";
        int lastIndex = lastIndexOf(text, pattern);
        System.out.println("Last occurrence of pattern '" + pattern + "' in text '" + text + "' is at inde
    }
}
```

```
<terminated> BoyerMooreLastOccurrence [Java Application] C:\Users\manis\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\bin\javaw.exe (31-May-2024, 7:01:01 pm –
Last occurrence of pattern 'ABC' in text 'ABAAABCDABC' is at index: 8
```