

Name: Swarnali Ghosh

Java Pre-Skilling Training Session

Assignment -5.4 (Task1 to Task 5)

Module-5 (DAY 11)

Mail-id: swarnalighosh666@gmail.com

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.

SOLUTION:

→ Java implementation for String Operations-

```
package com.wipro.swarnali;

public class StringOperations {

    public static String getMiddleSubstring(String str1, String
str2, int length) {
        // Concatenate the two strings
        String concatenated = str1.concat(str2);

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

```

        // Calculating the start index of the middle substring
        int startIndex = (reversed.length() - length) / 2;

        // Edge case: if the substring length is larger than the
concatenated string,
        // return the concatenated string itself
        if (length >= concatenated.length()) {
            return concatenated;
        }

        // Extract the middle substring
        String middleSubstring = reversed.substring(startIndex,
startIndex + length);
        return middleSubstring;
    }

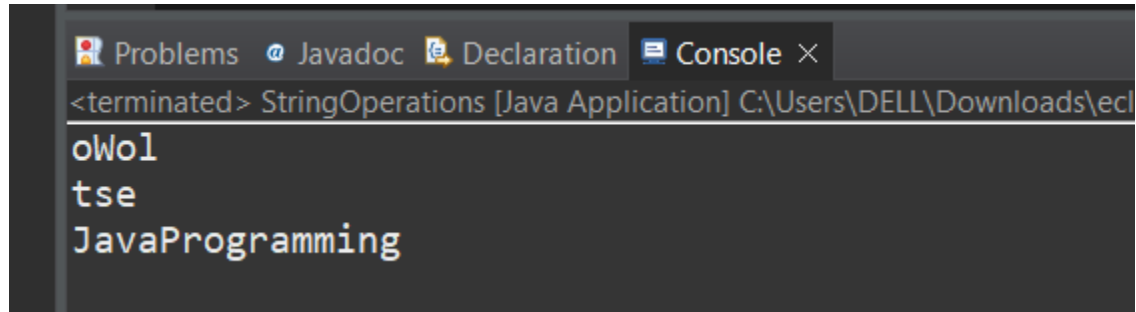
    public static void main(String[] args) {
        // Example usage
        String str1 = "Hello";
        String str2 = "World";
        int length = 4;
        System.out.println(getMiddleSubstring(str1, str2,
length)); // Output: "oWo1"

        // Edge case: Empty string
        String emptyStr1 = "";
        String emptyStr2 = "Test";
        int emptyLength = 3;
        System.out.println(getMiddleSubstring(emptyStr1,
emptyStr2, emptyLength)); // Output: "tse"

        // Edge case: Substring length larger than concatenated
string
        String testStr1 = "Java";
        String testStr2 = "Programming";
        int largeLength = 20;
        System.out.println(getMiddleSubstring(testStr1, testStr2,
largeLength)); // Output: "JavaProgramming"
    }
}

```

Output: -



```
<terminated> StringOperations [Java Application] C:\Users\DELL\Downloads\ec  
oWol  
tse  
JavaProgramming
```

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.

SOLUTION:

→Java Implementation the naive pattern searching algorithm to find all occurrences of a pattern within a given text string.

```
package com.wipro.swarnali;  
  
public class NaiveStringSearch {  
  
    public static void main(String[] args) {  
        String text = "I LOVE CATS";  
        String pattern = "CATS";  
  
        System.out.println("\nText: " + text);  
        System.out.println("Pattern: " + pattern);  
    }  
}
```

```

        System.out.println("Occurrences found at positions: ");

        int comparisons = search(text, pattern);

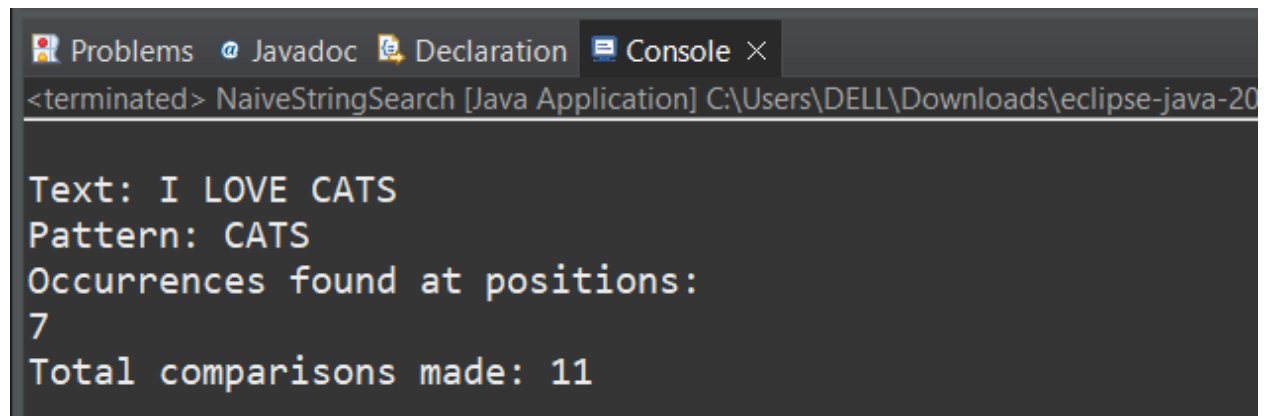
        System.out.println("\nTotal comparisons made: " +
comparisons);
    }

    public static int search(String text, String pattern) {
        int n = text.length();
        int m = pattern.length();
        int comparisons = 0;

        for (int i = 0; i <= n - m; i++) {
            int j;
            for (j = 0; j < m; j++) {
                comparisons++;
                if (text.charAt(i + j) != pattern.charAt(j))
                    break;
            }
            if (j == m) {
                System.out.print(i + " ");
            }
        }
        return comparisons;
    }
}

```

Output: -



The screenshot shows the Eclipse IDE's Console window. The title bar includes 'Problems', 'Javadoc', 'Declaration', and 'Console'. The console output for the 'NaiveStringSearch' Java application is as follows:

```

<terminated> NaiveStringSearch [Java Application] C:\Users\DELL\Downloads\eclipse-java-20
Text: I LOVE CATS
Pattern: CATS
Occurrences found at positions:
7
Total comparisons made: 11

```

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.

SOLUTION:

The Knuth-Morris-Pratt (KMP) algorithm pre-processes the pattern string to create an auxiliary array called the "longest proper prefix which is also a suffix" array, commonly known as LPS array.

This array helps in reducing the number of comparisons by efficiently skipping characters of the text when a mismatch occurs.

Java code Implementation: -

```
package com.wipro.swarnali;

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

public class KMPAlgorithm {

    public static List<Integer> searchPattern(String text, String
pattern) {
        List<Integer> matches = new ArrayList<>();
        int[] lps = computeLPSArray(pattern);

        int i = 0; // index for text
        int j = 0; // index for pattern
    }
}
```

```

        while (i < text.length()) {
            if (pattern.charAt(j) == text.charAt(i)) {
                j++;
                i++;
            }
            if (j == pattern.length()) {
                matches.add(i - j);
                j = lps[j - 1];
            } else if (i < text.length() && pattern.charAt(j) !=
text.charAt(i)) {
                if (j != 0)
                    j = lps[j - 1];
                else
                    i = i + 1;
            }
        }
        return matches;
    }

    private static int[] computeLPSArray(String pattern) {
        int[] lps = new int[pattern.length()];
        int len = 0; // length of the previous longest prefix
suffix

        int i = 1;
        while (i < pattern.length()) {
            if (pattern.charAt(i) == pattern.charAt(len)) {
                len++;
                lps[i] = 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 = "I LOVE CATS";
        String pattern = "CATS";
    }

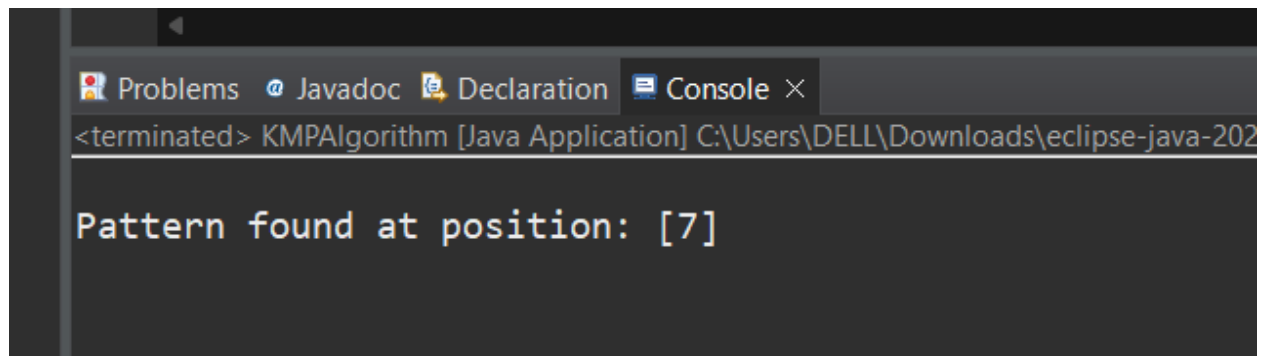
```

```

        List<Integer> matches = searchPattern(text, pattern);
        if (matches.isEmpty()) {
            System.out.println("Pattern not found in the text.");
        } else {
            System.out.println("\nPattern found at position: " +
matches);
        }
    }
}

```

Output: -



```

<terminated> KMPAlgorithm [Java Application] C:\Users\DELL\Downloads\eclipse-java-202
Pattern found at position: [7]

```

The Pre-Processing steps in the KMP algorithm improves the search time compared to the naive approach:

Naive Approach:

In the naive approach, we compare each character of the pattern with each character of the text in a brute-force manner. Whenever a mismatch occurs, we shift the pattern one position to the right and restart the comparison.

This approach leads to redundant comparisons because we might revisit characters in the text that have already been compared.

KMP Algorithm:

1. Pre-processing (Compute LPS Array):

- The KMP algorithm preprocesses the pattern to construct the Longest Prefix Suffix (lps) array.
- The lps array stores the length of the longest proper prefix of the pattern that matches a proper suffix.
- This pre-processing step is done only once for the pattern and takes $O(m)$ time, where m is the length of the pattern.

2. Search Phase:

- During the search phase, instead of starting over from the beginning of the pattern when a mismatch occurs, KMP utilizes the lps array to avoid redundant comparisons.
- When a mismatch happens at position j in the pattern, instead of shifting the pattern one position to the right and restarting the comparison from the beginning of the pattern, KMP jumps j positions ahead using the information stored in the lps array.
- By utilizing the lps array, KMP ensures that no character in the text is compared more than once with any character in the pattern.
- This allows the algorithm to skip unnecessary comparisons, leading to a significant reduction in search time.

Comparison:

• Naive Approach:

- Time complexity: $O(n * m)$ in the worst case, where n is the length of the text and m is the length of the pattern.
- Redundant comparisons are made, especially when encountering mismatches.

• KMP Algorithm:

- Time complexity: $O(n + m)$ in the worst case, where n is the length of the text and m is the length of the pattern.
- Avoids redundant comparisons by utilizing the information stored in the lps array.
- Achieves a linear time complexity due to the efficient search strategy.

Conclusion:

The pre-processing step in the KMP algorithm, which constructs the lps array, significantly improves search time compared to the naive approach by eliminating redundant comparisons.

This efficiency becomes particularly pronounced for larger texts and patterns, making KMP a preferred choice for pattern searching tasks.

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.

SOLUTION:

```
package com.wipro.swarnali;

public class RabinKarp_Hash {

    public final static int d = 10;

    static void search(String pattern, String txt, int q) {
        int m = pattern.length();
        int n = txt.length();
        int i, j;
        int p = 0;
        int t = 0;
        int h = 1;

        for (i = 0; i < m - 1; i++)
            h = (h * d) % q;

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

        // Find the match
```

```

        for (i = 0; i <= n - m; i++) {
            if (p == t) {
                for (j = 0; j < m; j++) {
                    if (txt.charAt(i + j) != pattern.charAt(j))
                        break;
                }

                if (j == m)
                    System.out.println("Pattern is found at position: " +
(i + 1));
            }

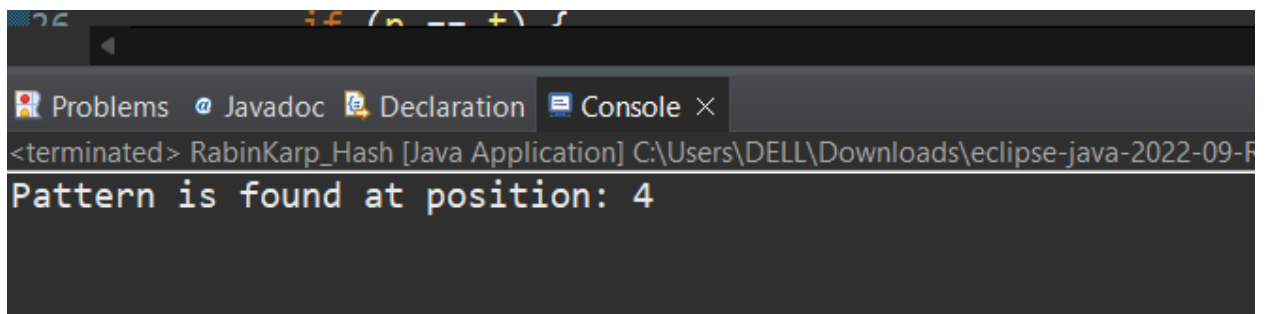
            if (i < n - m) {
                t = (d * (t - txt.charAt(i) * h) + txt.charAt(i + m)) %
q;

                if (t < 0)
                    t = (t + q);
            }
        }
    }

    public static void main(String[] args) {
        String txt = "ABCCDDAEFG";
        String pattern = "CDD";
        int q = 13;
        search(pattern, txt, q);
    }
}

```

Output: -



The screenshot shows an IDE interface with a 'Console' tab selected. The console output displays the message 'Pattern is found at position: 4'. Above the console, the file path 'C:\Users\DELL\Downloads\eclipse-java-2022-09-F' is partially visible.

Hash collisions can have a significant impact on the performance and accuracy of the Rabin-Karp algorithm. Here's how they affect the algorithm and how to handle them:

1. Impact on Performance:

- **Increased Time Complexity:** Hash collisions can lead to additional comparisons between strings to verify the match. This increases the time complexity of the algorithm, especially if there are many collisions.
- **Degraded Efficiency:** Collisions can result in unnecessary comparisons, slowing down the algorithm's overall efficiency, particularly for large texts or patterns.

2. Impact on Accuracy:

- **False Positives:** Hash collisions may incorrectly suggest matches where none exist, leading to false positives. This affects the accuracy of the algorithm's results.
- **Incorrect Matches:** Due to collisions, the algorithm may report matches that are not genuine, leading to incorrect outputs.

3. Handling Hash Collisions:

- **Use of Prime Numbers:** As seen in the implementations, choosing a prime number as the base for hashing helps reduce the likelihood of collisions. Prime numbers reduce common factors, decreasing the chance of collisions.
- **Fallback Mechanism:** If a hash collision occurs, a fallback mechanism is needed to verify the match accurately. This involves comparing the actual strings character by character to confirm the match. While slower, this ensures correctness.
- **Dynamic Hashing:** Implementing a dynamic hashing strategy can mitigate collisions by adjusting the hashing mechanism based on the characteristics of the input strings. This may involve dynamically adjusting the prime number used for hashing or employing other techniques to reduce collisions.
- **Collision Resolution:** In some cases, collision resolution techniques like separate chaining or open addressing can be applied to handle collisions efficiently. However, these techniques are more commonly used in hash table implementations rather than in the context of string searching algorithms like Rabin-Karp.

In summary, while hash collisions can impact the performance and accuracy of the Rabin-Karp algorithm, using prime numbers for hashing and implementing fallback mechanisms can help mitigate these issues. Careful consideration of hash function parameters and dynamic hashing strategies can further enhance the algorithm's effectiveness in handling collisions.

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.

SOLUTION:

```
package com.wipro.swarnali;

import java.util.HashMap;
import java.util.Map;

public class BoyerMoore_algo {

    public static int lastIndexOf(String text, String pattern) {
        if (pattern.length() == 0) {
            return -1;
        }

        Map<Character, Integer> badCharacterTable =
buildBadCharacterTable(pattern);
        int m = pattern.length();
        int n = text.length();
        int i = n - m;

        while (i >= 0) {
            int j = m - 1;
            while (j >= 0 && pattern.charAt(j) == text.charAt(i +
j)) {
                j--;
            }
            if (j < 0) {
                return i;
            } else {
                char badChar = text.charAt(i + j);
                int shift =
badCharacterTable.getOrDefault(badChar, -1);
                i -= Math.max(1, j - shift);
            }
        }
    }
}
```

```

    }
}

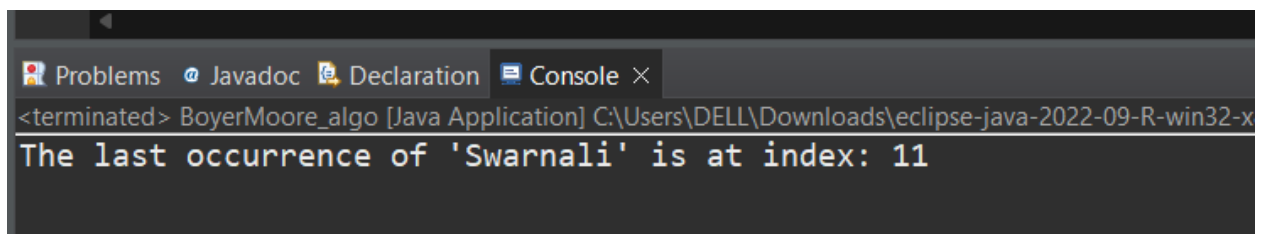
return -1;
}

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

public static void main(String[] args) {
    String text = "My name is Swarnali";
    String pattern = "Swarnali";
    int index = lastIndexOf(text, pattern);
    System.out.println("The last occurrence of '" + pattern +
    "' is at index: " + index);
}
}

```

Output: -



The screenshot shows an IDE window with a tab labeled 'Console'. The console output is as follows:

```

<terminated> BoyerMoore_algo [Java Application] C:\Users\DELL\Downloads\eclipse-java-2022-09-R-win32-x
The last occurrence of 'Swarnali' is at index: 11

```

Boyer-Moore algorithm can outperform other string search algorithms in certain scenarios:

1. **Character Comparison Optimization:** Boyer-Moore algorithm skips characters in the text based on a precomputed table (the bad character table in this case). This allows it to skip more characters and avoid unnecessary character comparisons, especially when there are mismatches between the pattern and the text.

2. **Performance on Large Texts:** Boyer-Moore tends to perform better on larger texts due to its ability to skip characters efficiently. This is particularly useful when the pattern being searched for occurs infrequently within the text.
3. **Worst-Case Complexity:** While the worst-case time complexity of Boyer-Moore is still $O(nm)$, where n is the length of the text and m is the length of the pattern, in practice, it often outperforms other algorithms like naive string matching and KMP (Knuth-Morris-Pratt) algorithm, especially for certain types of patterns.
4. **Adaptive Shifts:** Boyer-Moore adapts its shift strategy based on the character mismatch encountered, making it more flexible and potentially more efficient than algorithms with fixed shift strategies.

Overall, Boyer-Moore algorithm is advantageous in scenarios where there are mismatches between the pattern and the text, and when the pattern occurs infrequently within the text.