Day 11

1] 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:-

Code -

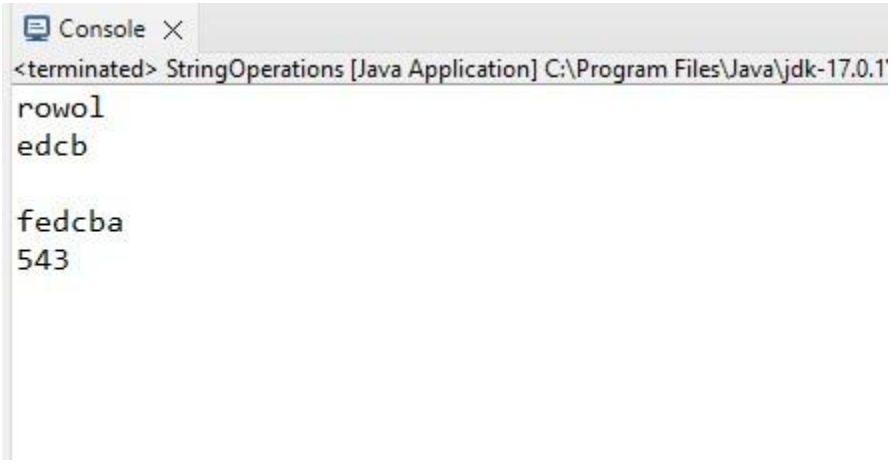
StringOperations.java

```java
package com.assignment;

public class StringOperations {


        public static String middleReversedSubstring(String str1, String str2, int length) {
            String concatenated = str1 + str2;
            String reversed = new StringBuilder(concatenated).reverse().toString();


            if (length >= reversed.length()) {
                return reversed;
            }


            int startIndex = (reversed.length() - length) / 2;

         return reversed.substring(startIndex, startIndex + length);
        }

        public static void main(String[] args) {
```

StringOperations.java

```java
                return reversed;
            }


            int startIndex = (reversed.length() - length) / 2;

         return reversed.substring(startIndex, startIndex + length);
        }

        public static void main(String[] args) {

            System.out.println(middleReversedSubstring("hello", "world", 5));
            System.out.println(middleReversedSubstring("abc", "def", 4));
            System.out.println(middleReversedSubstring("", "", 2));
            System.out.println(middleReversedSubstring("abc", "def", 10));
            System.out.println(middleReversedSubstring("123", "456", 3));
        }
    }
```

Output -



2] 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:-

Code -

NaivePatternSearch.java

```java
package com.assignment;

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

    public class NaivePatternSearch {

        public static class SearchResult {
            List<Integer> positions;
            int comparisons;

            public SearchResult(List<Integer> positions, int comparisons) {
                this.positions = positions;
                this.comparisons = comparisons;
            }
        }

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

```

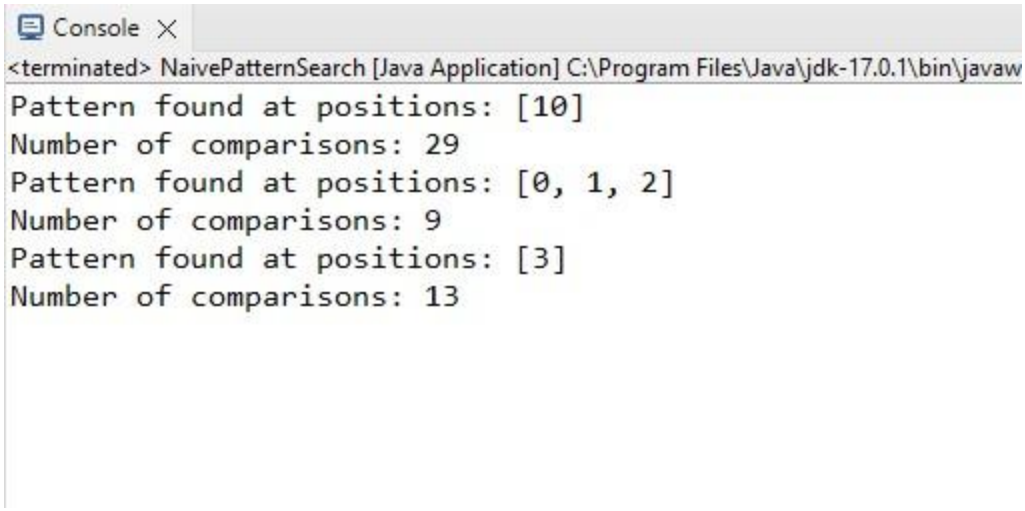
NaivePatternSearch.java

```
21              int n = text.length();
22              int m = pattern.length();
23
24
25              for (int i = 0; i <= n - m; i++) {
26                  int j;
27
28                  for (j = 0; j < m; j++) {
29                      comparisons++;
30                      if (text.charAt(i + j) != pattern.charAt(j)) {
31                          break;
32                      }
33                  }
34
35                  if (j == m) {
36                      positions.add(i);
37                  }
38              }
39
40              return new SearchResult(positions, comparisons);
41          }
42
43          public static void main(String[] args) {
44
```

NaivePatternSearch.java

```
39
40              return new SearchResult(positions, comparisons);
41          }
42
43          public static void main(String[] args) {
44
45              String text = "ABABDABACDABABCABAB";
46              String pattern = "ABABCABAB";
47
48              SearchResult result = naivePatternSearch(text, pattern);
49              System.out.println("Pattern found at positions: " + result.positions);
50              System.out.println("Number of comparisons: " + result.comparisons);
51
52
53              SearchResult result2 = naivePatternSearch("AAAAA", "AAA");
54              System.out.println("Pattern found at positions: " + result2.positions);
55              System.out.println("Number of comparisons: " + result2.comparisons);
56
57              SearchResult result3 = naivePatternSearch("HELLO WORLD", "LO");
58              System.out.println("Pattern found at positions: " + result3.positions);
59              System.out.println("Number of comparisons: " + result3.comparisons);
60          }
61      }
62
```

Output -



3] 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:-
Code -

KMPAlgorithm.java

```java
package com.assignment;

public class KMPAlgorithm {

    public static void KMPSearch(String pattern, String text) {
            int m = pattern.length();
            int n = text.length();


            int[] lps = new int[m];
            computeLPSArray(pattern, m, lps);

            int i = 0;
            int j = 0;
            while (i < n) {
                if (pattern.charAt(j) == text.charAt(i)) {
                    j++;
                    i++;
                }

                if (j == m) {
                    System.out.println("Found pattern at index " + (i - j));
                    j = lps[j - 1];
                } else if (i < n && pattern.charAt(j) != text.charAt(i)) {
```

KMPAlgorithm.java

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


        private static void computeLPSArray(String pattern, int m, int[] lps) {
            int len = 0;
            int i = 1;
            lps[0] = 0;

            while (i < m) {
                if (pattern.charAt(i) == pattern.charAt(len)) {
                    len++;
                    lps[i] = len;
                    i++;
                } else {
```

KMPAlgorithm.java

```
                lps[0] = 0;

                while (i < m) {
                    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++;
                        }
                    }
                }
            }

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

Output -

How Preprocessing Improves Search Time:

- **Naive Approach:**
  - The naive pattern searching algorithm checks for a match by comparing each character of the pattern with the characters of the text, leading to many redundant comparisons, especially when repetitive patterns are involved.
- **KMP Approach:**
  - The KMP algorithm preprocesses the pattern to create the LPS array.
  - When a mismatch occurs, the LPS array is used to skip a part of the text, reducing the number of comparisons.
  - Instead of moving the pattern by one character, the KMP algorithm uses the information in the LPS array to move the pattern to the right position, thus significantly improving the efficiency.

The KMP algorithm's preprocessing step ensures that each character in the text is compared only once, making it more efficient with a time complexity of O(n+m), where n is the length of the text and m is the length of the pattern. This is a significant improvement over the naive approach, which can have a time complexity of $O(n \cdot m)$in the worst case.

4] 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:-
Code -

RabinKarpAlgo.java

```java
package com.wipro.graphalgo;

public class RabinKarpAlgo {



        private final static int PRIME = 101;

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

            RabinKarpAlgo rk = new RabinKarpAlgo();
            rk.search(pattern, text);
        }

        public void search(String pattern, String text) {
            int m = pattern.length();
            int n = text.length();
            long patternHash = createHash(pattern, m - 1);
            long textHash = createHash(text, m - 1);

            for (int i = 1; i <= n - m + 1; i++) {
                if (patternHash == textHash && checkEqual(text, i - 1, i + m - 2, pattern, 0, m - 1)) {
```

RabinKarpAlgo.java

```java
            int n = text.length();
            long patternHash = createHash(pattern, m - 1);
            long textHash = createHash(text, m - 1);

            for (int i = 1; i <= n - m + 1; i++) {
                if (patternHash == textHash && checkEqual(text, i - 1, i + m - 2, pattern, 0, m - 1)) {
                    System.out.println("Pattern found at index: " + (i - 1));
                }
                if (i < n - m + 1) {
                    textHash = recalculateHash(text, i - 1, i + m - 1, textHash, m);
                }
            }
        }

        private long createHash(String str, int end) {
            long hash = 0;
            for (int i = 0; i <= end; i++) {
                hash += str.charAt(i) * Math.pow(PRIME, i);
            }
            return hash;
        }

private long recalculateHash(String str, int oldIndex, int newIndex, long oldHash, int patternLength) {
            long newHash = oldHash - str.charAt(oldIndex);
```

```java
RabinKarpAlgo.java
        }

private long recalculateHash(String str, int oldIndex, int newIndex, long oldHash, int patternLength) {
            long newHash = oldHash - str.charAt(oldIndex);
            newHash /= PRIME;
            newHash += str.charAt(newIndex) * Math.pow(PRIME, patternLength - 1);
            return newHash;
        }

        private boolean checkEqual(String str1, int start1, int end1, String str2, int start2, int end2)
            if (end1 - start1 != end2 - start2) {
                return false;
            }
            while (start1 <= end1 && start2 <= end2) {
                if (str1.charAt(start1) != str2.charAt(start2)) {
                    return false;
                }
                start1++;
                start2++;
            }
            return true;
        }
    }

```

Output -

```
Console
<terminated> RabinKarpAlgo [Java Application] C:\Program Files\Java\jdk-17.0.1\bin\javaw.exe (05-
Pattern found at index: 3
```

Handling Hash Collisions:

Impact of Hash Collisions:

- Hash collisions occur when different substrings have the same hash value.
- Collisions can lead to unnecessary character comparisons, reducing the algorithm's efficiency.

- In the worst case, the performance can degrade to that of the naive approach, i.e., $O(n \cdot m)$

Handling Hash Collisions:

- Use a good hash function and a large prime number to minimize the probability of collisions.
- When a hash collision occurs (i.e., when the hash values match), perform a character-by-character comparison to verify the match.
- If the characters match, a valid match is found.
- If the characters do not match, it was a false positive due to a hash collision.

5] 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:-
Code -

BoyerMooreAlgo.java

```
1  package com.wipro.graphalgo;
2
3  public class BoyerMooreAlgo {
4
5          private final int R; // the radix
6          private int[] right; // the bad-character skip array
7          private String pat; // or as a char array
8
9          public BoyerMooreAlgo(String pat) {
10             this.R = 256;
11             this.pat = pat;
12
13             // position of rightmost occurrence of c in the pattern
14             right = new int[R];
15             for (int c = 0; c < R; c++)
16                 right[c] = -1;
17             for (int j = 0; j < pat.length(); j++)
18                 right[pat.charAt(j)] = j;
19         }
20
21         public int search(String txt) {
```

BoyerMooreAlgo.java

```
17              for (int j = 0; j < pat.length(); j++)
18                  right[pat.charAt(j)] = j;
19          }
20
21          public int search(String txt) {
22              int m = pat.length();
23              int n = txt.length();
24              int skip;
25              for (int i = 0; i <= n - m; i += skip) {
26                  skip = 0;
27                  for (int j = m - 1; j >= 0; j--) {
28                      if (pat.charAt(j) != txt.charAt(i + j)) {
29                          skip = Math.max(1, j - right[txt.charAt(i + j)]);
30                          break;
31                      }
32                  }
33                  if (skip == 0) return i; // found
34              }
35              return n; // not found
36          }
37
38          public static void main(String[] args) {
```

BoyerMooreAlgo.java

```
32                  }
33                  if (skip == 0) return i; // found
34              }
35              return n; // not found
36          }
37
38          public static void main(String[] args) {
39              String pat = "WIPRO";
40              String txt = "WELCOMETOWIPRO";
41              BoyerMooreAlgo bm = new BoyerMooreAlgo(pat);
42              int offset = bm.search(txt);
43              if (offset < txt.length()) {
44                  System.out.println("Pattern found at index " + offset);
45              } else {
46                  System.out.println("Pattern not found");
47              }
48          }
49      }
50
```

Output -

Why Boyer-Moore Algorithm Can Outperform Others:

- **Efficiency:**
  - The Boyer-Moore algorithm often skips large sections of the text, resulting in fewer comparisons.
  - It compares the pattern from right to left, allowing quick identification and skipping over unmatched sections.
- **Best-Case Performance:**
  - In the best-case scenario, the algorithm achieves sub-linear performance, with the average-case time complexity being O(n/m), where n is the length of the text and mmm is the length of the pattern.
- **Heuristics:**
  - **Bad Character Heuristic:** Helps in skipping sections of text where mismatches occur.
  - **Good Suffix Heuristic (not used here but part of the full Boyer-Moore algorithm):** Uses information from the suffixes of the pattern to skip over sections when partial matches are found.