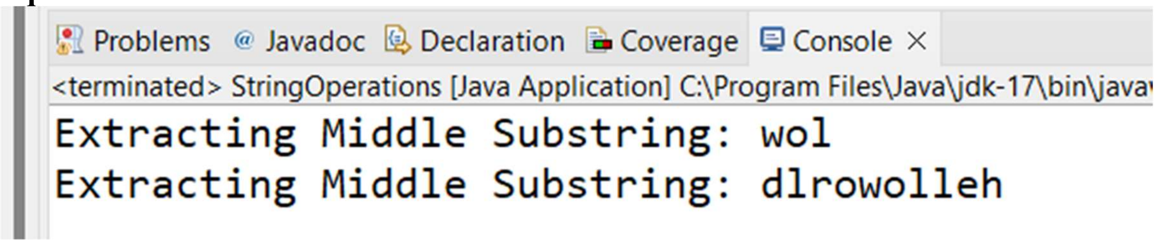
Day 11:
Task 1: String Operations
Write a method that takes two strings, concatenates them, reverses the result, and then extracts the middle substring of the given length. Ensure your method handles edge cases, such as an empty string or a substring length larger than the concatenated string.

```java
package com.dsassignment.day11;

public class StringOperations {

    public static String extractMiddleSubstring(String
str1, String str2, int length) {
        if (str1 == null || str2 == null || length <= 0)
{
            return "";
        }

        String concatenated = str1 + str2;

        StringBuilder reversed = new
StringBuilder(concatenated).reverse();

        int middleIndex = reversed.length() / 2;

        // Ensure the length of the substring is within
the bounds
        int start = Math.max(0, middleIndex - length /
2);
        int end = Math.min(reversed.length(), start +
length);

        String middleSubstring =
reversed.substring(start, end);

        return middleSubstring;
    }

    public static void main(String[] args) {
        System.out.println("Extracting Middle Substring:
" + extractMiddleSubstring("hello", "world", 3));
```

```
        System.out.println("Extracting Middle Substring:
" + extractMiddleSubstring("hello", "world", 10));
    }
}
```

Output:

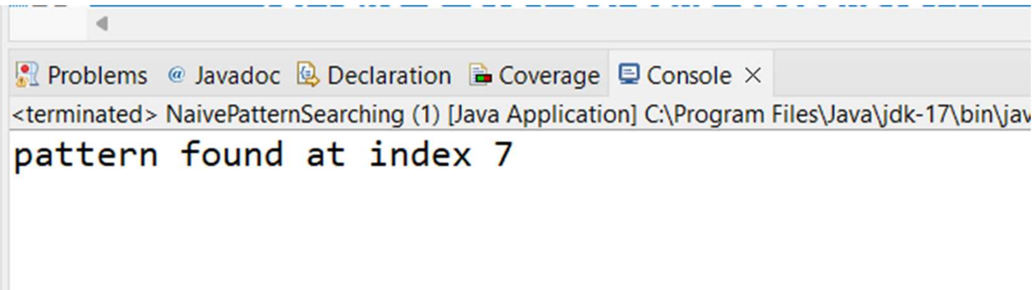


Task 2: Naive Pattern Search
Implement the naive pattern searching algorithm to find all occurrences of a pattern within a given text string. Count the number of comparisons made during the search to evaluate the efficiency of the algorithm.

```
package com.dsassignment.day11;

public class NaivePatternSearching {

    public static void main(String[] args) {
        String text = "I Love Cats";
        String pattern = "Cats";
        search(text, pattern);
    }

    private static void search(String text, String
pattern) {
        int strleng = text.length();
        int patleng = pattern.length();


        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);
            }
        }
    }

}
```

Output:



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.

```
package com.dsassignment.day11;

public class KMPPatternSearching {
    public static void main(String[] args) {
        String text = "AABAACAADAABAABA";
        String pattern = "AABA";

        System.out.println("\nKMP Pattern Searching:");
        searchKMP(text, pattern);
    }

    private static void searchKMP(String text, String 
pattern) {
        int n = text.length();
```

```
        int m = pattern.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)) {
                i++;
                j++;
            }
            if (j == m) {
                System.out.println("Pattern found 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 = i + 1;
            }
        }

    }

    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++;
```

```java
                } else {
                    if (len != 0) {
                        len = lps[len - 1];
                    } else {
                        lps[i] = len;
                        i++;
                    }
                }
            }
        }

}
```

```
KMP Pattern Searching:
Pattern found at index 0
Pattern found at index 9
Pattern found at index 12
```

1)Pre-processing (Computing LPS Array): Before searching for the pattern in the text, the KMP algorithm preprocesses the pattern to compute the Longest Prefix Suffix (LPS) array. This array stores the length of the longest proper prefix which is also a suffix for each prefix of the pattern. This pre-processing step allows the algorithm to avoid unnecessary comparisons by utilizing information about the pattern's structure.

2)Avoiding Redundant Comparisons: During the search phase, the algorithm compares the characters of the text and pattern intelligently based on the information stored in the LPS array. Whenever a mismatch occurs, instead of restarting the comparison from the beginning of the pattern as in the naive approach, the algorithm shifts the pattern by the maximum possible length based on the LPS array, thus avoiding redundant comparisons.

3)Efficient Pattern Matching: By leveraging the pre-processed LPS array, the KMP algorithm ensures that each character in the text is compared with at most once against the characters of the pattern. This significantly reduces the number of comparisons required, especially for patterns with repetitive substrings or patterns containing a long prefix that is also a suffix. As a result, the KMP algorithm achieves a linear time complexity O(n + m), where n is the length of the text and m is the length of the pattern, making it much more efficient than the naive approach, which has a time complexity of O(n * m).

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.

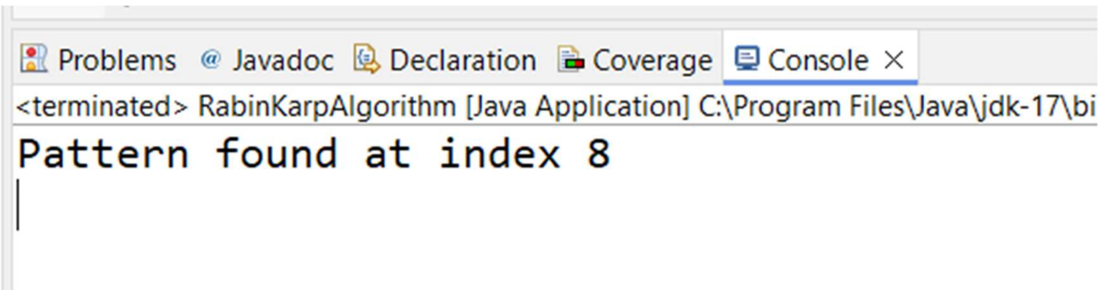
Solution:

The Rabin-Karp algorithm is a string-searching algorithm that uses hashing to find a pattern in a text efficiently. The key idea is to hash the pattern and then compare this hash to the hash of substrings of the text. If the hashes match, the algorithm then checks the actual substring to verify the match, thus handling potential hash collisions.

```
package com.dsassignment.day11;

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

    static void search(String pat, String txt) {
        int M = pat.length();
        int N = txt.length();
        int i, j;
        int p = 0; // hash value for pattern
        int t = 0; // hash value for text
        int h = 1;
        for (i = 0; i < M - 1; i++)
            h = h * d;
        for (i = 0; i < M; i++) {
            p = d * p + pat.charAt(i);
            t = d * t + txt.charAt(i);
        }
        for (i = 0; i <= N - M; i++) {
                if (p == t) {

                 for (j = 0; j < M; j++) {
                     if (txt.charAt(i + j) !=
pat.charAt(j))
                         break;
                 }
                 if (j == M)
```

```
                        System.out.println("Pattern found at
index " + i);
            }

            if (i < N - M) {
                t = d * (t - txt.charAt(i) * h) +
txt.charAt(i + M);
            }
        }
    }

    public static void main(String[] args) {
        String txt = "ccaccaaedba";
        String pat = "dba";

        search(pat, txt);
    }
}
```

Output:



Impact of Hash Collisions

Hash Collisions Impact:

1. **False Positives**: When two different strings (substrings in this context) have the same hash value, it's called a collision. This leads to false positives where the algorithm thinks it has found the pattern, but in reality, the actual substring does not match the pattern.
2. **Performance Degradation**: Each collision requires a character-by-character comparison to verify the match, which can degrade the algorithm's performance, especially in the worst-case scenario where many collisions occur.

Handling Hash Collisions

Handling Collisions:

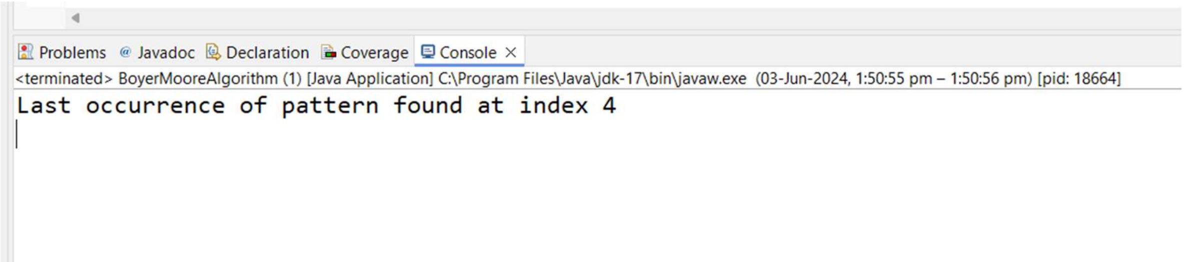
1. **Verification Step**: After finding a hash match, compare the actual substring with the pattern to confirm the match. This is crucial as it ensures correctness even in the presence of collisions.
2. **Using a Large Prime Number (q)**: A large prime number in the modulus operation reduces the chance of collisions by spreading out the hash values more evenly.
3. **Efficient Rolling Hash Computation**: The rolling hash efficiently updates the hash value when the window slides, making the algorithm more efficient despite potential 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.

```
package com.dsassignment.day11;

public class BoyerMooreAlgorithm {

    public final static int ALPHABET_SIZE = 256;

    static void badCharHeuristic(char[] str, int size,
int[] badChar) {
        for (int i = 0; i < ALPHABET_SIZE; i++) {
            badChar[i] = -1;
        }

        for (int i = 0; i < size; i++) {
            badChar[(int) str[i]] = i;
        }
    }

    static int search(char[] txt, char[] pat) {
        int m = pat.length;
        int n = txt.length;

        int[] badChar = new int[ALPHABET_SIZE];

        badCharHeuristic(pat, m, badChar);

        int s = 0;
        int lastOccurrence = -1; // Initialize to -1,
indicating no occurrence found
```

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

            while (j >= 0 && pat[j] == txt[s + j]) {
                j--;
            }

            if (j < 0) {
                lastOccurrence = s;

                s += (s + m < n) ? m - badChar[txt[s +
m]] : 1;

            } else {
                s += Math.max(1, j - badChar[txt[s +
j]]);
            }
        }

        return lastOccurrence;
    }

    public static void main(String[] args) {
        String txt = "ABAAABCD";
        String pat = "ABC";
        int lastOccurrence = search(txt.toCharArray(),
pat.toCharArray());
        if (lastOccurrence != -1) {
            System.out.println("Last occurrence of
pattern found at index " + lastOccurrence);
        } else {
            System.out.println("Pattern not found in the
text.");
        }
    }
}
```

Output:



The Boyer-Moore algorithm is particularly effective in scenarios where:

- **Large Alphabets**: It performs well when the alphabet size is large, as it uses a bad character heuristic to skip comparisons based on mismatches, reducing the number of comparisons needed.
- **Multiple Mismatches**: It efficiently handles situations with multiple mismatches by determining the maximum shift distance based on the bad character heuristic and the good suffix rule.
- **Preprocessing**: It preprocesses the pattern to create the bad character array, allowing for faster searches in the text.
- **Search from Right to Left**: The algorithm searches from right to left, which can be advantageous, especially when searching for the last occurrence of a pattern as it minimizes the search space.

Boyer-Moore algorithm's ability to efficiently skip comparisons and its preprocessing step make it well-suited for scenarios where other algorithms may struggle, such as searching for the last occurrence of a substring in a large text.