

## 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.

Ans:

```
package Assignment_Day11;
```

```
public class StringOperations {

    public static String middleReversedSubstring(String str1, String str2, int length) {

        String concatenated = str1 + str2;

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

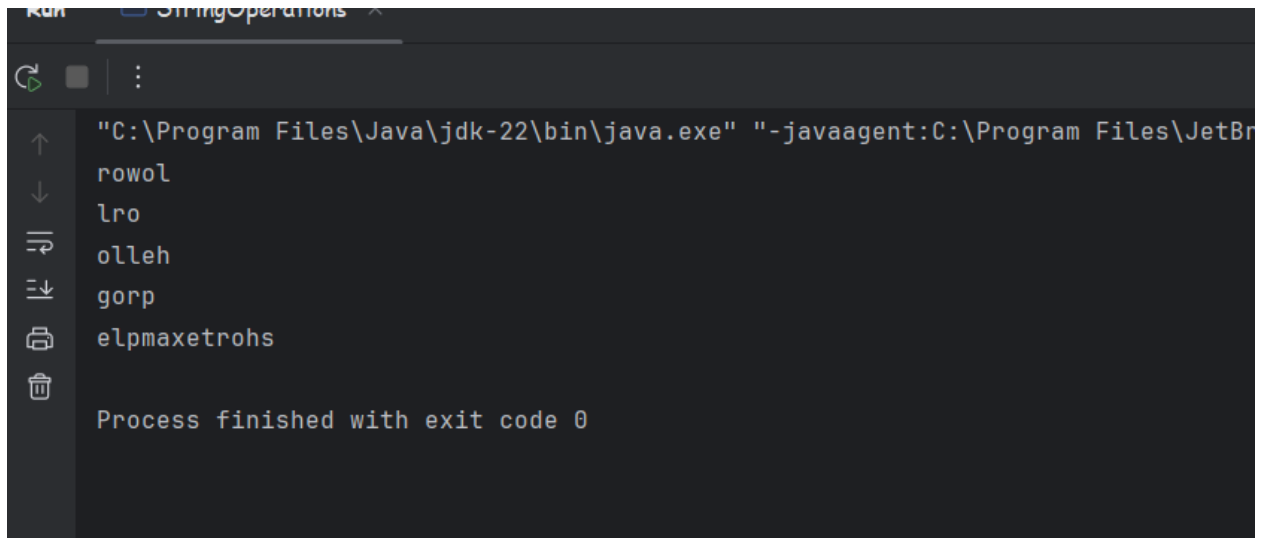
        if (length > reversed.length()) {
            return reversed;
        }

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

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

    public static void main(String[] args) {
        // Test cases
        System.out.println(middleReversedSubstring("hello", "world", 5));
        System.out.println(middleReversedSubstring("", "world", 3));
        System.out.println(middleReversedSubstring("hello", "", 10));
        System.out.println(middleReversedSubstring("java", "program", 4));
        System.out.println(middleReversedSubstring("short", "example", 20));
    }
}
```

OutPut:



```
"C:\Program Files\Java\jdk-22\bin\java.exe" "-javaagent:C:\Program Files\JetBr
rowol
lro
olleh
gorp
elpmaxetrohs

Process finished with exit code 0
```

## 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.

Ans:

```
package Assignment_Day11;
```

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

```
public class NaivePatternSearch {
```

```
    public static class SearchResult {
        List<Integer> occurrences;
        int comparisonCount;
```

```
        public SearchResult(List<Integer> occurrences, int comparisonCount) {
            this.occurrences = occurrences;
            this.comparisonCount = comparisonCount;
        }
    }
```

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

```

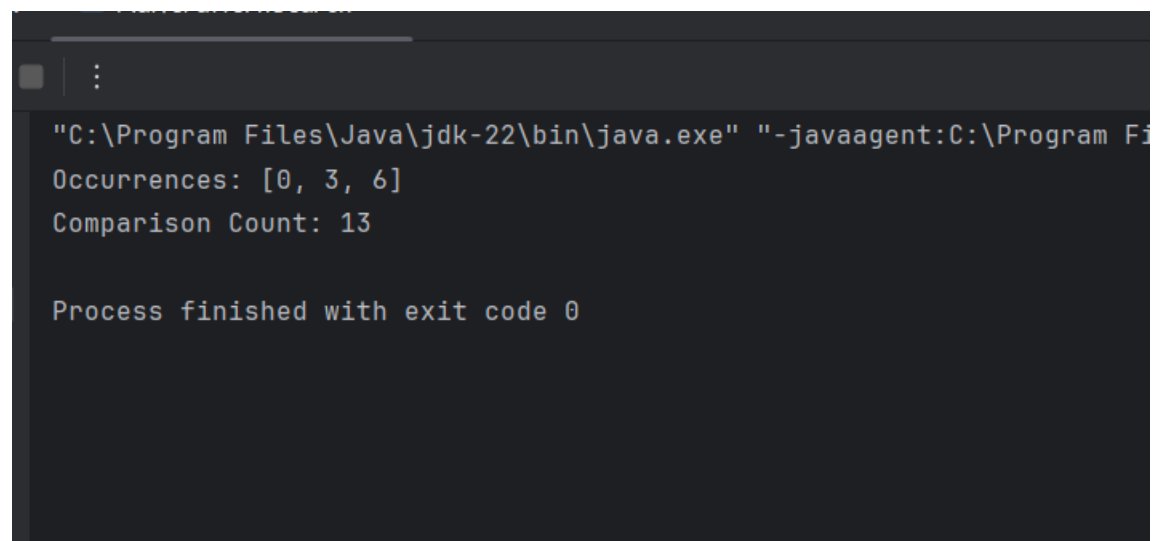
    for (int i = 0; i <= n - m; i++) {
        int j;
        for (j = 0; j < m; j++) {
            comparisonCount++;
            if (text.charAt(i + j) != pattern.charAt(j)) {
                break;
            }
        }
        if (j == m) {
            occurrences.add(i);
        }
    }

    return new SearchResult(occurrences, comparisonCount);
}

public static void main(String[] args) {
    String text = "SHUSHUSHU";
    String pattern = "SHU";
    SearchResult result = naivePatternSearch(text, pattern);
    System.out.println("Occurrences: " + result.occurrences);
    System.out.println("Comparison Count: " + result.comparisonCount);
}
}

```

OutPut:



```

"C:\Program Files\Java\jdk-22\bin\java.exe" "-javaagent:C:\Program Fi
Occurrences: [0, 3, 6]
Comparison Count: 13

Process finished with exit code 0

```

### 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.

Ans:

```
package Assignment_Day11;
```

```
public class KMPAlgorithm {
```

```
    public static void KMPSearch(String text, String pattern) {
```

```
        int n = text.length();
```

```
        int m = pattern.length();
```

```
        // Create lps[] that will hold the longest prefix suffix values for pattern
```

```
        int[] lps = new int[m];
```

```
        int j = 0; // index for pattern
```

```
        // Preprocess the pattern (calculate lps[] array)
```

```
        computeLPSArray(pattern, m, lps);
```

```
        int i = 0; // index for text
```

```
        int comparisonCount = 0; // to count the number of comparisons
```

```
        while (i < n) {
```

```
            comparisonCount++;
```

```
            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];
```

```
            }
```

```
            // mismatch after j matches
```

```
            else if (i < n && pattern.charAt(j) != text.charAt(i)) {
```

```
                // Do not match lps[0..lps[j-1]] characters, they will match anyway
```

```
                if (j != 0) {
```

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

```
                } else {
```

```
                    i = i + 1;
```

```
                }
```

```
            }
```

```
        }
```

```

        System.out.println("Total comparisons: " + comparisonCount);
    }

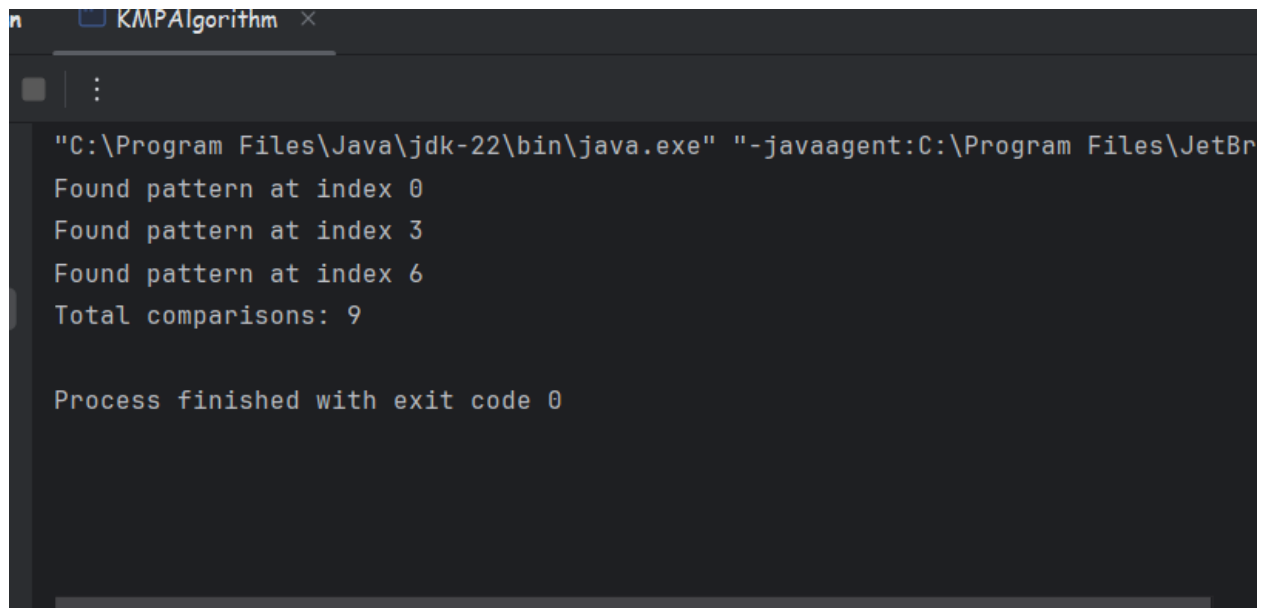
    // Fills lps[] for given pattern pat[0..m-1]
    private static void computeLPSArray(String pattern, int m, int[] lps) {
        int length = 0; // length of the previous longest prefix suffix
        int i = 1;
        lps[0] = 0; // lps[0] is always 0

        // the loop calculates lps[i] for i = 1 to m-1
        while (i < m) {
            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 {
                    lps[i] = length;
                    i++;
                }
            }
        }
    }

    public static void main(String[] args) {
        String text = "SHUSHUSHU";
        String pattern = "SHU";
        KMPSearch(text, pattern);
    }
}

```

OutPut:



```
"C:\Program Files\Java\jdk-22\bin\java.exe" "-javaagent:C:\Program Files\JetBr
Found pattern at index 0
Found pattern at index 3
Found pattern at index 6
Total comparisons: 9

Process finished with exit code 0
```

#### 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.

Ans:

```
package Assignment_Day11;

public class RabinKarpAlgorithm {

    public final static int  $d$  = 256;

    public final static int  $q$  = 101;

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

        for (i = 0; i < m - 1; i++) {
             $h = (h * d) \% q$ ;
        }
    }
}
```

```

for (i = 0; i < m; i++) {
    p = (d * p + pattern.charAt(i)) % q;
    t = (d * t + text.charAt(i)) % q;
}

for (i = 0; i <= n - m; i++) {

    if (p == t) {
        boolean match = true;
        for (j = 0; j < m; j++) {
            comparisonCount++;
            if (text.charAt(i + j) != pattern.charAt(j)) {
                match = false;
                break;
            }
        }

        if (match) {
            System.out.println("Pattern found at index " + i);
        }
    }

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

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

System.out.println("Total comparisons: " + comparisonCount);
}

public static void main(String[] args) {
    String text = "shubhamshubham";
    String pattern = "shubham";
    search(text, pattern);
}
}

```

Output:

```
"C:\Program Files\Java\jdk-22\bin\java.exe" "-javaagent:C:\Program Files\Je
Pattern found at index 0
Pattern found at index 7
Total comparisons: 14

Process finished with exit code 0
```

### 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 Assignment_Day11;

public class BoyerMoore {

    public static int lastIndexOf(String text, String pattern) {
        if (text == null || pattern == null) {
            return -1;
        }

        int[] badCharTable = buildBadCharTable(pattern);
        int[] goodSuffixTable = buildGoodSuffixTable(pattern);

        int textLength = text.length();
        int patternLength = pattern.length();

        int s = 0;
        int lastOccurrence = -1;

        while (s <= (textLength - patternLength)) {
            int j = patternLength - 1;

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



```

        if (j < 0) {
            lastOccurrence = s;
            s += goodSuffixTable[0];
        } else {
            s += Math.max(goodSuffixTable[j], j - badCharTable[text.charAt(s + j)]);
        }
    }

    return lastOccurrence;
}

private static int[] buildBadCharTable(String pattern) {
    final int ALPHABET_SIZE = 256;
    int[] table = new int[ALPHABET_SIZE];
    int patternLength = pattern.length();

    for (int i = 0; i < ALPHABET_SIZE; i++) {
        table[i] = -1;
    }

    for (int i = 0; i < patternLength - 1; i++) {
        table[pattern.charAt(i)] = i;
    }

    return table;
}

private static int[] buildGoodSuffixTable(String pattern) {
    int patternLength = pattern.length();
    int[] table = new int[patternLength];
    int[] suffixes = new int[patternLength];

    for (int i = patternLength - 1; i >= 0; i--) {
        int j = i;
        while (j >= 0 && pattern.charAt(j) == pattern.charAt(patternLength - 1 - i + j)) {
            j--;
        }
        suffixes[i] = i - j;
    }

    for (int i = 0; i < patternLength; i++) {
        table[i] = patternLength;
    }
}

```

```

int j = 0;
for (int i = patternLength - 1; i >= 0; i--) {
    if (i + 1 == suffixes[i]) {
        for (; j < patternLength - 1 - i; j++) {
            if (table[j] == patternLength) {
                table[j] = patternLength - 1 - i;
            }
        }
    }
}

for (int i = 0; i <= patternLength - 2; i++) {
    table[patternLength - 1 - suffixes[i]] = patternLength - 1 - i;
}

return table;
}

public static void main(String[] args) {
    String text = "shubhamshubham";
    String pattern = "shubham";
    int lastIndex = lastIndexOf(text, pattern);
    System.out.println("Last occurrence of the pattern is at index: " + lastIndex);
}
}

```

Output:

```

"C:\Program Files\Java\jdk-22\bin\java.exe" "-javaagent:C:\Program Files
Last occurrence of the pattern is at index: 7

Process finished with exit code 0
|

```