

Pattern Searching Algorithms - 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.

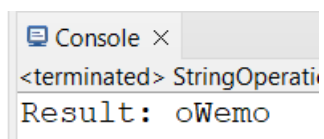
Code:

```
package stringoperations;

public class StringOperations {
    public static String reverseMiddleSubstring(String s1, String s2, int length) {
        String concatenated = s1 + s2;
        String reversed = new StringBuilder(concatenated).reverse().toString();
        int startIndex = (reversed.length() - length) / 2;
        startIndex = Math.max(0, startIndex);
        String middleSubstring = reversed.substring(startIndex, startIndex + length);
        return middleSubstring;
    }

    public static void main(String[] args) {
        String str1 = "Welcome";
        String str2 = "World";
        int length = 5;
        String result = reverseMiddleSubstring(str1, str2, length);
        System.out.println("Result: " + result);
    }
}
```

Output:



The screenshot shows a console window with a title bar that says "Console" and a close button. The output text in the console is: "<terminated> StringOperati" on the first line and "Result: oWemo" on the second line.

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.

Code:

```
package naivepatternsearch;

public class NaivePatternSearch {
    public static void search(String text, String pattern) {
        int textL = text.length();
        int patternL = pattern.length();
        int noOfComparisons = 0;

        for (int i = 0; i <= textL - patternL; i++) {
            int j;
            for (j = 0; j < patternL; j++) {
                noOfComparisons++;
                if (text.charAt(i + j) != pattern.charAt(j))
                    break;
            }
            if (j == patternL)
                System.out.println("Pattern found at index " + i);
        }
        System.out.println("Total comparisons made: " + noOfComparisons);
    }

    public static void main(String[] args) {
        String text = "ABABDAABCDAAABCCABCABAB";
        String pattern = "AABCC";
        search(text, pattern);
    }
}
```

Output:

```
Console ×  
<terminated> NaivePatternSearch [Java Application] C  
Pattern found at index 10  
Total comparisons made: 31
```

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.

Code:

```
package kmpalgorithm;  
  
public class KMPAlgorithmImplementation {  
    public static void implementingKMPSearch(String text, String pattern) {  
        int[] lps = computeLPSArray(pattern);  
        int textL = text.length();  
        int patternL = pattern.length();  
        int comparisons = 0;  
  
        int i = 0;  
        int j = 0;  
        while (i < textL) {  
            comparisons++;  
            if (pattern.charAt(j) == text.charAt(i)) {  
                j++;  
                i++;  
            }  
            if (j == patternL) {  
                System.out.println("Pattern found at index " + (i - j));  
                j = lps[j - 1];  
            }  
        }  
    }  
}
```

```

    } else if (i < textL && pattern.charAt(j) != text.charAt(i)) {
if (j != 0)
j = lps[j - 1];
else
i++;
    }
}
System.out.println("Total comparisons made: " + comparisons);
}

```

```

private static int[] computeLPSArray(String pattern) {
int patternLength = pattern.length();
int[] lps = new int[patternLength];
int len = 0;

lps[0] = 0;
int i = 1;
while (i < patternLength) {
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;
}

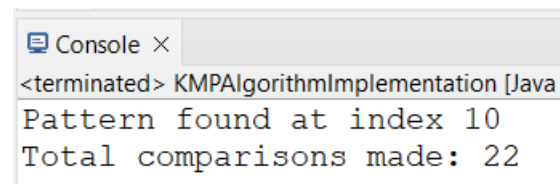
```

```

public static void main(String[] args) {
    String text = "ABABDAABCDABCCABCABAB";
    String pattern = "AABCC";
    implementingKMPSearch(text, pattern);
}
}

```

Output:



```

Console ×
<terminated> KMPAlgorithmImplementation [Java
Pattern found at index 10
Total comparisons made: 22

```

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.

Code:

```

package rabinkarpsubstringsearch;

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

public class RabinKarpSubstringSearch {
    private static final int PRIME = 101;

    public static List<Integer> search(String text, String pattern) {
        List<Integer> indices = new ArrayList<>();
        int textL = text.length();
        int patternL = pattern.length();
        long patternHash = calculateHash(pattern, patternL);
        long textHash = calculateHash(text, patternL);
    }
}

```

```

for (int i = 0; i <= textL - patternL; i++) {
if (textHash == patternHash && checkEqual(text, i, i + patternL - 1, pattern, 0,
patternL - 1)) {
indices.add(i);
}
if (i < textL - patternL) {
textHash = recalculateHash(text, i, i + patternL, textHash, patternL);
}
}

return indices;
}

```

```

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

```

```

private static 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 static boolean checkEqual(String text, int start1, int end1, String pattern, int
start2, int end2) {
if (end1 - start1 != end2 - start2) {
return false;
}
}

```

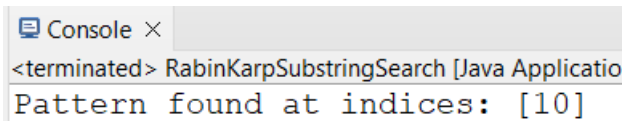
```

while (start1 <= end1 && start2 <= end2) {
    if (text.charAt(start1) != pattern.charAt(start2)) {
        return false;
    }
    start1++;
    start2++;
}
return true;
}

public static void main(String[] args) {
    String text = "ABABDAABCDAAABCCABCABAB";
    String pattern = "AABCC";
    List<Integer> indices = search(text, pattern);
    if (indices.isEmpty()) {
        System.out.println("Pattern not found in the text.");
    } else {
        System.out.println("Pattern found at indices: " + indices);
    }
}

```

Output:



```

Console ×
<terminated> RabinKarpSubstringSearch [Java Applicatio
Pattern found at indices: [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.

Code:

```

package boyermoorealgorithm;

```

```

import java.util.Arrays;

public class BoyerMooreAlgorithmApplication {
    private static final int CHAR_SET_SIZE = 256;

    public static int findLastOccurrence(String text, String pattern) {
        int[] lastOccurrence = getLastOccurrenceArray(pattern);
        int textL = text.length();
        int patternL = pattern.length();
        int i = patternL - 1;
        int j = patternL - 1;

        while (i < textL) {
            if (text.charAt(i) == pattern.charAt(j)) {
                if (j == 0) {
                    return i;
                }
                i--;
                j--;
            } else {
                i += patternL - Math.min(j, 1 + lastOccurrence[text.charAt(i)]);
                j = patternL - 1;
            }
        }
        return -1;
    }

    private static int[] getLastOccurrenceArray(String pattern) {
        int[] lastOccurrence = new int[CHAR_SET_SIZE];
        Arrays.fill(lastOccurrence, -1);
        for (int i = 0; i < pattern.length(); i++) {
            lastOccurrence[pattern.charAt(i)] = i;
        }
    }
}

```



```
return lastOccurrence;
```

```
}
```

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

```
String text = "ABABDAABCDAAABCCABCABAB";
```

```
String pattern = "AABCC";
```

```
int lastIndex = findLastOccurrence(text, pattern);
```

```
if (lastIndex != -1) {
```

```
System.out.println("Last occurrence found at index: " + lastIndex);
```

```
} else {
```

```
System.out.println("Pattern not found in the text.");
```

```
}
```

```
}
```

```
}
```

Output:

Console ×

<terminated> BoyerMooreAlgorithmApplication [Java Application]

Last occurrence found at index: 10