Day 21 Assignment

Name: Mehul Anjikhane Email: mehulanjikhane13@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.

```
package algorithms;

public class StringOperations {

	public static String getMiddleSubString(String str1, String
str2, int substringlength) {
			if(str1.isEmpty() || str2.isEmpty()) {
			
			return "Empty String(s) provided";
			}
			String concatenatedString = str1.concat(str2);
			int total_Length = concatenatedString.length();
			
			if(substringlength > total_Length) {
				return "Substirng length exceeds concatenated
string length";
			}
			
		int startIndex = (total_Length - substringlength) / 2;
		int endIndex = startIndex + substringlength;
		
			StringBuilder reversedString = new
StringBuilder(concatenatedString).reverse();
		
		return reversedString.substring(startIndex, endIndex);
		}
	
	public static void main(String[] args) {
		String input1 = "This is";
		String input2 = "Mehul, Hi guys";
		int length = 5;
		
		System.out.println("Input Strings: "+ input1 + ", "+
input2);
		System.out.println("Substring Length: "+length);
		System.out.println("Reverse concatenated String: "+ new
StringBuilder(input1 + input2).reverse());

              String result = getMiddleSubString(input1, input2, 
length);
              System.out.println("Reverse Middle SubString: "+ result);
       }
}
```

Output:

```
Input Strings: This is, Mehul, Hi guys
Substring Length: 5
Reverse concatenated String: syug iH ,luheMsi sihT
Reverse Middle SubString: ,luhe
```

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 algorithms;

public class NaivePatternSearch {

       public static void search(String text, String pattern) {
              int textLength = text.length();
              int patternLength = pattern.length();
              int comparisons = 0;
              for (int i = 0; i <= textLength - patternLength; i++) {
                            int j;
                            for (j = 0; j < patternLength; j++) {
                                          comparisons++;
                                          if (text.charAt(i + j) != 
pattern.charAt(j)) {
                                                        break;
                                          }
                            }
                            if (j == patternLength) {
                                   System.out.println("Pattern found at 
index " + i);
                            }
                     }
                     System.out.println("Total comparisons made: " + 
comparisons);
              }
```

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

		System.out.println("Given String: " + text);
		System.out.println("Pattern: " + pattern);

		search(text, pattern);
	}
}
```

Output:

```
Given String: AABAACAADAABAABA
Pattern: AABA
Pattern found at index 0
Pattern found at index 9
Pattern found at index 12
Total comparisons made: 30
```

Task 3: Implementing the KMP Algorithm

Code the Knuth-Morris-Pratt (KMP) algorithm in C# 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 algorithms;

public class KMPSearchAlgorithm {

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

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

		int i = 0;
		int j = 0;
		while (i < textLength) {
				if (pattern.charAt(j) == text.charAt(i)) {
					i++;
					j++;
				}
				if (j == patternLength) {
					System.out.println("Pattern found at 
index " + (i - j));
```

```
                        j = lps[j - 1];
                    }else if (i < textLength && pattern.charAt(j)
!= text.charAt(i)){
                        if (j != 0) {
                            j = lps[j - 1];
                        } else {
                            i++;
                        }
                    }
                }
            }

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

            while (i < 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++;
                    }
                }
            }
        }

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

        System.out.println("Given String: " + text);
        System.out.println("Pattern: " + pattern);

        KMPSearch(pattern, text);
    }
}
```
Output:

```
Given String: ABABDABACDABABCABAB
Pattern: ABABCABAB
Pattern found at index 10
```

How this Pre-Processing Improves the search Time Compared to the Naive Approach?

1) Prefix Function Computation:

• KMP pre-processes the pattern to create an array (lps) that stores the lengths of the longest proper prefix which is also a suffix for each sub-pattern.

• This helps in understanding how much to shift the pattern upon a mismatch.

2) Avoids Re-evaluation:

• The 'lps' array allows the algorithm to skip re-evaluating characters that are known to match.

• Instead of moving one character at a time (like in the naive approach), KMP uses the 'lps' array to determine the next positions to check, thus reducing redundant comparisons.

3) Linear Time Complexity:

• The pre-processing of the pattern takes O(m) time, where m is the length of the pattern.

• The search phase also takes O(n) time, where n is the length of the text.

• Overall, KMP runs in O(n + m) time, making it more efficient than the naive approach, which can have a worst-case time complexity of O(n*m).

4) Efficiency in Overlapping Patterns:

• When parts of the pattern overlap, the 'lps' array allows the algorithm to leverage the overlap, avoiding unnecessary comparisons.

• This is particularly beneficial for patterns with repetitive or self-similar structures.

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.

```
package algorithms;

public class RabinKarpAlgorithm {
```

```java
      private final int d = 256; // Number of characters in the
input alphabet

      public int search(String text, String pattern) {
            int textLength = text.length();
            int patternLength = pattern.length();

            // A prime number for modulo operation (can be changed)
            int prime = 101;

      // Calculate hash values of pattern and first window of text
            int patternHash = calculateHash(pattern, patternLength,
prime);
            int textHash = calculateHash(text, patternLength, prime);
            int hash = 1; // d^(patternLength - 1) for calculating
rolling hash.

            for (int i = 1; i < patternLength; i++) {
                        hash = (hash * d) % prime;
            }

            // Slide the pattern window over the text
            for (int i = 0; i <= textLength - patternLength; i++) {
                  // Check if hash values match
                  if (patternHash == textHash) {
                  // Potential match, compare individual characters
                              boolean isMatch = true;
                              for (int j = 0; j < patternLength; j++) {
                                    if (text.charAt(i + j) !=
pattern.charAt(j)) {
                                                isMatch = false;
                                                break;
                                    }
                              }
                              if (isMatch) {
                              return i; // Pattern found at index i
                              }
                  }
      // Update text hash for the next window using rolling hash
                  if (i < textLength - patternLength) {
                        textHash = (textHash + prime - hash *
text.charAt(i)) %
prime;
                        textHash = (textHash * d) % prime;
                        textHash = (textHash + text.charAt(i +
patternLength)) %
prime;
                  }
            }
```

```
		// Pattern not found
		return -1;
	}

	private int calculateHash(String str, int length, int prime) {
		int hash = 0;
		for (int i = 0; i < length; i++) {
				hash = (hash * d + str.charAt(i)) % prime;
		}
		return hash;
	}

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

		System.out.println("Given String: " + text);
		System.out.println("Pattern: " + pattern);

		RabinKarpAlgorithm rk = new RabinKarpAlgorithm();
		int index = rk.search(text, pattern);

		if (index == -1) {
				System.out.println("Pattern not found");
		} else {
				System.out.println("Pattern found at index: "
+ index);
		}
	}
}
```

Output:

```
Given String: Go Simon Go
Pattern: Simon
Pattern found at index: 3
```

Hash Collisions in Rabin-Karp:

• **Impact:** Collisions can lead to unnecessary character comparisons, reducing efficiency. In the

worst case, if every substring hashes to the same value, the algorithm behaves like a naive

search with time complexity O(n*m).

• **Handling:**

1. Use a good hash function with a low collision probability.

2. After a hash match, perform a character-by-character comparison to confirm the actual

match (as done in the code). This adds a small overhead but ensures accuracy.

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 algorithms;

public class Boyre_MooreAlgorithm {

	public static int lastOccurrence(String text, String pattern)
{

		int[] badChar = new int[256];
		badCharHeuristic(pattern, badChar);

		int m = pattern.length();
		int n = text.length();
		int shift = 0;
		int lastOccurrenceIndex = -1;

		while (shift <= (n - m)) {
			int j = m - 1;
			while (j >= 0 && pattern.charAt(j) ==
text.charAt(shift + j)) {
					j--;
			}

			if (j < 0) {
					lastOccurrenceIndex = shift;
					shift += (shift + m < n) ? m -
badChar[text.charAt(shift + m)] : 1;
			} else {
					shift += Math.max(1, j -
badChar[text.charAt(shift + j)]);
			}
		}
		return lastOccurrenceIndex;
	}

	private static void badCharHeuristic(String str, int[]
badChar) {
		for (int i = 0; i < 256; i++) {
				badChar[i] = -1;
		}

		for (int i = 0; i < str.length(); i++) {
				badChar[str.charAt(i)] = i;
```

```
            }
        }

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

            System.out.println("Given String: " + text);
            System.out.println("Pattern: " + pattern);

            int index = lastOccurrence(text, pattern);
            System.out.println("Last occurrence of the pattern is at
index: " + index);
            }
}
```

Output:

```
Given String: ABCAABBCCABCABCD
Pattern: ABC
Last occurrence of the pattern is at index: 12
```

Why Boyer-Moore Algorithm can Outperform others in Certain Scenarios?

• **Bad Character Heuristic:** The algorithm uses the bad character heuristic to skip sections of the text that do not match the pattern, making the search faster by potentially skipping more characters compared to other algorithms.

• **Good Suffix Heuristic:** It leverages the good suffix heuristic to further optimize the search by shifting the pattern more intelligently when a mismatch occurs after matching some suffix of the pattern.

• **Sub-linear Time Complexity:** Boyer-Moore often operates in sub-linear time on average because it doesn't examine each character of the text and pattern for every shift, unlike the naive approach.

• **Efficient for Large Alphabets:** The algorithm is particularly efficient when the alphabet size is large since the bad character heuristic becomes more effective with a greater variety of characters.

• **Pre-processing:** Although it requires preprocessing of the pattern, this overhead is generally small compared to the speedup gained during the search, especially for long texts.