Himanshu Kamane
kamanehimanshu76@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.

Code

```
package WiproEP;
public class StringOperations {

  public static String processStrings(String str1, String str2, int substringLength) {
      // Step 1: Concatenate the strings
      String concatenated = str1 + str2;

      // Step 2: Reverse the concatenated string
      String reversed = new StringBuilder(concatenated).reverse().toString();

      // Step 3: Extract the middle substring of the given length
      int totalLength = reversed.length();

      // Handle edge cases
      if (substringLength > totalLength) {
          return reversed; // If the requested substring length is greater than the string length, return the whole reversed string
      }

      int startIndex = (totalLength - substringLength) / 2;
      int endIndex = startIndex + substringLength;

      return reversed.substring(startIndex, endIndex);
  }
  public static void main(String[] args) {
      // Test cases
      System.out.println(processStrings("hello", "world", 5));
      System.out.println(processStrings("abc", "def", 3));
      System.out.println(processStrings("", "", 2));
      System.out.println(processStrings("a", "b", 5));
      System.out.println(processStrings("123", "456", 10));
```

```
    }
}
```

Output

```
rowol
edc
ba
654321
```

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

```java
package WiproEP;
import java.util.*;
public class NaivePatternSearch {
  public static class SearchResult {
      int[] positions;
      int comparisonCount;
      public SearchResult(int[] positions, int comparisonCount) {
          this.positions = positions;
          this.comparisonCount = comparisonCount;
      }
  }
  public static SearchResult naivePatternSearch(String text, String pattern) {
      int n = text.length();
      int m = pattern.length();
      int comparisonCount = 0;
      List<Integer> positions = new ArrayList<>();
      // Traverse the text string
      for (int i = 0; i <= n - m; i++) {
          int j;
          // Check for pattern match
          for (j = 0; j < m; j++) {
              comparisonCount++;
              if (text.charAt(i + j) != pattern.charAt(j)) {
                  break;
```

```
            }
        }
        // If the pattern is found
        if (j == m) {
            positions.add(i);
        }
    }
    // Convert positions list to an array
    int[] positionsArray = positions.stream().mapToInt(Integer::intValue).toArray();
    return new SearchResult(positionsArray, comparisonCount);
  }
  public static void main(String[] args) {
    String text = "ABABDABACDABABCABAB";
    String pattern = "ABABCABAB";
    SearchResult result = naivePatternSearch(text, pattern);
    System.out.println("Pattern found at positions: " + Arrays.toString(result.positions));
    System.out.println("Number of comparisons: " + result.comparisonCount);
  }
}
```

Output

```
Pattern found at positions: [10]
Number of comparisons: 29
```

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.

Code

```
package WiproEP;
public class KMPAlgorithm {
  // Method to preprocess the pattern and create the longest prefix suffix (LPS) array
  private static int[] computeLPSArray(String pattern) {
    int[] lps = new int[pattern.length()];
    for (int i = 1, len = 0; i < pattern.length(); ) {
        if (pattern.charAt(i) == pattern.charAt(len)) {
            lps[i++] = ++len;
        } else if (len > 0) {
            len = lps[len - 1];
```

```
            } else {
                lps[i++] = 0;
            }
        }
        return lps;
    }
    // KMP search algorithm
    public static void KMPSearch(String text, String pattern) {
        int[] lps = computeLPSArray(pattern);
        for (int i = 0, j = 0; i < text.length(); ) {
            if (pattern.charAt(j) == text.charAt(i)) {
                i++;
                j++;
                if (j == pattern.length()) {
                    System.out.println("Pattern found at index " + (i - j));
                    j = lps[j - 1];
                }
            } else if (j > 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }
    public static void main(String[] args) {
        String text = "ABABDABACDABABCABAB";
        String pattern = "ABABCABAB";
        KMPSearch(text, pattern);
    }
}
```

Output

```
Pattern found at index 10
```

How Pre-processing Improves Search Time:

The LPS array allows the KMP algorithm to skip portions of the text where a mismatch has already been found, avoiding redundant comparisons. This reduces the overall number of comparisons needed, making the search process more efficient compared to the naive approach. The naive approach may require re-checking characters multiple times, leading to a worst-case time complexity of
$O(n \times m)$,where n is the length of the text and m is the length of the pattern. In contrast, the KMP algorithm achieves a linear time complexity of
$O(n+m)$ by efficiently handling mismatches using the LPS array.

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 WiproEP;
public class RabinKarp {
  // Function to search for a pattern in a given text using Rabin-Karp algorithm
  public static void rabinKarpSearch(String text, String pattern) {
      int m = pattern.length();
      int n = text.length();
      int prime = 101; // A prime number to calculate hash values
      int patternHash = 0; // Hash value for the pattern
      int textHash = 0; // Hash value for the text
      int h = 1;
      // The value of h would be "pow(d, m-1)%q"
      for (int i = 0; i < m - 1; i++) {
          h = (h * 256) % prime;
      }
      // Calculate the hash value of the pattern and first window of text
      for (int i = 0; i < m; i++) {
          patternHash = (256 * patternHash + pattern.charAt(i)) % prime;
          textHash = (256 * textHash + text.charAt(i)) % prime;
      }
      // Slide the pattern over text one by one
      for (int i = 0; i <= n - m; i++) {
          // Check the hash values of current window of text and pattern
          if (patternHash == textHash) {
              // Check for characters one by one
              int j;
              for (j = 0; j < m; j++) {
                  if (text.charAt(i + j) != pattern.charAt(j)) {
                      break;
                  }
              }
              // If patternHash == textHash and pattern is found, print the index
              if (j == m) {
                  System.out.println("Pattern found at index " + i);
```

```java
            }
        }
        // Calculate hash value for next window of text
        if (i < n - m) {
            textHash = (256 * (textHash - text.charAt(i) * h) + text.charAt(i + m)) % prime;
            // We might get a negative value of textHash, convert it to positive
            if (textHash < 0) {
                textHash = (textHash + prime);
            }
        }
    }
  }
  public static void main(String[] args) {
      String text = "ABABDABACDABABCABAB";
      String pattern = "ABABCABAB";
      rabinKarpSearch(text, pattern);
  }
}
```

Output

```
Pattern found at index 10
```

Impact on Performance:

- Hash collisions occur when different substrings produce the same hash value. In such cases, even though the hash values match, the actual substrings might not.
- This necessitates an additional character-by-character comparison to verify the match, which can impact performance.

Handling Hash Collisions:

- Hash collisions are handled by the additional comparison step after detecting matching hash values. If the characters do not match, the algorithm continues sliding the window without reporting a false match.
- Using a good hash function and a large prime number reduces the probability of 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.

Code

```
package WiproEP;
public class BoyerMoore {
  // Method to create the bad character heuristic array
  private static int[] createBadCharTable(String pattern) {
      final int ALPHABET_SIZE = 256; // Total number of possible characters
      int[] badCharTable = new int[ALPHABET_SIZE];
      // Initialize all occurrences as -1
      for (int i = 0; i < ALPHABET_SIZE; i++) {
          badCharTable[i] = -1;
      }
      // Fill the actual value of the last occurrence of a character
      for (int i = 0; i < pattern.length(); i++) {
          badCharTable[pattern.charAt(i)] = i;
      }
      return badCharTable;
  }
  // Boyer-Moore search function to find the last occurrence of a pattern in a text
  public static int boyerMooreSearch(String text, String pattern) {
      int m = pattern.length();
      int n = text.length();
      int[] badCharTable = createBadCharTable(pattern);
      int lastIndex = -1;
      int shift = 0;
      while (shift <= (n - m)) {
          int j = m - 1;
          // Reduce the index of j while characters of pattern and text are matching
          while (j >= 0 && pattern.charAt(j) == text.charAt(shift + j)) {
              j--;
          }
          // If the pattern is present at current shift, record the last occurrence
          if (j < 0) {
              lastIndex = shift;
              shift += (shift + m < n) ? m - badCharTable[text.charAt(shift + m)] : 1;
```

```
        } else {
            // Shift the pattern to align the bad character in text with the last occurrence in the pattern
            shift += Math.max(1, j - badCharTable[text.charAt(shift + j)]);
        }
    }
    return lastIndex;
  }
  public static void main(String[] args) {
    String text = "ABABDABACDABABCABAB";
    String pattern = "ABABCABAB";
    int index = boyerMooreSearch(text, pattern);
    System.out.println("Last occurrence of pattern is at index " + index);
  }
}
```

Output

```
Last occurrence of pattern is at index 10
```

Why Boyer-Moore Can Outperform Other Algorithms:

Efficient Shifts:

- The Boyer-Moore algorithm can skip sections of the text, making it potentially much faster than algorithms that check every character.
- The bad character rule and the good suffix rule (not implemented here) allow for large jumps in the text, reducing the number of comparisons.

Pattern Length Influence:

- The algorithm's performance improves with longer patterns since the potential for larger shifts increases, making it particularly efficient for long patterns in long texts.

Optimal for Sparse Matches:

- When matches are infrequent, the Boyer-Moore algorithm performs fewer comparisons than others, making it highly efficient in scenarios where the pattern is not frequently found.