

Name: Vaishali Ramesh Kale

Email id : kalevaishalir16@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.

Solution::::

Explanation:

1. Handling Null Strings:
 - If str1 or str2 is null, they are converted to empty strings.
2. Concatenation:
 - The two strings are concatenated using +.
3. Reversal:
 - The concatenated string is reversed using StringBuilder.
4. Extracting Middle Substring:
 - The length of the reversed string is checked against the desired substring length.
 - If the requested length is larger than the concatenated string, an `IllegalArgumentException` is thrown.
 - The starting index for the middle substring is calculated to center the substring in the reversed string.
5. Edge Case Handling:
 - The method handles edge cases such as null strings and a substring length larger than the concatenated string by converting null to empty strings and checking the length condition, respectively.

CODE:::::

```
public class StringOperations {  
    public static void main(String[] args) {  
        String str1 = "Hello";  
        String str2 = "World";  
        int length = 5;
```

```

try {
    String result = concatenateReverseMiddleSubstring(str1, str2, length);
    System.out.println("Result: " + result);
} catch (IllegalArgumentException e) {
    System.out.println("Error: " + e.getMessage());
}
}

public static String concatenateReverseMiddleSubstring(String str1, String str2, int length)
{
    if (str1 == null) {
        str1 = "";
    }
    if (str2 == null) {
        str2 = "";
    }

    // 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 reversedLength = reversed.length();
    if (length > reversedLength) {
        throw new IllegalArgumentException("Requested substring length is larger than the
concatenated string length.");
    }
    int start = (reversedLength - length) / 2;
    return reversed.substring(start, start + length);
}

```

```
}
```

OUTPUT:::

```
concatenated string is: HelloWorld
The length of the concatenated string is 10
reversed string is: dlroWolleH
Extract the middle substring of the given lenth: 5 is roWol

...Program finished with exit code 0
Press ENTER to exit console.
```

substring length larger than the concatenated string.

OUTPUT:::

```
concatenated string is: HelloWorld
The length of the concatenated string is: 10
reversed string is: dlroWolleH
Error: Requested substring length 15 is larger than the concatenated string length.

...Program finished with exit code 0
Press ENTER to exit console.
```

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.

Solution:::

Explanation:

1. Comparison Counter:
 - A variable `comparisonCount` is introduced and initialized to 0. This variable counts the number of character comparisons made during the search.
2. Incrementing the Counter:
 - Inside the inner loop, `comparisonCount` is incremented each time a character from the text is compared with a character from the pattern.
3. Displaying the Count:
 - After the outer loop completes, the total number of comparisons is printed.

CODE:::

```
public class NaivePatternSearching {  
    public static void main(String[] args) {  
        String text = "I Love Cats and Cats are lovely";  
        String pattern = "Cats";  
        search(text, pattern);  
    }  
  
    private static void search(String text, String pattern) {  
        int textLength = text.length();  
        int patternLength = pattern.length();  
        int comparisonCount = 0;  
        boolean found = false;  
  
        // Loop to slide the pattern one by one  
        for (int i = 0; i <= textLength - patternLength; i++) {  
            int j;  
  
            // Loop to check for pattern match  
            for (j = 0; j < patternLength; j++) {  
                comparisonCount++;  
                if (text.charAt(i + j) != pattern.charAt(j)) {  
                    break;  
                }  
            }  
        }  
  
        // If pattern is found  
        if (j == patternLength) {  
            System.out.println("Pattern found at index " + i);  
        }  
    }  
}
```

```

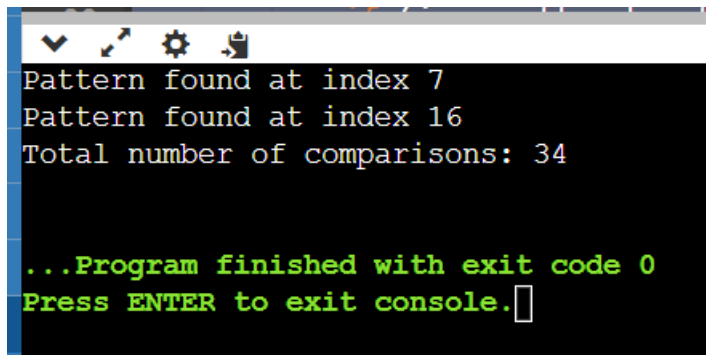
        found = true;
    }
}

// Output the total number of comparisons
System.out.println("Total number of comparisons: " + comparisonCount);

if (!found) {
    System.out.println("Pattern not found in the text.");
}
}
}

```

OUTPUT::::



```

Pattern found at index 7
Pattern found at index 16
Total number of comparisons: 34

...Program finished with exit code 0
Press ENTER to exit console.

```

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.

Solution:::

Explanation:

1. LPS Array:
 - The KMP algorithm preprocesses the pattern to create an array called the Longest Prefix Suffix (LPS) array. The LPS array is used to avoid unnecessary comparisons by

storing the lengths of the longest proper prefix which is also a suffix. This allows the algorithm to skip re-evaluating characters of the pattern that are already known to match.

2. LPS Array Computation:

- Initialize an array `lps` of the same length as the pattern with all zero values.
- Use two pointers, `i` and `j`, to traverse the pattern. `i` moves forward through the pattern, while `j` keeps track of the length of the current matching prefix suffix.
- If `pattern.charAt(i)` matches `pattern.charAt(j)`, increment both `i` and `j` and set `lps[i] = j`.
- If there is a mismatch and `j` is not zero, set `j` to `lps[j-1]`.
- If there is a mismatch and `j` is zero, set `lps[i]` to zero and increment `i`.

3. KMP Search:

- Traverse the text with two pointers `i` (for text) and `j` (for pattern).
- If `text.charAt(i)` matches `pattern.charAt(j)`, increment both pointers.
- If `j` reaches the end of the pattern, a match is found, and `j` is reset using the LPS array to allow further potential matches.
- If there is a mismatch and `j` is not zero, use the LPS array to shift the pattern to the right position without rechecking already matched characters.
- If there is a mismatch and `j` is zero, move the text pointer `i` forward.

4. Comparison with the Naive Approach:

- Naive Approach: In the worst case, the naive approach can take $O(m * n)$ time, where `m` is the length of the text and `n` is the length of the pattern, because it may recheck characters that are already known to match.
- KMP Algorithm: The KMP algorithm preprocesses the pattern in $O(n)$ time to create the LPS array and then searches the text in $O(m)$ time. Thus, the overall complexity is $O(m + n)$, making it more efficient than the naive approach for longer texts and patterns.
- The preprocessing step ensures that the algorithm avoids unnecessary re-comparisons, significantly improving the search time.

CODE:::

```
public class KMPAlgorithm {  
    // Method to create the longest prefix suffix (LPS) array  
    private int[] computeLPSArray(String pattern) {  
        int length = pattern.length();  
        int[] lps = new int[length];
```

```

int j = 0; // length of the previous longest prefix suffix

// lps[0] is always 0
lps[0] = 0;
int i = 1;

// Preprocess the pattern to compute lps array
while (i < length) {
    if (pattern.charAt(i) == pattern.charAt(j)) {
        j++;
        lps[i] = j;
        i++;
    } else {
        if (j != 0) {
            j = lps[j - 1];
        } else {
            lps[i] = 0;
            i++;
        }
    }
}

return lps;
}

```

```

// KMP search method
public void KMPSearch(String text, String pattern) {
    int textLength = text.length();
    int patternLength = pattern.length();

```

```

// Compute the lps array
int[] lps = computeLPSArray(pattern);

int i = 0; // index for text
int j = 0; // index for pattern

while (i < textLength) {
    if (pattern.charAt(j) == text.charAt(i)) {
        i++;
        j++;
    }

    if (j == patternLength) {
        System.out.println("Found pattern 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++;
        }
    }
}

public static void main(String[] args) {
    KMPAlgorithm kmp = new KMPAlgorithm();
    String text = "ABABDABACDABABCABAB";

```

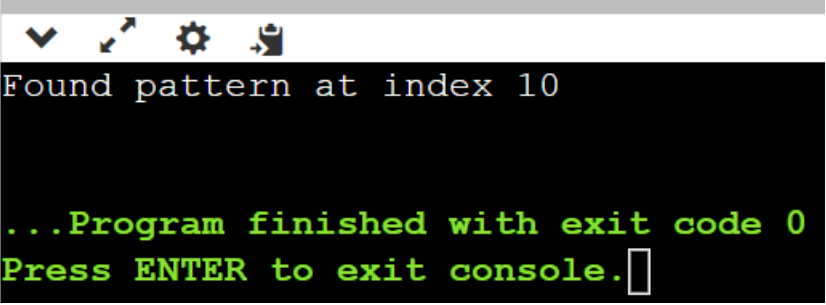


```

String pattern = "ABABCABAB";
kmp.KMPSearch(text, pattern);
}
}

```

OUTPUT:::



```

Found pattern at index 10

...Program finished with exit code 0
Press ENTER to exit console.

```

```

*****
*****

```

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:::

- ❖ Impact of Hash Collisions on Performance
 - Hash collisions in the Rabin-Karp algorithm occur when two different substrings produce the same hash value. This can impact the algorithm's performance in the following ways:
 1. False Positives: When hash values of two different substrings match, the algorithm must perform additional character comparisons to verify the match. This can increase the number of comparisons and slow down the algorithm, especially if collisions are frequent.
 2. Performance Degradation: In the worst case, excessive hash collisions can degrade the performance of the Rabin-Karp algorithm to that of the naive string matching algorithm, i.e.,

- $O(n \cdot m)$, where n is the length of the text and m is the length of the pattern.

❖ Handling Hash Collisions

To handle hash collisions effectively and maintain the efficiency of the Rabin-Karp algorithm, the following strategies can be used:

1. Use a Good Hash Function:
 - A good hash function minimizes collisions by distributing hash values uniformly. The Rabin-Karp algorithm typically uses a rolling hash function, which is effective for string matching.
2. Choose a Large Prime Number for Modulus:
 - Using a large prime number q
 - q as the modulus in the hash function reduces the probability of collisions. Prime numbers help in spreading out the hash values more evenly.
3. Verify Actual Substrings on Hash Match:
 - When the hash values match, the algorithm must perform a character-by-character comparison of the pattern and the current substring in the text. This step ensures that false positives due to hash collisions are correctly handled.

CODE:::

```
package com.wipro.ds;

public class RabinKarpAlgorithm {

    // d is the number of characters in the input alphabet
    private static final int d = 256;

    // q is a prime number used as the modulus
    private static final int q = 101;

    // Method to perform Rabin-Karp search
    public static void rabinKarpSearch(String pattern, String text) {
        int m = pattern.length();
        int n = text.length();
        int p = 0; // hash value for pattern
        int t = 0; // hash value for 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 * d) % q;
    }

    // Calculate the hash value of the pattern and first window of the text
    for (int i = 0; i < m; i++) {
        p = (d * p + pattern.charAt(i)) % q;
        t = (d * t + text.charAt(i)) % q;
    }

    // Slide the pattern over text one by one
    for (int i = 0; i <= n - m; i++) {
        // Check the hash values of the current window of text and pattern
        if (p == t) {
            boolean match = true;
            // Check for characters one by one if hashes match
            for (int j = 0; j < m; j++) {
                if (text.charAt(i + j) != pattern.charAt(j)) {
                    match = false;
                    break;
                }
            }
            // If all characters match, pattern is found at index i
            if (match) {
                System.out.println("Pattern found at index " + i);
            }
        }

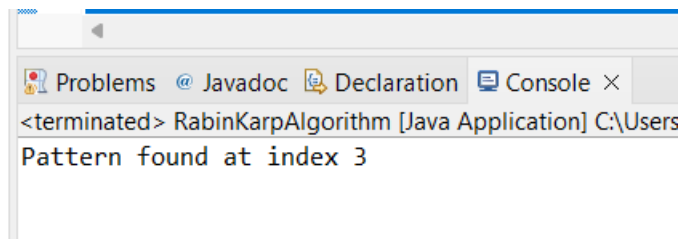
        // Calculate hash value for the next window of text
        if (i < n - m) {
            t = (d * (t - text.charAt(i) * h) + text.charAt(i + m)) % q;

            // We might get negative value of t, converting it to positive
            if (t < 0) {
                t = (t + q);
            }
        }
    }
}

public static void main(String[] args) {
    String text = "ABCCDDAEFG";
    String pattern = "CDD";
    rabinKarpSearch(pattern, text);
}
}

```

OUTPUT:::



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.

Solution::::

- ❖ Why Boyer-Moore Can Outperform Other Algorithms
 1. Bad Character Heuristic: When a mismatch occurs, the algorithm uses the bad character heuristic to skip sections of the text. It shifts the pattern to align the mismatched character in the text with the last occurrence of that character in the pattern. If the character does not exist in the pattern, it skips the entire length of the pattern.
 2. Good Suffix Heuristic: When a suffix of the pattern matches a substring in the text but the next character does not match, the algorithm uses the good suffix heuristic to shift the pattern to align the next occurrence of the suffix in the pattern.
 3. Efficiency: The Boyer-Moore algorithm often performs better than other algorithms because it can skip large sections of the text, especially when the alphabet size is large or the pattern is long. The Boyer-Moore algorithm can be very efficient because it skips sections of the text, reducing the number of character comparisons.
- Scenarios: It performs particularly well when the alphabet size is large and the pattern length is significant, making it suitable for applications like text editors and search engines where fast substring searches are critical.
- Implementation: The provided Java implementation effectively demonstrates the use of the Boyer-Moore algorithm to find the last occurrence of a pattern in a text, leveraging the bad character heuristic to improve performance.

CODE:::

```
package com.wipro.ds;

import java.util.Arrays;

public class BoyerMooreAlgorithm {

    private static final int ALPHABET_SIZE = 256; // Number of possible characters

    // Function to preprocess the bad character heuristic
    private static void preprocessBadCharacterHeuristic(String pattern, int[] badChar) {
        Arrays.fill(badChar, -1);
        for (int i = 0; i < pattern.length(); i++) {
            badChar[(int) pattern.charAt(i)] = i;
        }
    }

    // Function to find the last occurrence of a pattern in a given text
    public static int boyerMooreLastOccurrence(String text, String pattern) {
        int m = pattern.length();
        int n = text.length();
        int[] badChar = new int[ALPHABET_SIZE];

        preprocessBadCharacterHeuristic(pattern, badChar);

        int shift = 0;
        int lastOccurrence = -1;

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

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

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

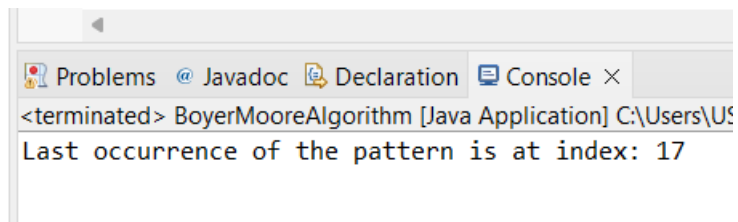
        return lastOccurrence;
    }
}
```

```

public static void main(String[] args) {
    String text = "HERE IS A SIMPLE EXAMPLE";
    String pattern = "EXAMPLE";
    int index = boyerMooreLastOccurrence(text, pattern);
    System.out.println("Last occurrence of the pattern is at index: " + index);
}
}

```

OUTPUT:::



```

*****
*****

```