DAY-11

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)

```java
package WiprpTask;
public class Middle {
  public static String getMiddleSubstring(String str1, String str2, int length) {
     if (str1 == null || str2 == null || length <= 0) {
        return "";
     }
     String concatenated = str1 + str2;
     String reversed = new StringBuilder(concatenated).reverse().toString();
     if (length > reversed.length()) {
        return "";
     }
     int startIndex = (reversed.length() - length) / 2;
     return reversed.substring(startIndex, startIndex + length);
  }
  public static void main(String[] args) {
     String str1 = "Sayan";
     String str2 = "Mridha";
     int length = 5;
     String middleSubstring = getMiddleSubstring(str1, str2, length);
     System.out.println("Middle substring: " + middleSubstring);
     str1 = "";
     str2 = "Mridha";
     length = 5;
     middleSubstring = getMiddleSubstring(str1, str2, length);
     System.out.println("Middle substring: " + middleSubstring);
     str1 = "Sayan";
     str2 = "";
     length = 5;
     middleSubstring = getMiddleSubstring(str1, str2, length);
     System.out.println("Middle substring: " + middleSubstring);
     str1 = "Sayan";
     str2 = "Mridha";
     length = 10;
     middleSubstring = getMiddleSubstring(str1, str2, length);
     System.out.println("Middle substring: " + middleSubstring);
  }
}
```

OUTPUT:-

```
Middle substring: irMna
Middle substring: ahdir
Middle substring: nayaS
Middle substring: ahdirMnaya
```

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)

Code:-

```
package com.wipro.patterns;
public class NaiveStringSearch {
  public static void main(String[] args) {
     String text = "I Love Cats";
     String pattern = "Cats";
     int result = naive(text, pattern);
     if (result!= -1) {
        System.out.println("Pattern found at index " + result);
     } else {
        System.out.println("Pattern not found");
     }
  }
  private static int naive(String string, String pattern) {
     int i = 0;
     int j = 0;
     if (string.length() < pattern.length()) {
        return -1;
     }
     while (i < string.length()) {
        if (string.charAt(i) == pattern.charAt(j)) {
           j++;
        } else {
           i = i - j;
           j = 0;
        }
        System.out.println(i + " " + j);
        if (j == pattern.length()) {
           return i - pattern.length() + 1;
        }
        i++;
     }
     return -1;
  }
```

```
}
```

OUTPUT:-

```
0 0
1 0
2 0
3 0
4 0
5 0
6 0
7 1
8 2
9 3
10 4
Pattern found at index 7
```

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.
Ans)
Code:-

```
package WiprpTask;
public class KMPAlgorithm {
  public static int[] computeLPSArray(String pattern) {
     int length = 0;
     int i = 1;
     int[] lps = new int[pattern.length()];
     lps[0] = 0;
     while (i < pattern.length()) {
        if (pattern.charAt(i) == pattern.charAt(length)) {
           length++;
           lps[i] = length;
           i++;
        } else {
           if (length != 0) {
              length = lps[length - 1];
           } else {
              lps[i] = 0;
              i++;
           }
        }
     }
     return lps;
  }
```

```
  public static void KMPSearch(String text, String pattern) {
      int[] lps = computeLPSArray(pattern);
      int i = 0;
      int j = 0;
      while (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 (i < text.length() && pattern.charAt(j) != text.charAt(i)) {
              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
```

The Boyer-Moore algorithm improves search time over the naive approach through two key preprocessing steps: the bad character heuristic and the good suffix heuristic. These steps allow the algorithm to skip sections of the text that do not match the pattern, rather than checking each character in the text sequentially. Here's how these heuristics work and why they improve the search time:

Bad Character Heuristic

1. Preprocessing Step:
    - The algorithm creates a "bad character table" that stores the last occurrence of each character in the pattern.

For example, if the pattern is "ABCD", the table would store:
rust

```
A -> 0
B -> 1
C -> 2
D -> 3
```

    -
2. During Search:
    - When a mismatch occurs, the algorithm uses this table to determine how far to shift the pattern.
    - If a character in the text does not match the current character in the pattern, the pattern is shifted so that the last occurrence of this mismatched character in the pattern aligns with the mismatched character in the text. If the character is not present in the pattern, the pattern is shifted past the mismatched character.
    - This heuristic allows the pattern to skip sections of the text, avoiding unnecessary comparisons.

Good Suffix Heuristic

1. Preprocessing Step:
    - The algorithm creates a "good suffix table" that stores how far the pattern can be shifted when a suffix (a substring at the end) of the pattern matches part of the text but the next character in the pattern does not match.
    - For example, if the pattern is "ABCDABC", the table might store shifts for suffixes like "ABC", "BC", and "C".
2. During Search:
    - When a mismatch occurs after some characters have matched, the algorithm uses this table to determine the shift.
    - If a suffix of the pattern matches a part of the text but the following character does not, the pattern is shifted so that the next occurrence of this suffix aligns with the text.
    - This heuristic also allows the pattern to skip sections of the text, avoiding redundant checks.

Comparison to the Naive Approach

- Naive Approach:

    - The naive approach checks every possible position of the pattern in the text, one character at a time.
    - For each position in the text, it compares each character in the pattern sequentially.
    - This results in a worst-case time complexity of O(m⋅n)O(m \cdot n)O(m⋅n), where mmm is the length of the text and nnn is the length of the pattern.
- Boyer-Moore Algorithm:
    - The Boyer-Moore algorithm uses the bad character and good suffix heuristics to skip sections of the text.
    - In the best case, the pattern can be shifted significantly further ahead, leading to fewer comparisons.
    - This results in an average-case time complexity that is sub-linear with respect to the length of the text, making it much more efficient for large texts.

Practical Impact

- Skips Non-Matching Sections: The bad character heuristic allows the algorithm to skip sections of the text where a mismatch is found, instead of checking each character sequentially.
- Uses Matched Information Efficiently: The good suffix heuristic allows the algorithm to make use of previously matched portions of the text to skip ahead, further reducing unnecessary comparisons.
- Improved Performance in Real-World Scenarios: In practice, these heuristics often result in the Boyer-Moore algorithm being significantly faster than the naive approach, especially for large texts and longer patterns.

By using these preprocessing steps, the Boyer-Moore algorithm can significantly reduce the number of character comparisons needed to find a pattern in a text, leading to improved search times compared to the naive approach.

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)
Code:-

```java
package com.wipro.patterns;
public class RabinKarp {
  public static final int PRIME = 101;
  public static void search(String pat, String txt) {
    int M = pat.length();
    int N = txt.length();
    long patHash = createHash(pat, M);
    long txtHash = createHash(txt, M);
    System.out.println("Hash value of pattern: " + patHash);
    for (int i = 0; i <= N - M; i++) {
      System.out.println("Hash value of text (window of size " + M + " starting at
index " + i + "): " + txtHash);
      if (patHash == txtHash && checkEqual(txt, i, i + M - 1, pat, 0, M - 1)) {
        System.out.println("Pattern found at index " + i);
      }
      if (i < N - M) {
        txtHash = recalculateHash(txt, i, i + M, txtHash, M);
      }
    }
  }
  public static long createHash(String str, int end) {
    long hash = 0;
    for (int i = 0; i < end; i++) {
      hash += str.charAt(i) * Math.pow(PRIME, i);
    }
    return hash;
  }
  public static long recalculateHash(String str, int oldIndex, int newIndex, long
oldHash, int patternLen) {
    long newHash = oldHash - str.charAt(oldIndex);
    newHash /= PRIME;
    newHash += str.charAt(newIndex) * Math.pow(PRIME, patternLen - 1);
    return newHash;
  }
  public static boolean checkEqual(String str1, int start1, int end1, String str2, int
start2, int end2) {
    if (end1 - start1 != end2 - start2) {
      return false;
    }
    while (start1 <= end1 && start2 <= end2) {
      if (str1.charAt(start1) != str2.charAt(start2)) {
        return false;
      }
      start1++;
      start2++;
    }
    return true;
  }
```

```
  public static void main(String[] args) {
      String pat = "Sayan";
      String txt = "My name is Sayan Mridha";
      search(pat, txt);
  }
}
```

OUTPUT:-

```
Hash value of pattern: 11547827508
Hash value of text (window of size 5 starting at index 0): 10207530737
Hash value of text (window of size 5 starting at index 1): 11443648369
Hash value of text (window of size 5 starting at index 2): 10623403949
Hash value of text (window of size 5 starting at index 3): 3435115049
Hash value of text (window of size 5 starting at index 4): 10960353144
Hash value of text (window of size 5 starting at index 5): 12075464462
Hash value of text (window of size 5 starting at index 6): 3449491885
Hash value of text (window of size 5 starting at index 7): 8671166667
Hash value of text (window of size 5 starting at index 8): 10179712032
Hash value of text (window of size 5 starting at index 9): 12692097748
Hash value of text (window of size 5 starting at index 10): 10219523230
Hash value of text (window of size 5 starting at index 11): 11547827508
Pattern found at index 11
Hash value of text (window of size 5 starting at index 12): 3444267757
Hash value of text (window of size 5 starting at index 13): 8046752537
Hash value of text (window of size 5 starting at index 14): 11942556530
Hash value of text (window of size 5 starting at index 15): 11044585238
Hash value of text (window of size 5 starting at index 16): 10515392428
Hash value of text (window of size 5 starting at index 17): 10926394500
Hash value of text (window of size 5 starting at index 18): 10202041020
```

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.

Ans)

Code:-

```
package com.wipro.patterns;
public class BoyerMoore {
  public static int lastIndexOf(String text, String pattern) {
      int[] badChar = preprocessBadCharacter(pattern);
      int[] goodSuffix = preprocessGoodSuffix(pattern);
      int textLength = text.length();
      int patternLength = pattern.length();
      int s = textLength - 1;
      int lastIndex = -1;
      while (s >= patternLength - 1) {
```

```
        int j = patternLength - 1;
        while (j >= 0 && pattern.charAt(j) == text.charAt(s)) {
            j--;
            s--;
        }
        if (j < 0) {
            lastIndex = s + 1;
            s -= patternLength;
        } else {
            s -= Math.max(1, Math.max(j - badChar[text.charAt(s)], goodSuffix[j]));
        }
    }
    return lastIndex;
}
private static int[] preprocessBadCharacter(String pattern) {
    int[] badChar = new int[256];
    for (int i = 0; i < 256; i++) {
        badChar[i] = -1;
    }
    for (int i = 0; i < pattern.length(); i++) {
        badChar[pattern.charAt(i)] = i;
    }
    return badChar;
}
private static int[] preprocessGoodSuffix(String pattern) {
    int patternLength = pattern.length();
    int[] goodSuffix = new int[patternLength];
    int[] borderPosition = new int[patternLength + 1];
    int i = patternLength;
    int j = patternLength + 1;
    borderPosition[i] = j;
    while (i > 0) {
        while (j <= patternLength && pattern.charAt(i - 1) != pattern.charAt(j - 1)) {
            if (j < patternLength && goodSuffix[j] == 0) {
                goodSuffix[j] = j - i;
            }
            j = borderPosition[j];
        }
        i--;
        j--;
        borderPosition[i] = j;
    }
    j = borderPosition[0];
    for (i = 0; i <= patternLength; i++) {
        if (i < patternLength && goodSuffix[i] == 0) {
            goodSuffix[i] = j;
        }
        if (i == j) {
```

```
            j = borderPosition[j];
        }
    }
    return goodSuffix;
  }
  public static void main(String[] args) {
    String text = "My name is Sayan Mridha and I am Sayan";
    String pattern = "Sayan";
    int index = lastIndexOf(text, pattern);
    System.out.println("Last occurrence of pattern '" + pattern + "' is at index: " +
index);
  }
}
```

OUTPUT:-

```
Last occurrence of pattern 'Sayan' is at index: 33
```