Assignment – 14

Prakash Manikanta Irrinki
Prakashnaidu9494@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.

Program:

```
public class StringOperations {

      public static String afterReverseSubString(String str1,
                                 String str2, int lenSubString) {
           String concatStr = str1 + str2;
           String reverseStr = new
                       StringBuffer(concatStr).reverse().toString();
           if (lenSubString > reverseStr.length()) {
                 return reverseStr;
           }
           int midindex = reverseStr.length() / 2;
           int startindex = Math.max(0, midindex-(lenSubString/2));
           int endindex = startindex + lenSubString;
           if (endindex > reverseStr.length()) {
                 endindex = reverseStr.length();
                 startindex = Math.max(0, endindex - lenSubString);
           }
           String midsubstring = reverseStr.substring(startindex,
                                                        endindex);
           return midsubstring;
      }
      public static void main(String[] args) {

           System.out.println(afterReverseSubString("hello",
"world!", 4));
           System.out.println(afterReverseSubString("prakash",
"manikanta", 2));
           System.out.println(afterReverseSubString("manohar",
"friend", 6));
           System.out.println(afterReverseSubString("frind",
"firoz", 3));
           System.out.println(afterReverseSubString("teja", "pavan",
1));

      }
}
```

Output:

```
rowo
am
irfrah
fdn
p
```

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.

Program:

```
public class NaivePattern {

    public static void searchPattern(String text,String pattern) {
        int textlen=text.length();
        int patternlen=pattern.length();
        int count=0;
        for(int i=0;i<=textlen-patternlen;i++) {
            int j;
            for(j=0;j<patternlen;j++) {
                count++;
                if(text.charAt(i+j) != pattern.charAt(j)) {
                    break;
                }
            }
            if(j == patternlen) {
                System.out.println("Pattern found at index:
                                             "+i);
            }
        }
        System.out.println("Total comparasions made: "+count);
    }

    public static void main(String[] args) {

        String text="This is the text naive pattern";
        String pattern="naive";

        searchPattern(text, pattern);

    }

}
```

Output:

```
Pattern found at index: 17
Total comparasions 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.

Program:

```
public class KMPAlgorithm {

     public static int[] computeLPSArray(String pattern,int m) {
          int[] lps=new int[m];
          int length=0;
          int i=1;
          lps[0]=0;
          while(i<m) {
               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 n=text.length();
          int m=pattern.length();
          int[] lps=computeLPSArray(pattern, m);
          int j=0;
          int count=0;
          int i=0;
          while(i<n) {
               count++;
               if(pattern.charAt(j)==text.charAt(i)) {
                    j++;
                    i++;
               }
               if(j==m) {
                    System.out.println("Pattern found at index:
                                            "+(i-j));
                    j=lps[j-1];
               }
               else if(i<n && pattern.charAt(j) != text.charAt(i)){
                    if(j !=0) {
```

```
                    j=lps[j-1];
                }else {
                    i++;
                }
            }
        }
        System.out.println("Total comparisons made: "+count);
    }

    public static void main(String[] args) {

        String text="This is the kmp search algorithm";
        String pattern="search";

        kMPSearch(text, pattern);

    }

}
```

Output:

```
Pattern found at index: 16
Total comparisons made: 32
```

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.

Program:

```
public class RabinKarpAlgorithm {

    private static final int NUM = 101;

    public static void search(String text, String pattern) {
       int n = text.length();
       int m = pattern.length();
       int patternHash = cHash(pattern, m);
       int textHash = cHash(text, m);
       int h = 1;
       for (int i = 0; i < m - 1; i++) {
           h = (h * NUM);
       }
       int count = 0;
       for (int i = 0; i <= n - m; i++) {
           if (patternHash == textHash) {
               count++;
               boolean match = true;
               for (int j = 0; j < m; j++) {
                        count++;
                   if (text.charAt(i + j) != pattern.charAt(j)) {
                       match = false;
                       break;
                        }
               }
```

```
                    if (match) {
                    System.out.println("Pattern found at index: "
                                           + i);
                }
            }
            if (i < n - m) {
                textHash = rehash(text, textHash, i, m);
            }
        }
        System.out.println("Total comparisons made: " + count);
     }


     private static int cHash(String pattern, int m) {
           int hash = 0;
        for (int i = 0; i < m; i++) {
            hash += pattern.charAt(i) * Math.pow(NUM, i);
        }
           return hash;
     }


     private static int rehash(String text, int textHash, int i,
                                       int m) {
           int newtextHash = textHash - text.charAt(i);
        newtextHash = newtextHash / NUM;
        newtextHash += text.charAt(i + m) * Math.pow(NUM, m - 1);
           return newtextHash;
     }


     public static void main(String[] args) {
           String text="This is the java rabin karp search
                           algorithm java";
           String pattern="java";

           search(text, pattern);

     }

}
```

Output:

```
Pattern found at index: 12
Pattern found at index: 45
Total comparisons made: 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.

Program:

```
public class BooyerMooreAlgorithm {
      private static void preprocessBadCharacter(String pattern,
                                                      int[] badChar) {
            int m = pattern.length();
            for (int i = 0; i < 256; i++) {
                  badChar[i] = -1;
            }
            for (int i = 0; i < m; i++) {
                  badChar[(int) pattern.charAt(i)] = i;
            }
      }
      private static int lastOccurrence(String text, String pattern)
      {
            int n = text.length();
            int m = pattern.length();
            int[] badChar = new int[256];
            preprocessBadCharacter(pattern, badChar);
            int s = 0;
            int lastIndex = -1;
            while (s <= (n - m)) {
                  int j = m - 1;
                  while (j >= 0 && pattern.charAt(j) == text.charAt(
                              s + j)) {
                        j--;
                  }
                  if (j < 0) {
                        lastIndex = s;
                        s += (s + m < n) ? m - badChar[text.charAt(
                                    s + m)] : 1;
                  } else {
                        s += Math.max(1, j - badChar[text.charAt(
                                          s + j)]);
                  }
            }
            return lastIndex;
      }
      public static void main(String[] args) {
            String text = "This is the booyar moore algorithm";
            String pattern = "moore";

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

Output:

```
Last occurrence of pattern is at index 19
```