| Experiment **#15** | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT] THANOS |

**Experiment Title: To implement programs on String Matching Algorithms.**

**Aim/Objective:** To understand the concept and implementation of programs on String Matching Algorithms.

**Description:** The students will understand the programs on Knuth-Morris-Pratt Algorithm, Rabin-Karp Algorithm, Boyer-Moore Algorithm, applying them to solve real-world problems.

**Pre-Requisites:**

**Knowledge:** Knuth-Morris-Pratt Algorithm, Rabin-Karp Algorithm, Boyer-Moore Algorithm.

**Tools:** Code Blocks/Eclipse IDE.

**Pre-Lab:**

Given two strings text and pattern, implement the Knuth-Morris-Pratt algorithm to determine the starting indices of all occurrences of pattern in text. If the pattern is not found, return an empty list.

**Input Format:**

- A string text of length n ($1 \le n \le 10^6$).
- A string pattern of length m ($1 \le m \le 10^5$).

**Output Format:**

- A list of integers representing the starting indices (0-based) of all occurrences of pattern in text.

**Constraints:**

- Both text and pattern consist of lowercase English letters.

**Sample Input:**

text: " ababcabcabababd"

pattern: "ababd"

**Sample Output:**

[10]

- **Procedure/Program:**

```
public class Main {

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

    while (i < m) {
      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 KMPSearch(String text, String pattern) {
    int n = text.length();
    int m = pattern.length();
    int[] lps = new int[m];
    computeLPSArray(pattern, m, lps);

    int i = 0;
    int j = 0;
    boolean found = false;
```

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

            if (j == m) {
                System.out.print((i - j) + " ");
                found = true;
                j = lps[j - 1];
            } else if (i < n && pattern.charAt(j) != text.charAt(i)) {
                if (j != 0) {
                    j = lps[j - 1];
                } else {
                    i++;
                }
            }
        }

        if (!found) {
            System.out.print("[]");
        }
    }

    public static void main(String[] args) {
        String text = "ababcabcabababd";
        String pattern = "ababd";
        KMPSearch(text, pattern);
    }
}
```

- **Data and Results:**

**Data:** KMP algorithm efficiently finds pattern occurrences using prefix table computation.

**Result:** Pattern found at specific indices or returns empty brackets if absent.

- **Analysis and Inferences:**

**Analysis:** Uses LPS array to optimize searching, reducing redundant comparisons.

**Inferences:** Efficient for long texts, but preprocessing LPS adds slight overhead.

**In-Lab:**

Write a function to find the occurrences of a pattern in a given string text using the Rabin-Karp algorithm. The function should return all starting indices of pattern in text.

**Input Format:**

- A string text of length n ($1 \le n \le 10^6$).
- A string pattern of length m ($1 \le m \le 10^5$).

**Output Format:**

- A list of integers representing the starting indices (0-based) of all occurrences of pattern in text.

**Constraints:**

- Both text and pattern consist of lowercase English letters.
- Use modular arithmetic to avoid integer overflow.

**Sample Input:**

text: "abracadabra"

pattern: "abra"

**Sample Output:**

[0,7]

- **Procedure/Program:**

```
import java.util.ArrayList;

public class Main {
    static final int d = 256;
    static final int q = 101;

    public static void rabinKarp(String text, String pattern, ArrayList<Integer> result) {
        int n = text.length();
        int m = pattern.length();
        int p = 0;
        int t = 0;
        int h = 1;

        for (int i = 0; i < m - 1; i++)
            h = (h * d) % q;

        for (int i = 0; i < m; i++) {
            p = (d * p + pattern.charAt(i)) % q;
            t = (d * t + text.charAt(i)) % q;
        }

        for (int i = 0; i <= n - m; i++) {
            if (p == t) {
                int j;
                for (j = 0; j < m; j++) {
                    if (text.charAt(i + j) != pattern.charAt(j))
                        break;
                }
                if (j == m) {
                    result.add(i);
                }
            }
```

```
          if (i < n - m) {
            t = (d * (t - text.charAt(i) * h) + text.charAt(i + m)) % q;

            if (t < 0)
                t = t + q;
          }
      }
  }

  public static void main(String[] args) {
      String text = "abracadabra";
      String pattern = "abra";
      ArrayList<Integer> result = new ArrayList<>();

      rabinKarp(text, pattern, result);

      System.out.print("[");
      for (int i = 0; i < result.size(); i++) {
          System.out.print(result.get(i));
          if (i < result.size() - 1) {
              System.out.print(",");
          }
      }
      System.out.println("]");
  }
}
```

- **Data and Results:**

**Data:** Rabin-Karp algorithm searches pattern occurrences in given text efficiently.

**Result:** Pattern found at specific indices, stored in result array dynamically.

- **Analysis and Inferences:**

**Analysis:** Rolling hash technique minimizes comparisons, improving search speed significantly.

**Inferences:** Efficient for multiple pattern searches, but hash collisions may occur.

| | | | |
|---|---|---|---|
| Experiment **#15** | | Student ID | |
| Date | | Student Name | [@KLWKS_BOT] THANOS |

- **Sample VIVA-VOCE Questions (In-Lab):**

1. When would you choose KMP over Rabin-Karp or Boyer-Moore?

Use KMP for multiple patterns, worst-case efficiency.

2. How do Rabin-Karp and Boyer-Moore handle patterns with repetitive characters?

Rabin-Karp: more collisions; Boyer-Moore: weaker shifts.

3. What happens if the pattern length is longer than the text?

No match, terminates.

4. How would you test the performance of Boyer-Moore on different input patterns?

Vary patterns, analyze cases.

5. What potential errors can occur while implementing Rabin-Karp with modular arithmetic?

Overflow, negative hash issues.

| | |
|---|---|
| **Evaluator Remark (if Any):** | **Marks Secured ___out of 50** |
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.**