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

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:**

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void computeLPSArray(char* pattern, int m, int* lps) {
   int len = 0;
   lps[0] = 0;
   int i = 1;

   while (i < m) {
      if (pattern[i] == pattern[len]) {
         len++;
         lps[i] = len;
         i++;
      } else {
         if (len != 0) {
            len = lps[len - 1];
         } else {
            lps[i] = 0;
            i++;
         }
      }
   }
}

void KMPSearch(char* text, char* pattern) {
   int n = strlen(text);
   int m = strlen(pattern);
   int* lps = (int*)malloc(m * sizeof(int));
   computeLPSArray(pattern, m, lps);

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

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

    if (j == m) {
      printf("%d ", i - j);
      found = 1;
      j = lps[j - 1];
    } else if (i < n && pattern[j] != text[i]) {
      if (j != 0) {
        j = lps[j - 1];
      } else {
        i++;
      }
    }
  }

  if (!found) {
    printf("[]");
  }

  free(lps);
}

int main() {
  char text[] = "ababcabcabababd";
  char pattern[] = "ababd";
  KMPSearch(text, pattern);
  return 0;
}
```

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

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:**

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define d 256
#define q 101

void rabinKarp(char *text, char *pattern, int *result, int *count) {
   int n = strlen(text);
   int m = strlen(pattern);
   int i, j;
   int p = 0;
   int t = 0;
   int h = 1;


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


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

   for (i = 0; i <= n - m; i++) {

      if (p == t) {

         for (j = 0; j < m; j++) {
            if (text[i + j] != pattern[j])
               break;
         }
         if (j == m) {
            result[(*count)++] = i;
```

```
        }
      }


      if (i < n - m) {
        t = (d * (t - text[i] * h) + text[i + m]) % q;


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

int main() {
    char text[] = "abracadabra";
    char pattern[] = "abra";
    int result[100];
    int count = 0;

    rabinKarp(text, pattern, result, &count);

    printf("[");
    for (int i = 0; i < count; i++) {
      printf("%d", result[i]);
      if (i < count - 1) {
        printf(",");
      }
    }
    printf("]\n");

    return 0;
}
```

- **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.

- **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.