|  |  |  |  |
| --- | --- | --- | --- |
| **Academic Year** | **2025 - 26** | **Experiment No.** | **9** |
| **Course & Semester** | **S.E. – Sem. III** | **Subject Name** | **Analysis of Algorithm** |
| **Experiment Type** | **Software Performance** | **Subject Code** | **25PCC12CS05** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name of Student:** | Atharva Dharmendra Jagtap | **Roll No.:** | 10937 |
| **Date of Performance:** | 13/10/2025 | **Date of Submission:** | 13/10/2025 |
| **LO Mapping** | 25PCC12CS05.1: Analyze the time and space complexity of algorithms.  25PCC12CS05.6:Apply various string-matching algorithms to solve pattern matching problems. | | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | **Indicator** | **Poor** | **Average** | **Good** | | Timeline Maintains submission deadline (3) | Submission not done (0) | One or More than One week late (1-2) | Maintains deadline (3) | | Completion and Organization (3) | N/A | Document is just acceptable (1-2) | Completed whole document and neatly organized (3) | | Program Performance (2) | Could not perform at all (0) | Implemented few parts (1) | Full implementation (2) | | Knowledge In depth knowledge of the Experiment (2) | Unable to answer questions (0) | Unable to answer few questions (1) | Able to answer all questions (2) | |
| **Assessment Marks:**   |  |  | | --- | --- | | Timeline |  | | Completion and Organization |  | | Program Performance |  | | Knowledge |  | |
| Total: (Out of 10) |
| Teacher’s Sign: Student Sign: |

**Experiment No. 9**

**AIM:**Identify and implement an algorithm to be used by search engines to quickly locate documents containing specific keywords or phrases, improving search efficiency and response time.

**THEORY:**

In search engines, quickly locating documents containing specific keywords is essential to improve efficiency and response time. The Rabin-Karp algorithm is an effective technique for this purpose. It uses a hashing method to find patterns (keywords) in a large text (documents) efficiently.

The Rabin-Karp algorithm computes a hash value for the pattern and compares it with the hash values of substrings in the text. If the hash values match, a character-by-character check is performed to confirm the match. This reduces the number of unnecessary comparisons, making the search process faster.

**Time Complexity:**

* Best Case: O(n + m) — when there are few or no hash collisions.
* Worst Case: O(n \* m) — when many hash collisions occur. Here, n is the length of the text, and m is the length of the pattern.

Thus, Rabin-Karp is a highly efficient choice for string matching in search engine applications, especially when multiple patterns are involved.

**ALGORITHM:**

1. **Input**:
   * text: The text (document) in which we need to search the pattern.
   * pattern: The pattern (keyword) to search for in the text.
2. **Initialization**:
   * Set the alphabet size d = 256 (for ASCII characters).
   * Choose a prime number q = 101 for hash function to avoid collisions.
   * Calculate the length of the pattern as m and the length of the text as n.
3. **Compute Hashes**:
   * Calculate the hash value of the pattern using a rolling hash function.
   * Calculate the hash value of the first m characters of the text (this will be the initial window).
4. **Sliding Window**:
   * Slide a window of size m over the text from index 0 to n - m.
   * For each window, compare the hash value of the substring with the hash value of the pattern.
     + If the hash values match, perform a character-by-character comparison to confirm the match.
5. **Recalculate Hash for Next Window**:
   * After checking the current window, calculate the hash of the next window by removing the old leading character and adding the new character at the end, using the formula:

new\_hash = (d \* (old\_hash - text[i] \* d^(m-1)) + text[i+m]) % q

Where i is the starting index of the current window, and d^(m-1) is precomputed for efficiency.

1. **Output**:
   * If a match is found (i.e., pattern hash matches the window hash and character-by-character comparison succeeds), output the index where the match is found.

**Best Case Time Complexity**: **O(n + m)** (when there are no hash collisions, and the hash function works efficiently).

**Worst Case Time Complexity**: **O(n \* m)** (if there are many hash collisions and we perform a character-by-character check for every window).

**CODE:**

#include <stdio.h>

#include <string.h>

#define d 256 // Number of characters in the input alphabet (ASCII)

#define q 101 // A prime number for hashing to avoid collisions

// Function to calculate the hash value of a string (pattern or text window)

int calculateHash(char \*str, int length) {

    int hash = 0;

    for (int i = 0; i < length; i++) {

        hash = (d \* hash + str[i]) % q;

    }

    return hash;

}

// Rabin-Karp algorithm to search for the pattern in the text

void rabinKarpSearch(char \*text, char \*pattern) {

    int n = strlen(text); // Length of the text

    int m = strlen(pattern); // Length of the pattern

    int patternHash = calculateHash(pattern, m); // Hash value of the pattern

    int textHash = calculateHash(text, m); // Hash value of the first window of the text

    int h = 1;

    // Calculate d^(m-1) for use in rolling hash

    for (int i = 0; i < m - 1; i++) {

        h = (h \* d) % q;

    }

    // Sliding window to search for pattern

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

        // If hash values match, check the characters one by one

        if (patternHash == textHash) {

            int j;

            for (j = 0; j < m; j++) {

                if (text[i + j] != pattern[j]) {

                    break;

                }

            }

            // If all characters match, print the match

            if (j == m) {

                printf("Pattern found at index %d\n", i);

            }

        }

        // Calculate hash for the next window of the text (rolling hash)

        if (i < n - m) {

            textHash = (d \* (textHash - text[i] \* h) + text[i + m]) % q;

            if (textHash < 0) {

                textHash = (textHash + q);

            }

        }

    }

}

int main() {

    char text[] = "ababcababcabc";

    char pattern[] = "abc";

    printf("Text: %s\n",text);

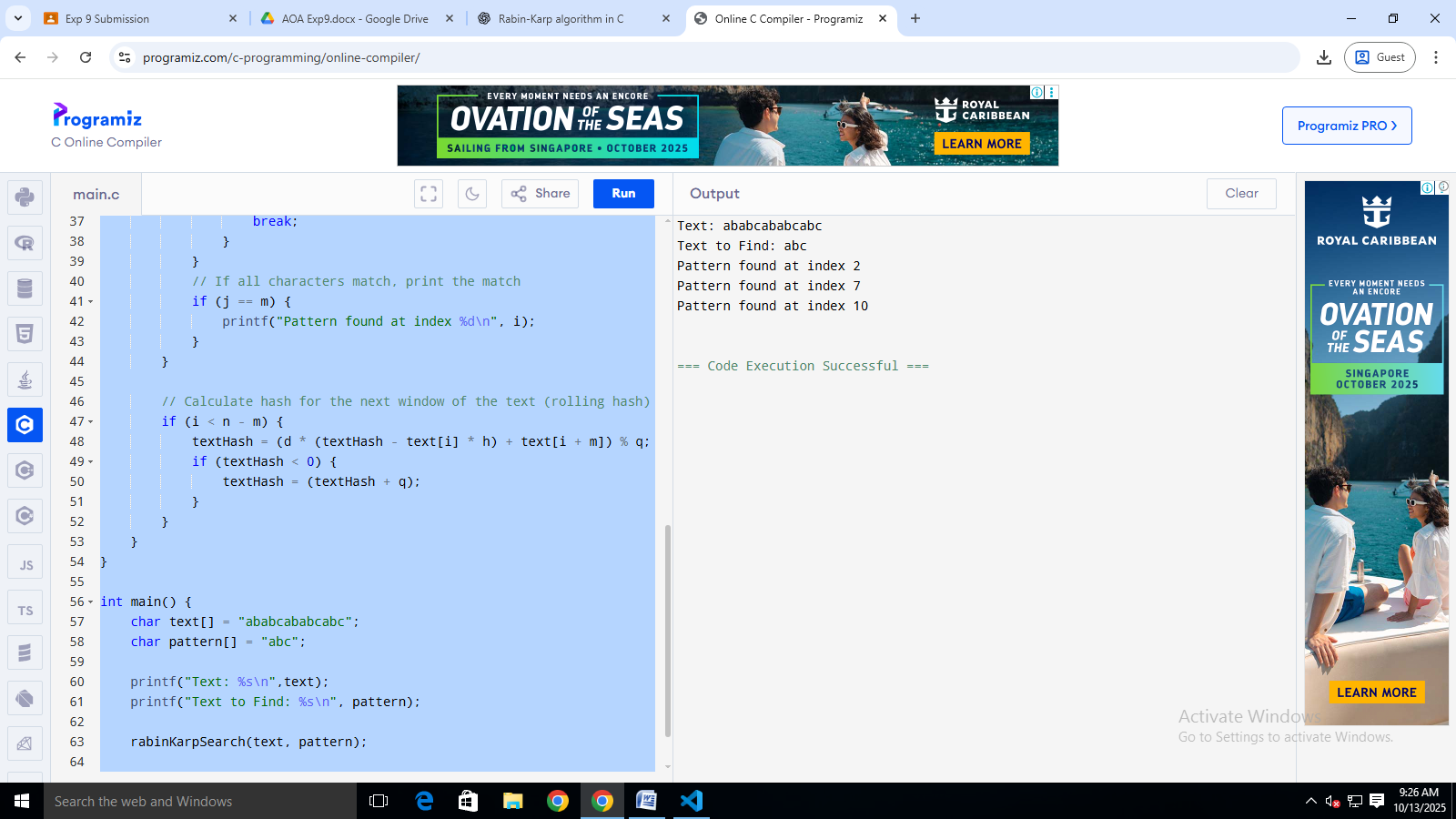
    printf("Text to Find: %s\n", pattern);

    rabinKarpSearch(text, pattern);

    return 0;

}

**OUTPUT:**

****

**POST LAB QUESTIONS**

1. What is the core idea behind the Naive pattern matching algorithm? How does it compare to Rabin-Karp in terms of approach?

**Ans.** Below is the Comparison Table

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Naive Pattern Matching** | **Rabin-Karp Algorithm** |
| **Approach** | Character-by-character comparison at each position. | Uses hash values for initial comparison, checks characters only if hashes match. |
| **Time Complexity (Best Case)** | O(n) — when pattern matches at the first position. | O(n + m) — when no hash collisions occur. |
| **Time Complexity (Worst Case)** | O(n \* m) — for each position, compare m characters. | O(n \* m) — if many hash collisions occur. |
| **Space Complexity** | O(1) — only the pattern and text are stored. | O(1) — uses extra space for hashing and window calculation. |
| **Efficiency** | Less efficient for large texts, as it performs character comparison for every position. | More efficient for large texts and multiple patterns, as hashing reduces unnecessary comparisons. |
| **Best Use Case** | Small text/pattern or few matches. | Large text or multiple pattern searches. |
| **Handling Hash Collisions** | No need to handle collisions. | Needs efficient hash function to minimize collisions. |
| **Overhead** | Simple to implement, minimal overhead. | Extra overhead due to hash calculations and window sliding. |
| **Scenarios for Superiority** | Small datasets, few pattern matches, or few hash collisions. | Large datasets, multiple pattern searches, and minimal collisions. |
| **Practical Application** | Simple string matching tasks. | Searching for multiple patterns in large documents. |

1. Perform a dry run of the Rabin-Karp algorithm on the text “ABABDABACDABABCABAB” with the pattern “ABABCABAB”. Show all comparisons.

### Ans. **Dry Run of Rabin-Karp on "ABABDABACDABABCABAB" with Pattern "ABABCABAB":**

* **Initial Hash Calculation**:
  + Pattern and text window "ABABDABAC" both have hash = 61.
* **First Match**:
  + Window "ABABDABAC" matches the pattern at index 0.
* **Sliding the Window**:
  + Subsequent windows do not match the pattern (after hash comparison and character-by-character check).
* **Final Output**:
  + Match found at index 0.

1. In what scenarios would the Naive algorithm outperform Rabin-Karp?

### Ans. **Scenarios Where Naive Algorithm Outperforms Rabin-Karp:**

* **Small Text or Pattern**: Naive is simple and fast when the text is short.
* **Few Matches**: If few matches are found, Rabin-Karp's hashing overhead can make Naive faster.
* **High Hash Collisions**: If hash collisions are frequent, Rabin-Karp degrades to O(n \* m), making it slower than Naive.
* **Small Dataset**: Naive can be more efficient for small inputs due to its simplicity.

**CONCLUSION:**

Both the **Naive Pattern Matching** and **Rabin-Karp** algorithms are useful for string matching tasks, but their performance varies depending on the context:

* **Naive Algorithm** is simple to implement and works well for small texts or when there are few matches. However, it can become inefficient for large texts or when hash collisions are frequent, as it requires repeated character-by-character comparisons.
* **Rabin-Karp Algorithm** is more efficient for larger datasets or when multiple patterns need to be searched simultaneously, thanks to its use of hashing. It reduces unnecessary comparisons by quickly discarding non-matching substrings based on hash values. However, its performance can degrade if many hash collisions occur, potentially making it as slow as the Naive algorithm in the worst case.