A MINI-PROJECT REPORT ON

“String Matching Algorithms”

SUBMITTED TO THE SAVITRIBAI PHULE PUNE UNIVERSITY, PUNE,IN THE PARIAL FULFILLMENT OF THE REQUIREMENTS FOR SUBJECT

OF

LEARNING PRACTICAL – III

BY

Name: ABHAY CHABUK **Roll No.: BECB106**

Name: DNYANESHWARI JANGALE **Roll No.: BECB123**

UNDER THE GUIDANCE

OF

PROF. NILESH KAMBLE

Nutan Maharashtra Vidya Prasarak Mandal's

NUTAN MAHARASHTRA INSTITUTE OF ENGINEERING & TECHNOLOGY, PUNE

Samarth Vidya Sankul, Vishnupuri,

Talegaon Dabhade, Maharashtra 410507

CERTIFICATE

This is to certify that the Mini-Project entitled

"String Matching Algorithms"

Submitted By

Name: Abhay Chabuk

Roll No: BECB106

is a bonafide work carried out by him under the supervision of Prof. Nilesh Kamble and it is approved for the partial fulfillment of the requirement of subject Learning Practical- III

The Mini-Project work has not been earlier submitted to any other institute or university for the award of degree or diploma.

Prof. Nilesh Kamble	**Prof. Rohini Hanchate**	**Dr. Pramod Patil**
(Subject Co-ordinator)	(Head of Department)	(Principal)

Place : Pune

Date :

INDEX

Table of Contents

1. Problem Statement:

This project addresses the **String Matching Problem**: finding a pattern () within a larger text (). Efficient solutions are critical for widespread applications like search, text editing, and bioinformatics. The project compares two solutions: the basic, direct **Naive Algorithm** and the optimized, hash-based **Rabin-Karp Algorithm**.

2. Objectives:

The primary goals were to:

1. **Implement** both the Naive and Rabin-Karp algorithms in Java.
2. **Compare** the algorithms by observing the total number of **character comparisons** required for the same input, thereby quantifying the difference in efficiency.

3. Methedology:

3.1 The Naive String Matching Algorithm

The Naive algorithm is the simplest approach, operating on a brute-force mechanism.

- **Mechanism:** It slides the pattern one position at a time across the text . At each position (or "shift"), it compares every character of with the corresponding character of the window.
- **Drawback:** In cases where the pattern and the text window have long prefixes that match (e.g., searching for AAAAB in AAAAAAA), the algorithm repeatedly performs costly partial comparisons before shifting just one position, leading to a worst-case time complexity of.

3.2 The Rabin-Karp Algorithm

The Rabin-Karp algorithm introduces a significant optimization by using a hashing technique to pre-filter non-matching windows.

- **Mechanism:** It computes a unique numerical hash value for the pattern and for the first -length window of the text . As the window slides, the algorithm only compares the two hash values.

- **Verification:** A full character-by-character comparison (time) is performed **only** when the pattern hash matches the window hash. This avoids the time spent checking non-matching characters in the Naive algorithm.
- **Complexity:** On average, the algorithm achieves a linear time complexity of , making it far more efficient for typical inputs.

3.3 The Rolling Hash Principle

The core innovation of Rabin-Karp is the **rolling hash**. Instead of re-calculating the hash value for a new -length window from scratch (which would take time), the rolling hash updates the hash value in **time** by:

1. Subtracting the numerical value of the character leaving the window.
2. Multiplying the result by the base and taking the modulus.
3. Adding the numerical value of the new character entering the window.

This constant-time update allows the algorithm to slide the search window almost instantly, only pausing for a full character verification when a hash match occurs.

Feature	Naive Algorithm	Rabin-Karp Algorithm
Principle	Brute-force: Compares characters in *every* possible window sequentially.	Hashing: Compares **hash values** first, using a rolling hash function.
Verification	Always checks characters unless a mismatch is found early.	Checks characters *only* if the hash values match (to prevent spurious hits/collisions).
Worst-Case Complexity	**Error! Filename not specified.**	(only if many hash collisions occur)
Average-Case Complexity	**Error! Filename not specified.**	(Highly efficient)

4. Implementation:

The implementation was completed in a single Java class, StringMatchingAlgorithms.java. The code includes functions for both algorithms and a robust main method to handle user input, execute both searches, and print the step-by-step working process, ensuring transparent observation of the difference

CODE:

```
import java.util.Scanner;

public class StringMatchingAlgorithms {

    private static final int ALPHABET_SIZE = 256; // Base for hashing (ASCII characters)

    private static final int PRIME = 101; // A prime modulus for the hash function

    // Helper class to store and return results for comparison

    private static class SearchResult {

        int index;

        int comparisons;

        public SearchResult(int index, int comparisons) {

            this.index = index;

            this.comparisons = comparisons;

        }

    }

    // ----------------------------------------------------------------------

    // 1. NAIVE STRING MATCHING ALGORITHM

    // ----------------------------------------------------------------------

    public static SearchResult naiveSearch(String text, String pattern) {

        int N = text.length();

        int M = pattern.length();

        int comparisons = 0;
```

```
    System.out.println("\n--- Naive Search Execution ---\n");
    for (int i = 0; i <= N - M; i++) {
        System.out.println("Shift " + i + ": Checking window \"" + text.substring(i, i + M)
                + "\" against pattern \"" + pattern + "\"");
        int j;
        // Character-by-character comparison
        for (j = 0; j < M; j++) {
            comparisons++;
            if (text.charAt(i + j) != pattern.charAt(j)) {
                System.out.println(
                        "  Mismatch found at Pattern index " + j + " (Text index " + (i + j) + "). Shifting...");
                break;
            }
        }
        if (j == M) {
            System.out.println("  -> COMPLETE MATCH FOUND at index " + i + "!");
            return new SearchResult(i, comparisons);
        }
    }
    return new SearchResult(-1, comparisons);
}
// ---------------------------------------------------------------------
// 2. RABIN-KARP STRING MATCHING ALGORITHM
// ---------------------------------------------------------------------
public static SearchResult rabinKarpSearch(String text, String pattern) {
    int N = text.length();
    int M = pattern.length();
```

```
        int pHash = 0; // Pattern hash
        int tHash = 0; // Current window hash
        int h = 1; // d^(M-1) % q
        int comparisons = 0;
        System.out.println("\n--- Rabin-Karp Search Execution ---\n");
         System.out.println("Parameters: Base (d) = " + ALPHABET_SIZE + ", Modulo Prime (q) = " +
PRIME);
        // 1. Precompute h and initial hashes
        for (int i = 0; i < M - 1; i++) {
            h = (h * ALPHABET_SIZE) % PRIME;
        }
        for (int i = 0; i < M; i++) {
            pHash = (ALPHABET_SIZE * pHash + pattern.charAt(i)) % PRIME;
            tHash = (ALPHABET_SIZE * tHash + text.charAt(i)) % PRIME;
        }
        System.out.println("Pattern Hash (pHash) is calculated as: " + pHash);
        // 2. Slide the pattern
        for (int i = 0; i <= N - M; i++) {
            System.out.print("Shift " + i + ": Window \"" + text.substring(i, i + M) + "\" | tHash: " + tHash);
            // Compare hash values
            if (pHash == tHash) {
                // HASH MATCH: Perform the character-by-character check (avoids spurious hits)
                System.out.print(" | *** HASH MATCH! Starting verification... ");
                int j;
                for (j = 0; j < M; j++) {
                    comparisons++; // Character comparison count
                    if (text.charAt(i + j) != pattern.charAt(j)) {
                        System.out.println("Spurious Hit (Mismatch)");
```

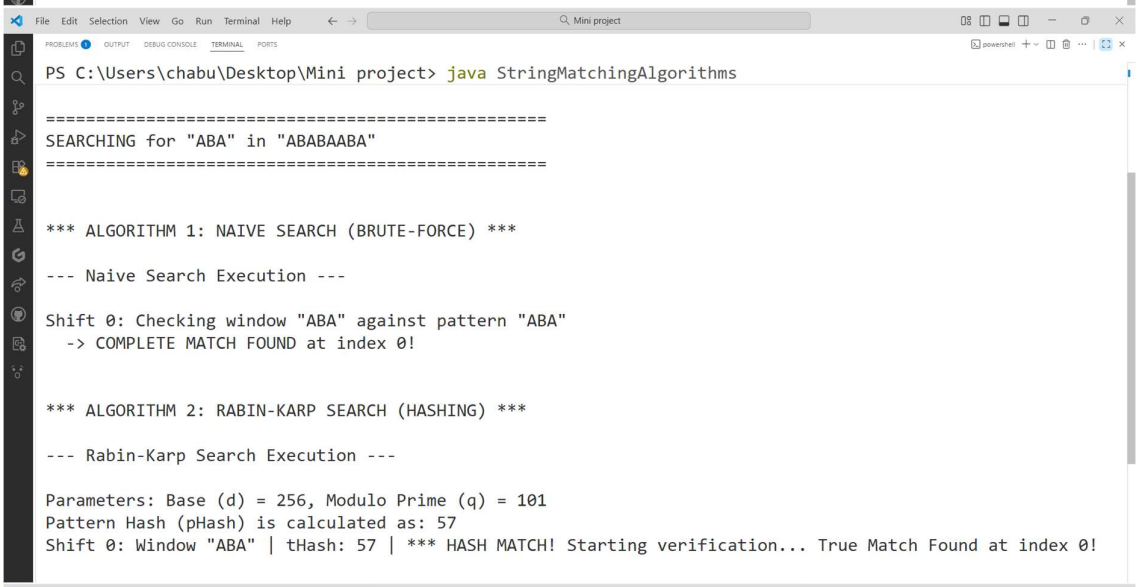
```
                break;
            }
        }
        if (j == M) {
            System.out.println("True Match Found at index " + i + "!");
            return new SearchResult(i, comparisons);
        }
    } else {
        System.out.println(" | Hash Mismatch. Skipping character check.");
    }
    // Calculate hash for the next window (rolling hash)
    if (i < N - M) {
        // t(i+1) = [ d(t(i) - T[i]*h) + T[i+M] ] mod q
        tHash = (ALPHABET_SIZE * (tHash - text.charAt(i) * h) + text.charAt(i + M)) % PRIME;
        // Ensure tHash is non-negative
        if (tHash < 0) {
            tHash = (tHash + PRIME);
        }
    }
  }
  return new SearchResult(-1, comparisons);
}
// ---------------------------------------------------------------------
// MAIN METHOD & COMPARISON LOGIC
// ---------------------------------------------------------------------
public static void main(String[] args) {
  Scanner scanner = new Scanner(System.in);
```

```
String text, pattern;
System.out.println("=======================================================");
System.out.println("STRING MATCHING ALGORITHMS COMPARISON");
System.out.println("=======================================================");
// 1. Get Text Input
System.out.println("\n--- Step 1: Input the MAIN TEXT (Haystack) ---");
System.out.print("Example: 'ABABAABABA' | Enter your Text: ");
text = scanner.nextLine().toUpperCase();
// 2. Get Pattern Input
System.out.println("\n--- Step 2: Input the PATTERN (Needle) ---");
System.out.print("Example: 'ABA' | Enter your Pattern: ");
pattern = scanner.nextLine().toUpperCase();
if (pattern.length() > text.length()) {
    System.out.println("\nError: Pattern length cannot exceed Text length.");
    return;
}
System.out.println("\n=================================================");
System.out.println("SEARCHING for \"" + pattern + "\" in \"" + text + "\"");
System.out.println("=================================================");
// Run Naive Search
System.out.println("\n\n*** ALGORITHM 1: NAIVE SEARCH (BRUTE-FORCE) ***");
SearchResult naiveResult = naiveSearch(text, pattern);
// Run Rabin-Karp Search
System.out.println("\n\n*** ALGORITHM 2: RABIN-KARP SEARCH (HASHING) ***");
SearchResult rkResult = rabinKarpSearch(text, pattern);
// Final Comparison
```

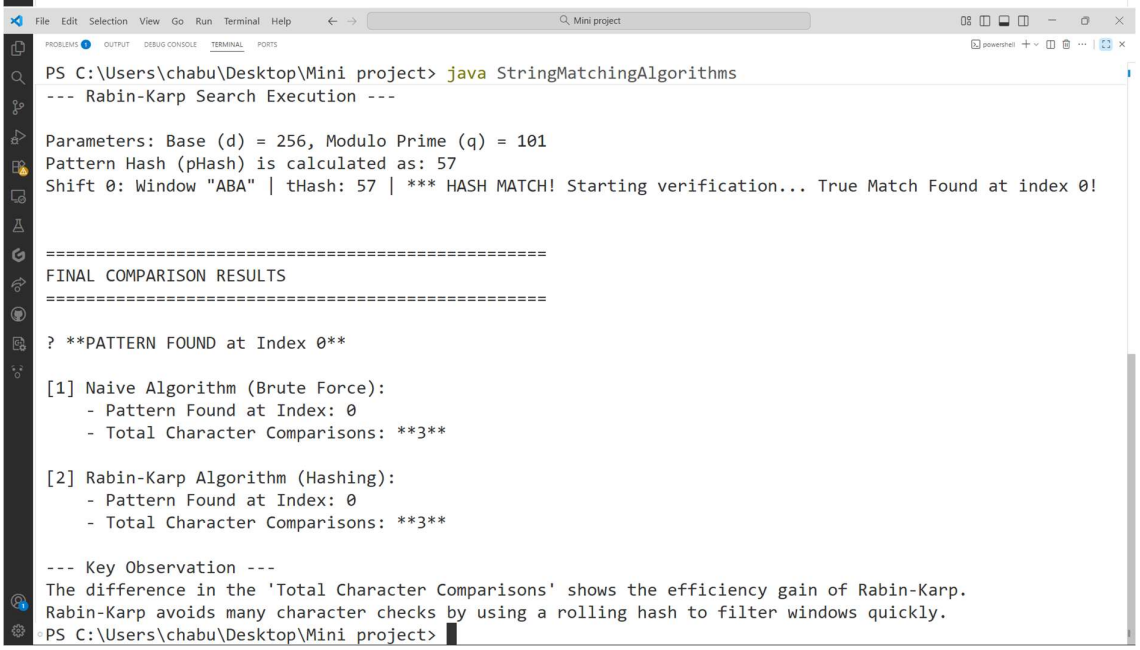
```
        System.out.println("\n\n==========================================================");

        System.out.println("FINAL COMPARISON RESULTS");

        System.out.println("==========================================================");

        if (naiveResult.index != -1) {

            System.out.println("\n✅ **PATTERN FOUND at Index " + naiveResult.index + "**");

        } else {

            System.out.println("\n❌ **PATTERN NOT FOUND**");

        }

        System.out.println("\n[1] Naive Algorithm (Brute Force):");

            System.out.println("    - Pattern Found at Index: " + (naiveResult.index == -1 ? "N/A" :
naiveResult.index));

        System.out.println("    - Total Character Comparisons: **" + naiveResult.comparisons + "**");

        System.out.println("\n[2] Rabin-Karp Algorithm (Hashing):");

       System.out.println("    - Pattern Found at Index: " + (rkResult.index == -1 ? "N/A" : rkResult.index));

        System.out.println("    - Total Character Comparisons: **" + rkResult.comparisons + "**");

        System.out.println("\n--- Key Observation ---");



        System.out.println(

                "The difference in the 'Total Character Comparisons' shows the efficiency gain of Rabin-
Karp.");

        System.out

                .println("Rabin-Karp avoids many character checks by using a rolling hash to filter windows
quickly.");

        scanner.close();

    }

}
```

5. Output:

```
PS C:\Users\chabu\Desktop\Mini project> javac StringMatchingAlgorithms.java
PS C:\Users\chabu\Desktop\Mini project> java StringMatchingAlgorithms
==================================================
STRING MATCHING ALGORITHMS COMPARISON
==================================================

--- Step 1: Input the MAIN TEXT (Haystack) ---
Example: 'ABABAABABA' | Enter your Text: ABABAABA

--- Step 2: Input the PATTERN (Needle) ---
Example: 'ABA' | Enter your Pattern: ABA
```



```
PS C:\Users\chabu\Desktop\Mini project> java StringMatchingAlgorithms

==================================================
SEARCHING for "ABA" in "ABABAABA"
==================================================


*** ALGORITHM 1: NAIVE SEARCH (BRUTE-FORCE) ***

--- Naive Search Execution ---

Shift 0: Checking window "ABA" against pattern "ABA"
  -> COMPLETE MATCH FOUND at index 0!


*** ALGORITHM 2: RABIN-KARP SEARCH (HASHING) ***

--- Rabin-Karp Search Execution ---

Parameters: Base (d) = 256, Modulo Prime (q) = 101
Pattern Hash (pHash) is calculated as: 57
Shift 0: Window "ABA" | tHash: 57 | *** HASH MATCH! Starting verification... True Match Found at index 0!
```



```
PS C:\Users\chabu\Desktop\Mini project> java StringMatchingAlgorithms
--- Rabin-Karp Search Execution ---

Parameters: Base (d) = 256, Modulo Prime (q) = 101
Pattern Hash (pHash) is calculated as: 57
Shift 0: Window "ABA" | tHash: 57 | *** HASH MATCH! Starting verification... True Match Found at index 0!


==================================================
FINAL COMPARISON RESULTS
==================================================

? **PATTERN FOUND at Index 0**

[1] Naive Algorithm (Brute Force):
    - Pattern Found at Index: 0
    - Total Character Comparisons: **3**

[2] Rabin-Karp Algorithm (Hashing):
    - Pattern Found at Index: 0
    - Total Character Comparisons: **3**

--- Key Observation ---
The difference in the 'Total Character Comparisons' shows the efficiency gain of Rabin-Karp.
Rabin-Karp avoids many character checks by using a rolling hash to filter windows quickly.
PS C:\Users\chabu\Desktop\Mini project>
```