SAVITRIBAI PHULE PUNE UNIVERSITY, PUNE

A

MINI PROJECT REPORT ON "String Matching Visualizer"

SUBMITTED TO THE SAVITRIBAI PHULE PUNE UNIVERSITY, PUNE

IN THE FULFILLMENT OF THE REQUIREMENT

OF

Laboratory Practice IV

Fourth Year Computer Engineering

Academic Year 2024-25

BY

Name of Students:	**Roll No.:**
Pratik Abhang	4101001
Darshana Borase	4101030
Omkar Darekar	4101032
Chaitanya Gunde	4101056

Under the Guidance of

Mr. S. A. Shinde

DEPARTMENT OF COMPUTER ENGINEERING
STES'S SINHGAD INSTITUTE OF TECHNOLOGY AND SCIENCE
NARHE, PUNE – 411041

CERTIFICATE

This is to certify that,

Name of Students:	**Roll No.:**
Pratik Abhang	4101001
Darshana Borase	4101030
Omkar Darekar	4101032
Chaitanya Gunde	4101056

studying in SEM-VII, B.E. Computer Engineering, they have successfully completed their LP-III Lab Mini-Project report titled **"String Matching Visualizer"** under the supervision of **Mr. S. A. Shinde**, in fulfillment of the requirements for Laboratory Practice-III for the academic year 2024-2025, as prescribed by Savitribai Phule Pune University, Pune.

Mr. S. A. Shinde	Dr. G. S. Navale
Guide	**HOD**
Department of Computer Engineering	**Department of Computer Engineering**

Dr. S. D. Markande

Principal

SINHGAD INSTITUTE OF TECHNOLOGY AND SCIENCE, NARHE, PUNE– 411041

Place:

Date:

ACKNOWLEDGEMENT

We take this opportunity to acknowledge each and every one who contributed towards our work. We express our sincere gratitude towards guide **Mr. S. A. Shinde**, Assistant Professor at Sinhgad Institute of Technology and Science, Narhe, Pune for her valuable inputs, guidance and support throughout the course.

We wish to express our thanks to **Dr. G. S. Navale,** Head of Computer Engineering Department, Sinhgad Institute of Technology and Science, Narhe for giving us all the help and important suggestions all over the Work.

We thank all the teaching staff members, for their indispensable support and priceless suggestions. We also thank our friends and family for their help in collecting data, without their help **LP-III** report have not been completed. At the end our special thanks to **Dr. S. D. Markande,** Principal Sinhgad Institute of Technology and Science, Narhe for providing ambience in the college, which motivate us to work.

Name of Students:	**Sign:**
Pratik Abhang	
Darshana Borase	
Omkar Darekar	
Chaitanya Gunde	

CONTENTS

1. INTRODUCTION

String matching is a fundamental problem in computer science with applications in text search, DNA sequencing, plagiarism detection, and data mining. Efficiently finding a pattern within a larger text is crucial for optimizing these tasks. Two common algorithms used for string matching are the Naive String Matching Algorithm and the Rabin-Karp Algorithm.

Naive String Matching performs a brute-force comparison of the pattern with every substring of the text.

Rabin-Karp improves efficiency by using a rolling hash function to compare hash values instead of direct string comparison.

This project focuses on implementing and comparing both algorithms to evaluate their performance, particularly in large-scale data. By doing so, we aim to observe their differences in terms of time complexity, handling of collisions, and performance on various input sizes. Understanding these differences is critical to selecting the right algorithm for real-world string matching problems where efficiency is paramount.

Problem Statement:

Implement the Naive string matching algorithm and Rabin-Karp algorithm for string matching. Observe difference in working of both the algorithms for the same input..

2. OBJECTIVE

The primary objective of this project is to implement two string matching algorithms the Naive String Matching Algorithm and the Rabin-Karp Algorithm and compare their performance on the same input data. The goals include:

- Understand Algorithmic Differences: Analyze how the Naive algorithm performs exhaustive comparisons, while Rabin-Karp uses hashing for efficiency.
- Evaluate Time Complexity: Measure and compare their time complexity in practical cases.
- Observe Hash Collisions: Examine Rabin-Karp's behavior in cases of hash collisions.
- Demonstrate Real-World Applications: Identify scenarios where each algorithm is most effective (e.g., small vs. large datasets).
- Assess Scalability: Test the algorithms with increasingly large text sizes to see how they scale.
- Optimization Opportunities: Investigate if optimizations such as better hash functions can reduce the chance of collisions in Rabin-Karp.
- Benchmark Performance: Compare the computational performance and memory usage of both algorithms, providing insights into the conditions under which each algorithm is ideal.
- Enhance Problem-Solving Skills: Improve understanding of practical algorithm design and analysis, aiding in better problem-solving strategies for string matching tasks.

3. HARDWARE AND SOFTWARE REQUIREMENT

Hardware: Laptop

Software: vs code

4. ARCHITECTURE & COMPONENTS

Naive String Matching Algorithm:

This algorithm works by comparing the given pattern with every possible substring of the text. The algorithm moves one character at a time and performs direct character-by-character comparisons for each possible position of the pattern in the text.

Component Breakdown:

Input: A pattern P and text T.

Logic: For each position i in the text T, check if the substring T[i...i+m-1] matches the pattern P.

Output: Positions where the pattern occurs in the text.

Rabin-Karp Algorithm:

This algorithm improves efficiency by using a rolling hash function. Instead of directly comparing strings, it first computes the hash of the pattern and each possible substring of the text. If the hash values match, it performs a direct comparison to verify the match.

Component Breakdown:

Input: A pattern P, text T, and a prime number q for the hash function.

Logic: Compute the hash of the pattern and the first window of text, then slide the window one character at a time, updating the hash.

Collision Handling: In case of a hash match, verify by comparing the actual characters.

Output: Positions where the pattern occurs in the text.

5. WORKING

Naive String Matching Algorithm:

The Naive algorithm works by comparing the pattern with every possible substring of the text, starting from the first character and shifting by one until the end.

Steps:

Begin with the first position of the text.

Compare each character of the pattern with the corresponding character in the text.

If a mismatch occurs, move the pattern one position to the right and repeat.

If all characters match, store the current position as a match.

Complexity: Time complexity is O((n-m+1) * m), where n is the length of the text and m is the length of the pattern. This leads to performance degradation for large texts or patterns due to redundant comparisons at every step.

Rabin-Karp Algorithm:

This algorithm improves the Naive approach by using hashing. Instead of directly comparing characters, it compares the hash values of the pattern and the current window of text.

Steps:

Precompute the hash of the pattern.

Compute the hash of the first window (substring of length equal to the pattern) of the text.

Compare the hash values:

If the hash values match, check the actual characters to confirm a match.

If they don't match, slide the window by one character and update the hash using a rolling hash function (which reuses the previous hash value to compute the next).

Continue this process until the entire text has been checked.

Complexity: The Rabin-Karp algorithm runs in O(n) in the best case, but hash collisions can lead to a worst-case time complexity of O(n * m). However, in practice, it is much faster than the Naive approach when hash collisions are infrequent.

Comparative Working:

Naive Method: Performs a direct comparison for each possible substring, making it straightforward but computationally expensive.

Rabin-Karp: Reduces unnecessary comparisons by first comparing hashes and only comparing the actual characters when the hashes match, making it more efficient for large inputs. However, its performance degrades if hash collisions are frequent, as direct comparisons are then required.

Example for Better Understanding:

Let’s say you have a text: "abcdabcabcd" and a pattern: "abc".

Naive Approach:

Start comparing "abc" with "abc" at position 1, then compare with "bcd" at position 2, and continue until the end of the text.

The algorithm checks every position, which involves repeated comparisons even when the previous comparisons already ruled out a match.

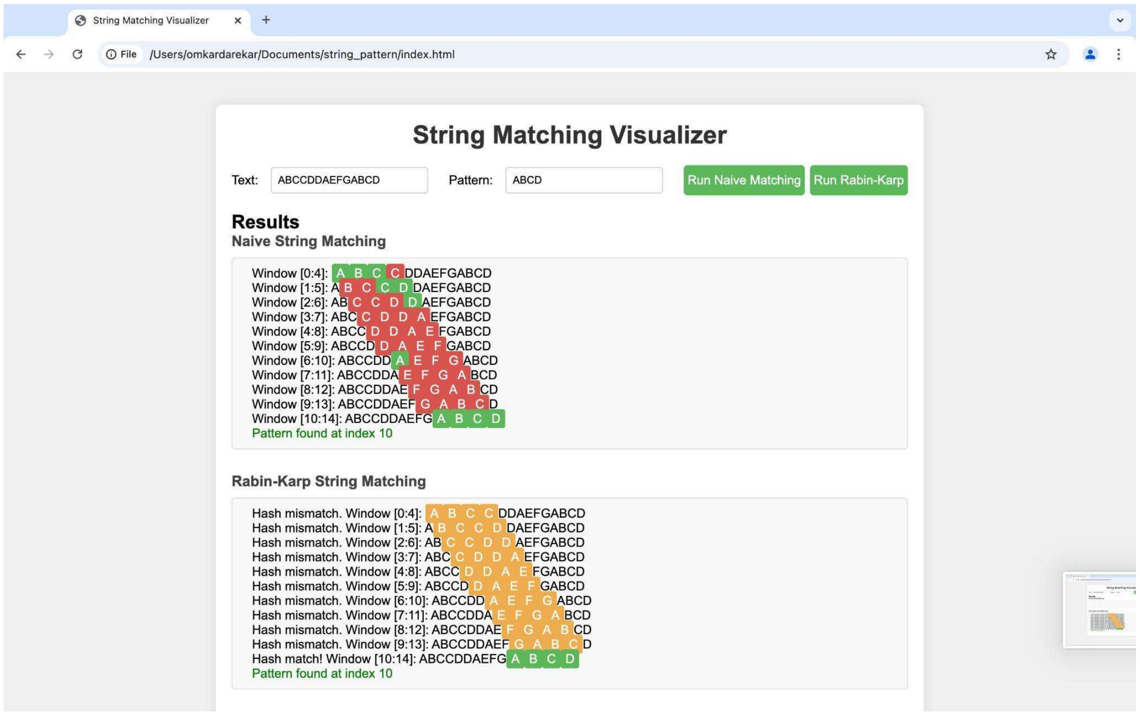
Rabin-Karp Approach:

Hash "abc" and compare the hash value with the hash of the first substring of the text.

If the hashes match, compare the characters. If not, shift the window and update the hash in constant time.

The algorithm avoids redundant character-by-character comparison for every position.

6. RESULTS


String Matching Visualizer
File /Users/omkardarekar/Documents/string_pattern/index.html

String Matching Visualizer

Text: ABCCDDAEFGABCD Pattern: ABCD

Run Naive Matching
Run Rabin-Karp

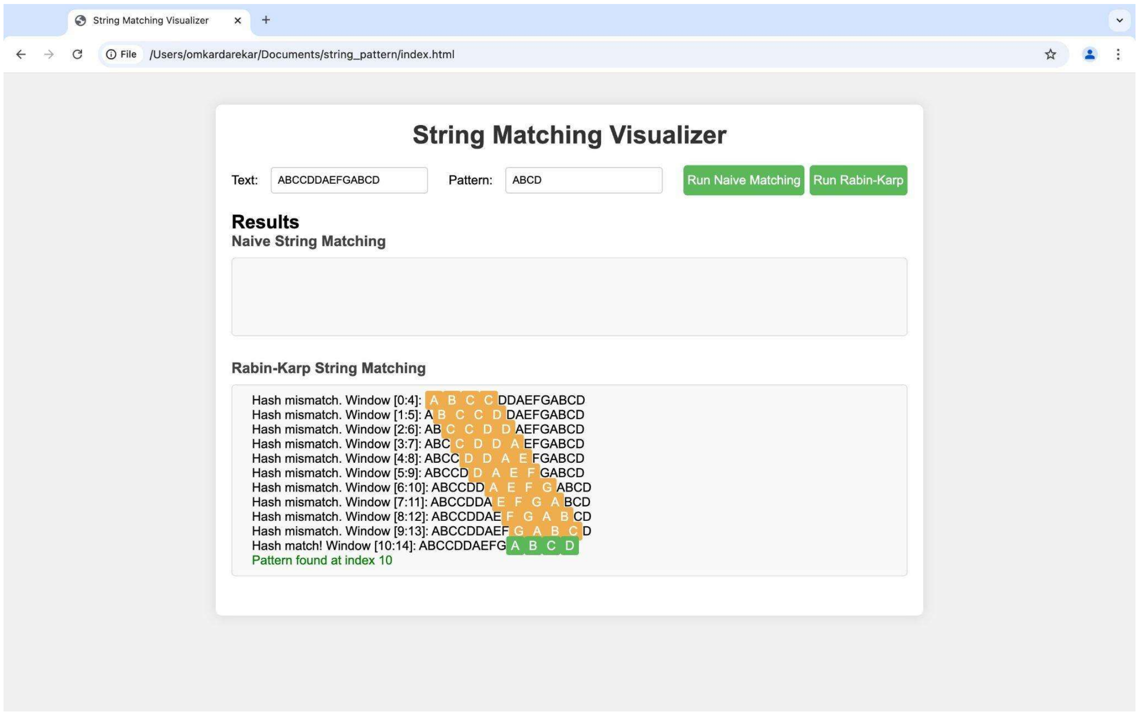
Results

Naive String Matching

```
Window [0:4]: A B C C DDAEFGABCD
Window [1:5]: A B C C D DAEFGABCD
Window [2:6]: AB C C D D AEFGABCD
Window [3:7]: ABC C D D A EFGABCD
Window [4:8]: ABCC D D A E FGABCD
Window [5:9]: ABCCD D A E F GABCD
Window [6:10]: ABCCDD A E F G ABCD
Window [7:11]: ABCCDDA E F G A BCD
Window [8:12]: ABCCDDAE F G A B CD
Window [9:13]: ABCCDDAEF G A B C D
Window [10:14]: ABCCDDAEFG A B C D
Pattern found at index 10
```

Rabin-Karp String Matching

```
Hash mismatch. Window [0:4]: A B C C DDAEFGABCD
Hash mismatch. Window [1:5]: A B C C D DAEFGABCD
Hash mismatch. Window [2:6]: AB C C D D AEFGABCD
Hash mismatch. Window [3:7]: ABC C D D A EFGABCD
Hash mismatch. Window [4:8]: ABCC D D A E FGABCD
Hash mismatch. Window [5:9]: ABCCD D A E F GABCD
Hash mismatch. Window [6:10]: ABCCDD A E F G ABCD
Hash mismatch. Window [7:11]: ABCCDDA E F G A BCD
Hash mismatch. Window [8:12]: ABCCDDAE F G A B CD
Hash mismatch. Window [9:13]: ABCCDDAEF G A B C D
Hash match! Window [10:14]: ABCCDDAEFG A B C D
Pattern found at index 10
```


String Matching Visualizer
File /Users/omkardarekar/Documents/string_pattern/index.html

String Matching Visualizer

Text: ABCCDDAEFGABCD Pattern: ABCD

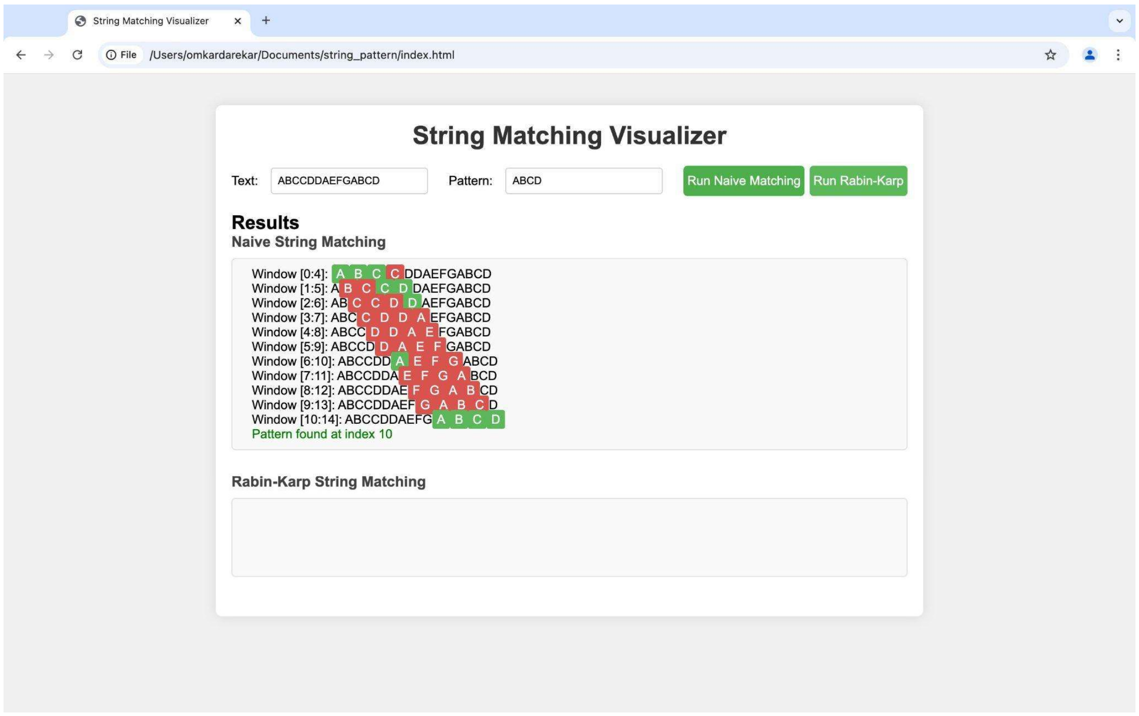
Run Naive Matching
Run Rabin-Karp

Results

Naive String Matching

Rabin-Karp String Matching

```
Hash mismatch. Window [0:4]: A B C C DDAEFGABCD
Hash mismatch. Window [1:5]: A B C C D DAEFGABCD
Hash mismatch. Window [2:6]: AB C C D D AEFGABCD
Hash mismatch. Window [3:7]: ABC C D D A EFGABCD
Hash mismatch. Window [4:8]: ABCC D D A E FGABCD
Hash mismatch. Window [5:9]: ABCCD D A E F GABCD
Hash mismatch. Window [6:10]: ABCCDD A E F G ABCD
Hash mismatch. Window [7:11]: ABCCDDA E F G A BCD
Hash mismatch. Window [8:12]: ABCCDDAE F G A B CD
Hash mismatch. Window [9:13]: ABCCDDAEF G A B C D
Hash match! Window [10:14]: ABCCDDAEFG A B C D
Pattern found at index 10
```


String Matching Visualizer

/Users/omkardarekar/Documents/string_pattern/index.html

String Matching Visualizer

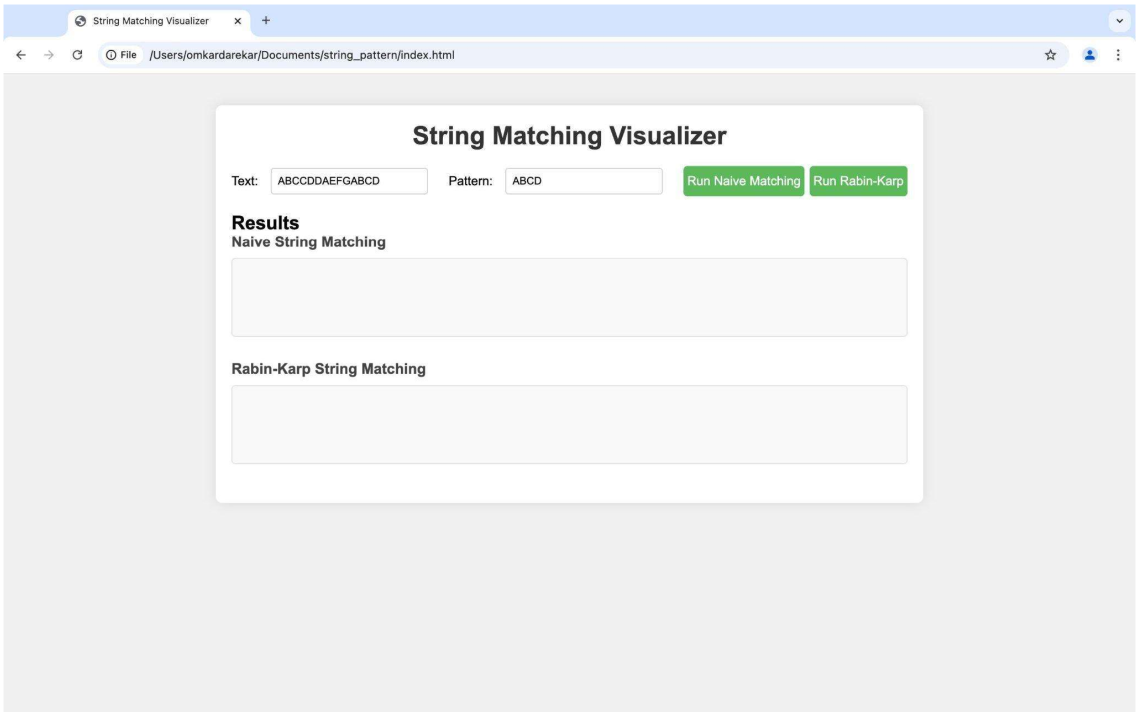
Text: ABCCDDAEFGABCD   Pattern: ABCD   Run Naive Matching   Run Rabin-Karp

Results

Naive String Matching

```
Window [0:4]: A B C C DDAEFGABCD
Window [1:5]: A B C C D DAEFGABCD
Window [2:6]: AB C C D D AEFGABCD
Window [3:7]: ABC C D D A EFGABCD
Window [4:8]: ABCC D D A E FGABCD
Window [5:9]: ABCCD D A E F GABCD
Window [6:10]: ABCCDD A E F G ABCD
Window [7:11]: ABCCDDA E F G A BCD
Window [8:12]: ABCCDDAE F G A B CD
Window [9:13]: ABCCDDAEF G A B C D
Window [10:14]: ABCCDDAEFG A B C D
Pattern found at index 10
```

Rabin-Karp String Matching


String Matching Visualizer

/Users/omkardarekar/Documents/string_pattern/index.html

String Matching Visualizer

Text: ABCCDDAEFGABCD   Pattern: ABCD   Run Naive Matching   Run Rabin-Karp

Results

Naive String Matching

Rabin-Karp String Matching

7. CONCLUSION

In this project, both the Naive String Matching Algorithm and the Rabin-Karp Algorithm were implemented and evaluated for performance on the same input.

Naive Algorithm: While straightforward and easy to implement, it performs poorly for larger datasets due to its O(n*m) time complexity, making it inefficient for larger-scale string matching tasks.
Rabin-Karp Algorithm: The use of a rolling hash function significantly improves performance, especially for large texts. It efficiently reduces the number of direct comparisons. However, it can suffer from hash collisions, which degrade its performance in some cases.
Overall, Rabin-Karp is more efficient for larger inputs and can scale well, especially with optimized hash functions. The choice between these algorithms should depend on the dataset size and the importance of reducing collisions for Rabin-Karp. For small-scale problems or when simplicity is prioritized, the Naive algorithm may still be an acceptable choice. However, for larger and more complex applications, Rabin-Karp provides a clear advantage.

Future improvements could involve experimenting with more advanced hash functions to further minimize the impact of collisions, making Rabin-Karp even more robust in real-world applications.

8. REFERENCES

1) Cormen, Thomas H., et al. "Introduction to Algorithms." MIT Press, 2009.
2) Rabin, Michael O., and Richard Karp. "Efficient Randomized Pattern-Matching Algorithms." IBM Journal of Research and Development, 1987.
3) Online resources like GeeksforGeeks, TutorialsPoint, and Python documentation for practical implementations.
4) Research papers and articles on string matching algorithms and their practical applications in areas like DNA sequence analysis, data mining, and information retrieval.
5) orithm
6) "How to Build Ethereum Dapp with React.js · Complete Step-By-Step Guide" https://www.dappuniversity.com/articles/ethereum-dapp-react-tutorial