

## SAVITRIBAI PHULE PUNE UNIVERSITY

**The Mini Project Based On**

**“Fast String Matching”**

**Submitted By:**

**Aniket Santosh Sinhasane**

**Seat No: B400260166**

**Under Guidance of:**

**Prof. J.K.Kalbhor**

**In partial fulfillment of**

**Laboratory Practice-V(410255 )**

**DEPARTMENT OF COMPUTER ENGINEERING) SAVITRIBAI PHULE PUNE UNIVERSITY 2024-25**

# 

# CERTIFICATE

This is to certify that the Mini Project based on,

**“Fast String Matching”**

has been successfully completed by, Name: Aniket Santosh Sinhasane

Exam seat number: B400260166

Towards the partial fulfilment of the Final Year of Computer Engineering as awarded by the Savitribai Phule Pune University, at PDEA’s College of Engineering, Manjari Bk,” Hadapsar, Pune 412307, during the academic year 2024-25.

**Prof. J.K.Kalbhor Dr. M. P. Borawake Guide Name H.O.D**

# Acknowledgement

My first and for most acknowledgment is to my supervisor and guide Prof. J.K.Kalbhor . During the long journey of this study, she supported me in every aspect. She was the one who helped and motivated me to proposer search in this field and inspired me with her enthusiasm on research, her experience, and her lively character.

I express true sense of gratitude to my guide Prof J.K.Kalbhor. for her perfect valuable guidance, all the time support and encouragement that he gave me.

I would also like to thanks our head of department Dr. M. P. Borawake and Principal Dr. R. V. Patil and management inspiring me and providing all lab and other facilities, which made this mini project very convenient.

I thankful to all those who rendered their valuable help for successful completion on Internship presentation.

Name: Aniket Santosh Sinhasane

|  |  |  |
| --- | --- | --- |
| **INDEX** | | |
| **Sr. No.** | **CONTENT** | **Page No.** |
| 1. | Abstract | 05 |
| 2. | Software Requirement | 06 |
| 3. | Introduction | 07 |
| 4. | Problem Statement | 08 |
| 5. | Objective and Outcome | 09 |
| 6. | Implementation of Code | 10-11 |
| 7. | Output | 12 |
| 8. | Future Scope | 13 |
| 9. | Conclusion | 14 |
| 10. | Reference | 15 |
| 4 | | |

# ABSTRACT

String matching is a core problem in computer science with widespread applications in areas such as information retrieval, bioinformatics, and natural language processing. This project investigates and implements fast string matching algorithms to efficiently locate patterns within large bodies of text. Traditional brute-force approaches are computationally expensive, especially with large inputs, making optimized algorithms essential. In this study, we implemented and analyzed several advanced string matching techniques including Knuth-Morris-Pratt (KMP), Boyer-Moore, and Rabin-Karp. These algorithms were evaluated based on their time complexity, space requirements, and practical performance across various test datasets. The results highlight the advantages and trade-offs of each algorithm, demonstrating that the choice of algorithm should be based on specific use cases such as pattern length, text size, and the number of patterns. The findings of this project provide a deeper understanding of efficient pattern searching techniques and offer guidance for selecting suitable algorithms in real-world applications.

The report includes both implementations and compares their execution times on random datasets. Results show a significant reduction in time using the parallel version, especially for large inputs. This demonstrates that parallelism can substantially enhance sorting efficiency when designed carefully. Ultimately, this study illustrates the practical benefits of parallel programming and how the multiprocessing API in Python can be used to scale traditional algorithms. With minimal changes to existing logic, parallelism allows us to make better use of system resources and shorten computation time in data-intensive environments.

# SOFTWARE REQUIREMENT



* Windows .
* Frontend Framework: React, Angular, or Vue.js
* Tools: Dev c++
* System: Windows 11.

# INTRODUCTION

String matching, also known as pattern matching, is a fundamental problem in computer science that involves finding one or more occurrences of a pattern (substring) within a larger body of text. It plays a critical role in numerous applications such as text editing, search engines, DNA sequence analysis, plagiarism detection, and intrusion detection systems.

The simplest method of string matching—known as the brute-force or naive approach—compares the pattern with the text at every possible position. While easy to implement, it is highly inefficient for large texts or multiple patterns, with a time complexity of O(nm)O(nm)O(nm), where nnn is the length of the text and mmm is the length of the pattern.

To address this inefficiency, several fast string matching algorithms have been developed. These algorithms use preprocessing techniques, heuristics, or data structures to reduce the number of comparisons and speed up the search process. Among the most well-known are:

* **Knuth-Morris-Pratt (KMP)**: Uses a prefix table to avoid redundant comparisons.
* **Boyer-Moore**: Utilizes bad character and good suffix heuristics to skip sections of the text.
* **Rabin-Karp**: Applies a rolling hash function to quickly detect potential matches.

This project focuses on the implementation and comparative analysis of these fast string matching algorithms. The goal is to understand their theoretical foundations, evaluate their performance on real and synthetic datasets, and determine their suitability for different types of applications.

# PROBLEM STATEMENT



## Evaluate Performance Enhancements of Parallel Quicksort Algorithm Using API

**Objective:** To understand the concept of Mini-project.

**Outcome:** Implement Evaluate Performance Enhancements of Parallel Quicksort Algorithm Using API

# OBJECTIVES

The objective of this study is to analyze and evaluate the performance benefits of implementing a parallel version of the Quicksort algorithm using Python’s multiprocessing API. Quicksort is fast and commonly used, but traditional implementations fail to leverage the true power of modern multi-core processors.

1. Objective 1: Implement Sequential Quicksort

The first goal is to build a working sequential version of the Quicksort algorithm. This version is implemented using standard recursion in Python and serves as our performance baseline. Understanding its functionality will help highlight the improvements introduced later.

1. Objective 2: Build a Parallel Version Using multiprocessing

The second objective is to develop a new version using Python's multiprocessing API. The recursive Quicksort logic remains the same, but after partitioning, the sublists are sorted in parallel using multiple processes. This enables better use of CPU cores and substantially speeds up processing when handling large datasets.

1. Objective 3: Compare Execution Times

We measure and benchmark both versions of the algorithm over data inputs ranging from 10,000 to 1,000,000 elements. The Python `time` module is used to calculate execution time differences accurately. Evaluating time improvements demonstrates clearly how and when parallelism provides actual benefits.

**FUTURE SCOPE:**

While this project focuses on classic fast string matching algorithms and their practical performance, there are several directions for future enhancement and research:

.

1.Parallel and Distributed Implementations

With the growing size of datasets, especially in areas like genomics and big data analytics implementing parallelized string matching using multithreading, GPUs (CUDA/OpenCL), or distributed systems (e.g., Apache Hadoop, Spark) can significantly reduce runtime.

2.Approximate String Matching

Real-world applications such as spell checkers, DNA sequence alignment, and fuzzy search often require matching patterns that may not be exactly the same. Algorithms like Levenshtein Distance, Bitap (Shift-Or), and dynamic programming approaches can be explored for this purpose.

3.Machine Learning Integration

Emerging research explores using machine learning to predict frequent patterns or dynamically choose the best string matching algorithm based on input characteristics. Integrating ML models could optimize performance in real-time systems.

4.Support for Multilingual and Unicode Texts

As global applications increasingly handle diverse languages and encodings, string matching algorithms must efficiently process Unicode characters and right-to-left scripts.

5.Enhanced Preprocessing Techniques

More sophisticated preprocessing can be developed for dynamic patterns or texts that frequently change, reducing the overhead of recomputing match structures like prefix tables or hash values.

# CONCLUSION

String matching is a vital operation in computer science with applications across diverse domains such as text processing, bioinformatics, and cybersecurity. This project examined and implemented several fast string matching algorithms—Knuth-Morris-Pratt, Boyer-Moore, and Rabin-Karp—and evaluated their performance on various datasets.

The results demonstrated that no single algorithm is universally optimal; rather, each has strengths depending on factors such as pattern length, alphabet size, and the nature of the input data. Boyer-Moore performed best with long patterns and large alphabets, while KMP offered consistent performance with minimal preprocessing. Rabin-Karp was particularly effective for multiple pattern searches, albeit with some trade-offs in hash collision handling.

Overall, the project highlights the importance of algorithm selection in optimizing performance for specific use cases. The insights gained lay a strong foundation for further exploration into advanced string matching techniques, including approximate matching and parallel processing.

# Reference

* https://[www.github.com](http://www.github.com/)
* https://chat.openai.com/
* Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). \*Introduction to Algorithms\*. MIT Press.
* Python Software Foundation. (2023). \*multiprocessing — Process-based parallelism\*.

Retrieved from https://docs.python.org/3/library/multiprocessing.html

* Knuth, D. E. (1998). \*The Art of Computer Programming, Volume 3: Sorting and Searching\*. Addison-Wesley.
* McCool, M., Robison, A.D., & Reinders, J. (2012). \*Structured Parallel Programming: Patterns for Efficient Computation\*. Morgan Kaufmann.