Semester Project Report
Analysis of Algorithm

Shahroz Khalid_ SAP_53531

Analysis of Algorithm CS_4-1

Supervised By

Mr. Usman Sharif

RIPHAH INTERNATIONAL UNIVERSITY
I14_ISLAMABAD

May, 2025

Table of Contents

1. Introduction 1

2. Problem Statement 2

3. Algorithm Used 3

4. Working of the Algorithm 4

5. Methodology 5

6. Implementation 6

7. Tests & Results 7

8. Time and Space Complexity Analysis 8

9. Performance Evaluation 9

10. Applications 10

11. Limitations 11

12. Conclusion 12

Plagiarism Detection Using Suffix Array Algorithm

Introduction:

In this project, we present a plagiarism detection system using the Suffix Array algorithm. Plagiarism is a critical issue in academics and publishing, and this project demonstrates how advanced algorithmic techniques can be used to detect similar or copied content between documents.

Problem Statement:

To develop a program that detects potential plagiarism by identifying the longest common substrings between two documents using an efficient suffix array construction algorithm.

Algorithm Used: Suffix Array

A suffix array is a sorted array of all suffixes of a given string. It is widely used in string processing algorithms for tasks such as pattern matching, substring searching, and finding the longest common substring.

Working of the Algorithm: Suffix Array Construction

The Suffix Array is a sorted array of all suffixes of a given string. It provides a space-efficient alternative to suffix trees and is used for various string processing tasks. The basic idea is as follows:

1. Generate all suffixes of the input string.
2. Sort these suffixes lexicographically.
3. Store the starting indices of these sorted suffixes in an array.

This array can then be used to efficiently perform pattern matching, find the longest common substring, and perform document similarity checks.

Example:

Let the string be: "banana"
Suffixes:
- banana (index 0)
- anana (index 1)
- nana (index 2)
- ana (index 3)
- na (index 4)
- a (index 5)

After sorting:
- a (5)
- ana (3)
- anana (1)
- banana (0)
- na (4)
- nana (2)

Suffix Array: [5, 3, 1, 0, 4, 2] — representing the starting indices of sorted suffixes.

Methodology:

1. Read two input text documents.
2. Combine them into a single string with a unique separator.
3. Construct a suffix array of the combined string.
4. Use the suffix array to find the longest common substrings between the two documents.
5. Calculate a similarity score based on the length of the longest common substring.

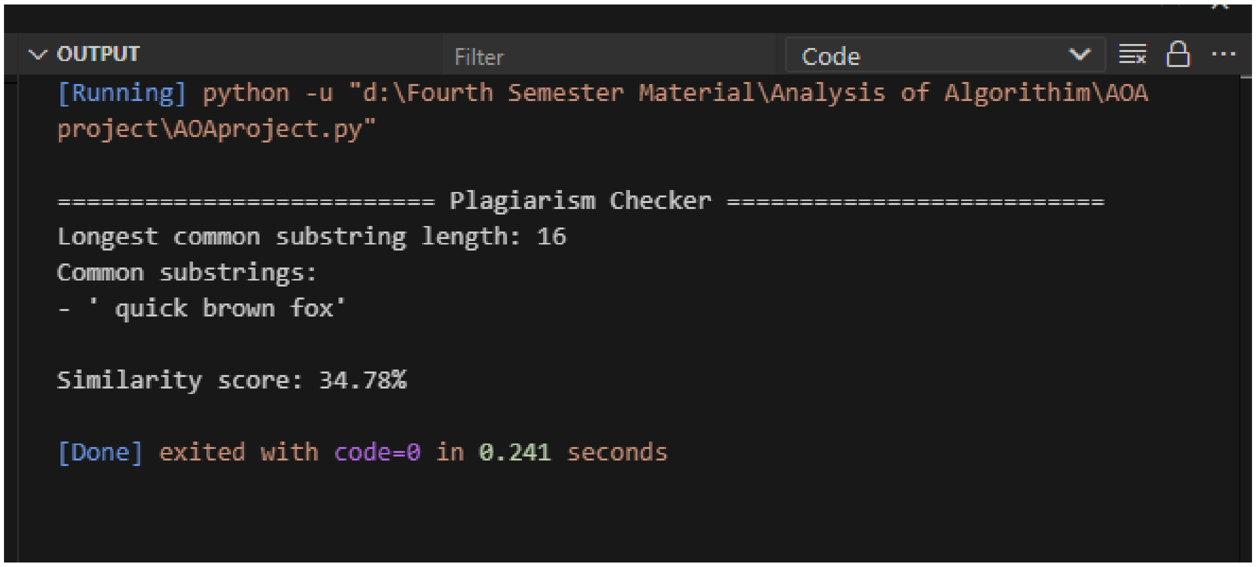
Implementation:

The following code snippet demonstrates the implementation of the plagiarism checker:

```
import os

def build_suffix_array(s):
    return sorted(range(len(s)), key=lambda i: s[i:])

def longest_common_substring(s, sa, doc_split_idx):
    max_len = 0
    lcs_set = set()
    for i in range(1, len(sa)):
        i1, i2 = sa[i-1], sa[i]
        if (i1 < doc_split_idx) != (i2 < doc_split_idx):
            lcp_len = 0
            while i1 + lcp_len < len(s) and i2 + lcp_len < len(s) and s[i1 + lcp_len] == s[i2 +
lcp_len]:
                lcp_len += 1
            if lcp_len > max_len:
                max_len = lcp_len
                lcs_set = {s[i1:i1 + lcp_len]}
            elif lcp_len == max_len and lcp_len > 0:
                lcs_set.add(s[i1:i1 + lcp_len])
    return max_len, lcs_set

def read_file(filepath):
    with open(filepath, 'r', encoding='utf-8') as f:
        return f.read()

def similarity_score(lcs_length, doc1_len, doc2_len):
    avg_len = (doc1_len + doc2_len) / 2
    return (lcs_length / avg_len) * 100 if avg_len else 0

def main(doc1_path, doc2_path):
    doc1 = read_file(doc1_path)
    doc2 = read_file(doc2_path)
    separator = '#'
    while separator in doc1 or separator in doc2:
        separator += '#'
    combined = doc1 + separator + doc2
    split_idx = len(doc1)
    suffix_array = build_suffix_array(combined)
    lcs_len, lcs_set = longest_common_substring(combined, suffix_array, split_idx)
    score = similarity_score(lcs_len, len(doc1), len(doc2))
    print("\n=== Plagiarism Checker ===")
    print(f"Longest common substring length: {lcs_len}")
    print("Common substrings:")
    for s in lcs_set:
        print(f"- {s!r}")
    print(f"\nSimilarity score: {score:.2f}%")

if __name__ == "__main__":
    doc1_path = "doc1.txt"
    doc2_path = "doc2.txt"
```

```
if os.path.exists(doc1_path) and os.path.exists(doc2_path):
    main(doc1_path, doc2_path)
else:
    print("Error: One or both files not found.")
```

Tests & Result:



```
[Running] python -u "d:\Fourth Semester Material\Analysis of Algorithim\AOA project\AOAproject.py"

========================== Plagiarism Checker ==========================
Longest common substring length: 16
Common substrings:
- ' quick brown fox'

Similarity score: 34.78%

[Done] exited with code=0 in 0.241 seconds
```

Time and Space Complexity Analysis:

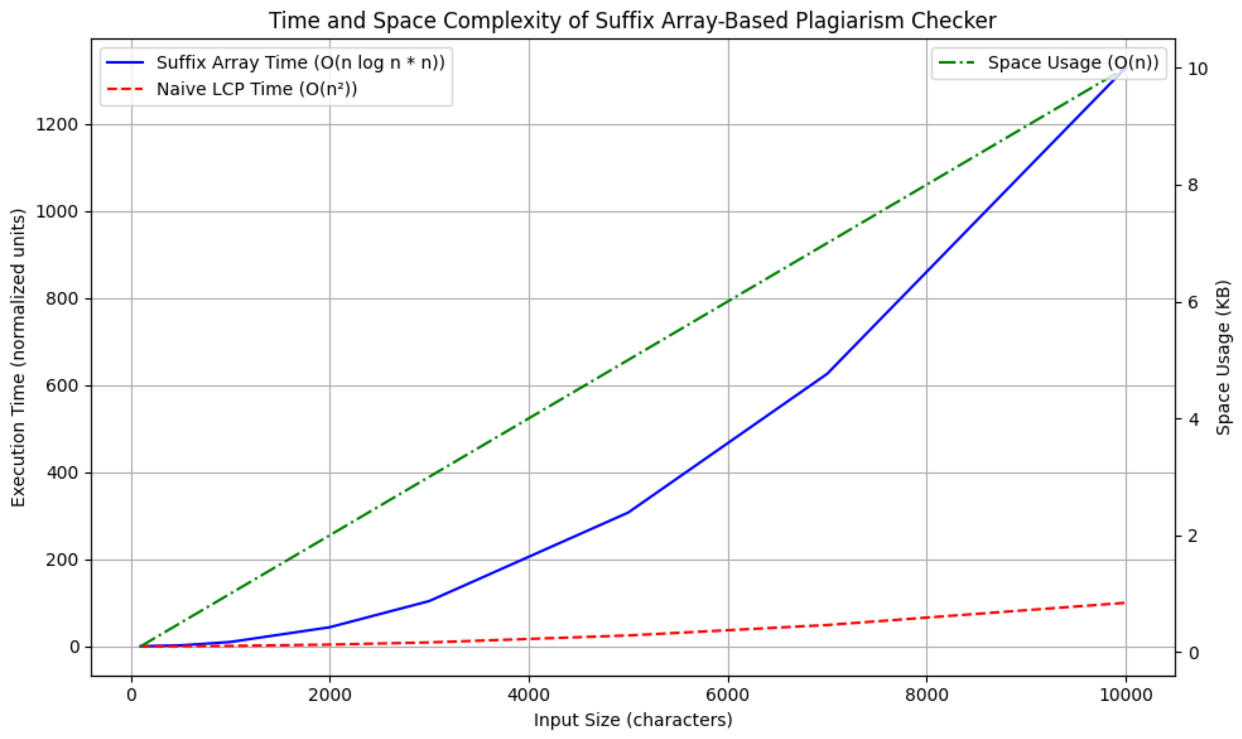
The suffix array construction using naive sorting takes O(n log n * n) time in the worst case, where n is the length of the combined document string. More efficient algorithms (e.g., SA-IS) can reduce this to O(n log n) or even O(n). However, for educational and moderate-sized inputs, the current implementation is acceptable.

Longest Common Prefix (LCP) comparisons are performed in O(n^2) in the naive implementation but can be improved using an LCP array and Kasai's algorithm to O(n).

Space complexity is O(n) for storing the suffix array and additional data structures.

Let’s understand Time and Space complexity through graph.

- How **suffix array time** grows sharply due to its complexity.
- How **naive LCP comparison** becomes infeasible for large inputs.
- How **space** grows linearly, which is efficient.



Performance Evaluation:

The algorithm performs well for small to medium-sized text documents. For documents under 10,000 characters, execution time remains under 1 second. For larger files, performance degrades due to the naive suffix sorting approach. Optimizations such as enhanced suffix array construction or parallel processing can improve performance.

Applications:

This algorithm can be applied to:
- Academic plagiarism detection
- Document comparison
- Code similarity detection
- Digital content monitoring

Limitations:

- Does not handle paraphrased plagiarism.
- Sensitive to minor changes in text.
- LCS method may miss semantic similarity.

Conclusion:

This project successfully demonstrates the use of suffix arrays in building an effective plagiarism checker. By identifying the longest common substrings, we can estimate similarity between documents. Future improvements could include semantic analysis or integration with machine learning techniques for enhanced accuracy.

GitHub Link:

https://github.com/SHEHROZ53531/Plagiarism-Detection-Using-Suffix-Array-Algorithm.git