Plagiarism Detection System

Algorithm design & analysis project
CS315 - Group :5262
Instructor: Dr. Renad Alsweed

Group Members:
Sarah Alharbi - 431201842
Marah Alshebel - 441203542
Budur Aljohani - 432205315
Elaf Aldobian - 432205184
Lama Alomair - 422205736

Table of Contents

1. Problem Identification

1.1 Introduction

In this project, we are working on detecting how similar two texts are in order to identify possible plagiarism. Plagiarism Detection is working by comparing a submission file against a selected document within subjects (scientific, literature, and academic) This process is important in academic settings because it helps determine whether a student's work is original or taken from existing sources.

To Implement this, we use two different string-matching techniques: a naive Algorithm and an optimized Algorithm.

1.2 Problem Description

The main goal of this project is to detect plagiarism in academic documents by comparing a submission file directly against a selected document within a categorized collection (scientific, literature, academic).

1.3 Importance of the Problem

- Helps maintain academic integrity and prevent cheating.
- Saves time and effort for instructors and reviewers.
- Can be implemented in universities and educational platforms.
- Improves the overall quality of research and writing.

1.4 Key Challenges

- Variation in data volume: Submissions and library documents vary in length.
- Text variation: Plagiarism may occur through paraphrasing rather than direct copying.
- Speed vs accuracy: Balancing detection accuracy with efficient performance.

2. Algorithm Selection

2.1 Naive Algorithm

Compares every word in the submission file with every word in the document directly.
Drawback: Very slow for large datasets

2.2 Optimized Algorithm — Rabin-Karp

Uses string hashing to convert text segments into numerical hash values.
Advantage: Much faster and more efficient, especially for large document.

2.3 Input & Output

Input: A submission file and a chosen category tag and a chosen length.

Output: Percentage similarity of the submission against a document in that category, and length.

2.4 Examples

First Example :

Input: Submission file, and a folder tag, and chosen length.

File: DataSet/small/Academic/document.txt

Submission: DataSet/small/Academic/submission.txt

document.txt → "The process of photosynthesis converts light energy into chemical energy in plants. Chlorophyll captures sunlight to produce glucose and oxygen."

submission.txt → "Photosynthesis in plants converts light energy into chemical energy.

Chlorophyll captures sunlight and produces glucose as a result."

Output:

Naive approach Output:

- Plagiarism Detection - Naive Approach
- DataSet Size: small
- Subject: Academic
- Window Size: 8
- Similarity score: 68%
- Time: 0.12 seconds

Rabin Karp Approach Output:

- Plagiarism Detection - Rabin Karp Approach
- DataSet Size: small
- Subject: Academic
- Window Size: 8
- Similarity score: 68%
- Time: 0.01 seconds

Second Example :

Input: Submission file, folder tag, and chosen length.

File: DataSet/small/Science/document.txt
Submission: DataSet/small/Science/submission.txt

document.txt → "Plants convert sunlight into usable energy through photosynthesis."
submission.txt → "Photosynthesis allows plants to turn sunlight into energy."

Output:

Naive approach Output:

- ❖ Plagiarism Detection - Naive Approach
- ❖ DataSet Size: small
- ❖ Subject: Science
- ❖ Window Size: 8
- ❖ Similarity score: 14%
- ❖ Time: 0.19 seconds

Rabin Karp Approach Output:

- ❖ Plagiarism Detection - Rabin Karp Approach
- ❖ DataSet Size: small
- ❖ Subject: Science
- ❖ Window Size: 8
- ❖ Similarity score: 14%
- ❖ Time: 0.02 seconds

Third Example :

Input: Submission file, folder tag, and chosen length.

File: DataSet/small/Literature/document.txt
Submission: DataSet/small/Literature/submission.txt

document.txt → "The old library held stories whispered through dusty shelves."
submission.txt → "Dusty shelves in the old library carried forgotten stories."

Output:

Naive approach Output:

- ❖ Plagiarism Detection - Naive Approach
- ❖ DataSet Size: small
- ❖ Subject: Literature
- ❖ Window Size: 8
- ❖ Similarity score: 3%
- ❖ Time: 0.17 seconds

Rabin Karp Approach Output:

- ❖ Plagiarism Detection - Rabin Karp Approach
- ❖ DataSet Size: small
- ❖ Subject: Literature
- ❖ Window Size: 8
- ❖ Similarity score: 3%
- ❖ Time: 0.09 seconds

3. Algorithm Development

3.1 Naive Algorithm – Pseudocode

The naive approach in our implementation compares each window of words in the submission file with every possible window of the same size in the document.This ensures accuracy but becomes slow as the document grows because every window is checked manually.

Pseudocode:

```
function NaivePlagiarismDetection(submission_words, document_words, window_size):
    matches = 0
    total_windows = length(submission_words) - window_size + 1
    for i from 0 to total_windows - 1:
        sub_window = join words from submission_words[i to i + window_size]
        for j from 0 to (length(document_words) - window_size):
            doc_window = join words from document_words[j to j + window_size]
            if sub_window == doc_window:
                matches = matches + 1
                break
    plagiarism_percentage = (matches / total_windows) * 100
print("Similarity:", plagiarism_percentage)
```

Explanation:

- Strengths:
    - Straightforward, easy to understand and implement.
    - Compares actual text windows directly, no hashing needed.
- Limitations:
    - Very slow for medium and large datasets because every window is compared manually.
    - Repeated building of window strings increases runtime.
- Use Case:
    - Works for small datasets or for demonstrating the basic idea of plagiarism checking.

3.2 Optimized Algorithm – Rabin–Karp Pseudocode

The optimized algorithm improves performance by converting each window of text into a hash value.Instead of comparing text directly, the algorithm compares hash values first, and only checks full text when hashes match (to avoid collisions).

Pseudocode:

```
function RabinKarpPlagiarismDetection(submission_words, document_words, window_size):
    doc_hashes = empty dictionary
    for j from 0 to (length(document_words) - window_size):
        doc_window = join words from document_words[j to j + window_size]
        doc_hash = calculate_hash(doc_window)
        add doc_window to doc_hashes[doc_hash]
    matches = 0
    total_windows = length(submission_words) - window_size + 1
    for i from 0 to total_windows - 1:
        sub_window = join words from submission_words[i to i + window_size]
        sub_hash = calculate_hash(sub_window)
        if sub_hash in doc_hashes:
            if sub_window in doc_hashes[sub_hash]:
                matches = matches + 1
    plagiarism_percentage = (matches / total_windows) * 100
                print("Similarity:", plagiarism_percentage)
```

Explanation:

- Strengths:
    - Much faster than the naive approach, especially for large datasets.
    - Hashing reduces repeated comparisons and speeds up matching significantly.
- Limitations:
    - Hash collisions can occur, requiring extra verification.
    - Slightly more complex due to hashing and dictionary use.
- Use Case:
    - Suitable for real plagiarism detection where speed is required for large files.

4. Theoretical Analysis

4.1 Naive Algorithm

- Description:

This algorithm compares each window of words in the submission with every possible window of the same size in the document. It checks for exact matches without any hashing or optimization. Best suited for small datasets or demonstration purposes.

PSEUDOCODE:	**#OPERATIONS**
FUNCTION NAIVEPLAGIARISMDETECTION(SUBMISSION_WORDS, DOCUMENT_WORDS, WINDOW_SIZE):	1
MATCHES = 0	1
TOTAL_WINDOWS = LENGTH(SUBMISSION_WORDS) -WINDOW_SIZE + 1	1
FOR I FROM 0 TO TOTAL_WINDOWS - 1:	n
SUB_WINDOW = JOIN WORDS FROM SUBMISSION_WORDS[I TO I + WINDOW_SIZE]	w
FOR J FROM 0 TO (LENGTH(DOCUMENT_WORDS) - WINDOW_SIZE):	m
DOC_WINDOW = JOIN WORDS FROM DOCUMENT_WORDS[J TO J + WINDOW_SIZE]	w
IF SUB_WINDOW == DOC_WINDOW:	w
MATCHES = MATCHES + 1	1
BREAK	1
PLAGIARISM_PERCENTAGE = (MATCHES / TOTAL_WINDOWS) *100	1
PRINT("SIMILARITY:", PLAGIARISM_PERCENTAGE)	1

- Complexity Analysis:

- Time Complexity (per document): **O(n * m * w)**

  n = number of windows in submission

  m = number of windows in document

  w = window size (words per window)

- Time Complexity (all documents): **O(D * n * m * w)**

- Space Complexity: **O(w)** (stores only current submission and document windows).

- Explanation:

The algorithm manually constructs each window from both submission and document, and compares them character by character. Ensures that all possible matches are counted. Very simple and easy to implement, but inefficient for medium or large datasets.

- Correctness:

Correct because it checks all possible windows for exact matches. Each match increments the counter only once (break ensures no double-counting in the same document window).No false positives occur since comparison is direct.

4.2 Rabin-Karp - Optimized Algorithm

- Description:

Converts each window of words into a hash value and stores document windows in a hash table. Submission windows are hashed and checked against the document hash table. Only verifies full window text when hashes match to avoid false positives due to collisions. Much faster than naive for large datasets.

PSEUDOCODE:	**#OPERATIONS**
FUNCTION RABINKARPPLAGIARISMDETECTION(SUBMISSION_WORDS, DOCUMENT_WORDS, WINDOW_SIZE):	1
DOC_HASHES = EMPTY DICTIONARY	1
FOR J FROM 0 TO (LENGTH(DOCUMENT_WORDS) - WINDOW_SIZE):	m
DOC_WINDOW = JOIN WORDS FROM DOCUMENT_WORDS[J TO J + WINDOW_SIZE]	w
DOC_HASH = CALCULATE_HASH(DOC_WINDOW)	w
ADD DOC_WINDOW TO DOC_HASHES[DOC_HASH]	1
MATCHES = 0	1
TOTAL_WINDOWS = LENGTH(SUBMISSION_WORDS) - WINDOW_SIZE + 1	1
FOR I FROM 0 TO TOTAL_WINDOWS - 1:	n
SUB_WINDOW = JOIN WORDS FROM SUBMISSION_WORDS[I TO I + WINDOW_SIZE]	w
SUB_HASH = CALCULATE_HASH(SUB_WINDOW)	w
IF SUB_HASH IN DOC_HASHES:	1
IF SUB_WINDOW IN DOC_HASHES[SUB_HASH]:	k
MATCHES = MATCHES + 1	1
PLAGIARISM_PERCENTAGE = (MATCHES / TOTAL_WINDOWS) * 100	1
PRINT("SIMILARITY:", PLAGIARISM_PERCENTAGE)	1

- Complexity Analysis:

- Time Complexity (per document): **O((n + m) * w)**

  n = number of windows in submission

  m = number of windows in document

  w = window size

- Time Complexity (all documents): **O(D * (n + m) * w)** average case

- for worst case it can be (n * m * w) where all collisions collide, but it rarely happen.

- Space Complexity: **O(m * w)** (hash table storing document windows) + O(w) temporary window

- Explanation:

Hashing reduces the number of direct string comparisons. Only windows with matching hashes are checked fully, which speeds up the algorithm. Efficient for large documents and many windows.

- Correctness:

Correct because every potential match is verified through direct comparison after hash match. Hash collisions are handled properly, ensuring no false positives. Each match is counted accurately.

5. Implementation

5.1 Naive Approach (Algorithm1.py)

```
import time

# Intial information

Window_Size = 8 # The number of words to compare at once

# Choose size: "small" ~1200-1700 characters, "medium" ~4k-5k characters, "large"
~22k-30k characters
DataSet_Size = "small"

# Choose subject: "Academic", "Literature", "Science"
Subject = "Academic"

# Set file paths for dataset and subject
Submission_File = f"DataSet/{DataSet_Size}/{Subject}/submission.txt"
Document_File = f"DataSet/{DataSet_Size}/{Subject}/document.txt"

# Read submission file
submission_file = open(Submission_File, 'r', encoding='utf-8')
submission_text = submission_file.read()
submission_file.close()

# Split the submission text into lowercase words
submission_words = submission_text.lower().split()

# Read document file
document_file = open(Document_File, 'r', encoding='utf-8')
document_text = document_file.read()
document_file.close()

# Split the document text into lowercase words
document_words = document_text.lower().split()

# Start timing from here
start_time = time.time()

matches = 0 # Counting matches
```

```
total_windows = len(submission_words) - Window_Size + 1 # How many chunks of
text can be created from the submission

# Go through each window (chunks of text) in submission
for i in range(total_windows):
  # Get words from submission
  sub_window = ""
  for j in range (Window_Size):
    sub_window = sub_window + submission_words[i + j] + " "
  sub_window = sub_window.strip()

  # Compare with each window (chunks of text) in document
  found_match = False
  for k in range(len(document_words) - Window_Size + 1):
    # Get words from document
    doc_window = ""
    for m in range(Window_Size):
      doc_window = doc_window + document_words[k + m] + " "
    doc_window = doc_window.strip()

    # Check if they match :
    if sub_window == doc_window:
      matches = matches + 1
      found_match = True
      break

# Calculate similarity percentage
if total_windows > 0:
  similarity = (matches / total_windows) * 100
else:
  similarity = 0.0

# Stop timing
end_time = time.time()
total_time = end_time - start_time

print("Plsigirism Detection - Naive Approach")
print(f"DataSet size: {DataSet_Size}")
print(f"Subject: {Subject}")
print(f"Window size: {Window_Size}")
```

```
print(f"comparing: {Submission_File} to {Document_File}")
print(f"Similarity Score: {similarity: .2f}%")
print(f"Time: {total_time: .4f} seconds")
```

5.2 Optimized Approach - Rabin Karp (algorithm2.py)

```
import time

# Intial information

Window_Size = 8 # The number of words to compare at once

# Choose size: "small" ~1200-1700 characters, "medium" ~4k-5k characters, "large"
~22k-30k characters
DataSet_Size = "small"

# Choose subject: "Academic", "Literature", "Science"
Subject = "Academic"

# Set file paths for dataset and subject
Submission_File = f"DataSet/{DataSet_Size}/{Subject}/submission.txt"
Document_File = f"DataSet/{DataSet_Size}/{Subject}/document.txt"

Prime = 101 # Prime number for hashing (to keep the hash value between 0-100)
Base = 256 # Base for hash calculation (ASCII has 256 possible values)

# Function to calculate hash of a text
def calculate_hash(text):
    hash_value = 0
    position = 0
    for char in text:
        hash_value = hash_value + ord(char) * pow(Base, position)
        hash_value = hash_value % Prime
        position = position + 1
    return hash_value

# Read submission file
submission_file = open(Submission_File, 'r', encoding='utf-8')
submission_text = submission_file.read()
submission_file.close()
```

```
# Split the submission text into lowercase words
submission_words = submission_text.lower().split()

# Read document file
document_file = open(Document_File, 'r', encoding='utf-8')
document_text = document_file.read()
document_file.close()

# Split the document text into lowercase words
document_words = document_text.lower().split()

# Start timing from here
start_time = time.time()

# Step 1: Pre-compute all document hashes once and store them
doc_hashes = {}
for j in range(len(document_words) - Window_Size + 1):
    # Get words from document
    doc_window = ""
    for k in range(Window_Size):
        doc_window = doc_window + document_words[j + k] + " "
    doc_window = doc_window.strip()

    # Calculate hash for document
    doc_hash = calculate_hash(doc_window)

    # Store hash in window text
    if doc_hash not in doc_hashes:
        doc_hashes[doc_hash] = []
    doc_hashes[doc_hash].append(doc_window)

# Step 2 : Check submission windows against document hashes (already stored)
matches = 0 # Counting matches
total_windows = len(submission_words) - Window_Size + 1 # How many chunks of
text can be created from the submission

for i in range(total_windows):
  # Get words from submission
  sub_window = ""
  for j in range (Window_Size):
```

```
    sub_window = sub_window + submission_words[i + j] + " "
  sub_window = sub_window.strip()

  # Calculate hash for submission
  sub_hash = calculate_hash(sub_window)

  #check if submission hash exits in doucment hash
  if sub_hash in doc_hashes:
     # Verify if the actual text match as well (not just hash) for hash collisions
     if sub_window in doc_hashes[sub_hash]: # Condition to check if submission text
in dictionary of document hashes
        matches = matches + 1

# Calculate similarity percentage
if total_windows > 0:
  similarity = (matches / total_windows) * 100
else:
  similarity = 0.0

# Stop timing
end_time = time.time()
total_time = end_time - start_time

print("Plsigirism Detection - Rabin-Karp Approach")
print(f"DataSet size: {DataSet_Size}")
print(f"Subject: {Subject}")
print(f"Window size: {Window_Size}")
print(f"comparing: {Submission_File} to {Document_File}")
print(f"Similarity Score: {similarity: .2f}%")
print(f"Time: {total_time: .4f} seconds")
```

6. Empirical Analysis

6.1 Execution Result Tables

To assess the performance of both the naive and optimized plagiarism-detection algorithms, we conducted a series of empirical tests across multiple dataset sizes, each dataset consisted of three subjects (Academic, Literature, and Science) provided in small (~1200-1700 characters), medium (4k-5k characters) , and large (~22k-30k characters) versions.

For every file, we measured the **execution time** required by each algorithm to process the input and compute the similarity score, below are the result:

Academic Table

Sample Size	Naive(s)	Optimized(s)	Similarity
Small	0.1297	0.0183	93.20%
Medium	0.8595	0.0271	40.24%
Large	36.0208	0.1722	0.18%

(Execution Result)

Literature Table

Sample Size	Naive(s)	Optimized(s)	Similarity
Small	0.0691	0.0107	7.84%
Medium	1.7274	0.0222	0.16%
Large	46.1127	0.1330	2.23%

(Execution Result)

Science Table

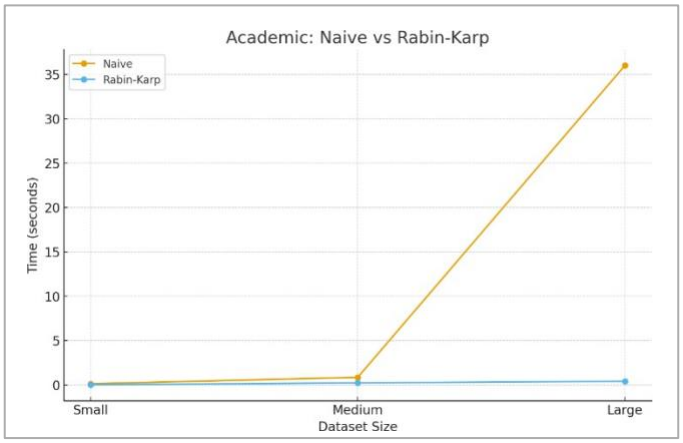
Sample Size	Naive(s)	Optimized(s)	Similarity
Small	0.0833	0.0055	22.00 %
Medium	0.6471	0.0189	0.00 %
Large	26.0274	0.1437	0.39%

(Execution Result)

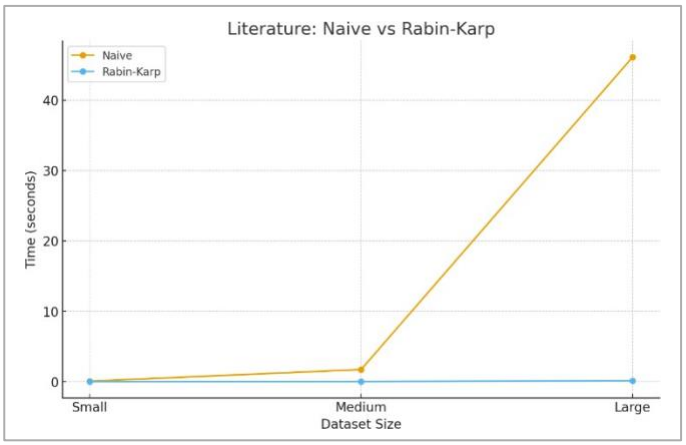
6.2 Graphs

The Comparison clearly show big difference in speed between the naive and optimized algorithms, as the file size increases, the naive algorithm becomes much slower because it repeats many comparisons.

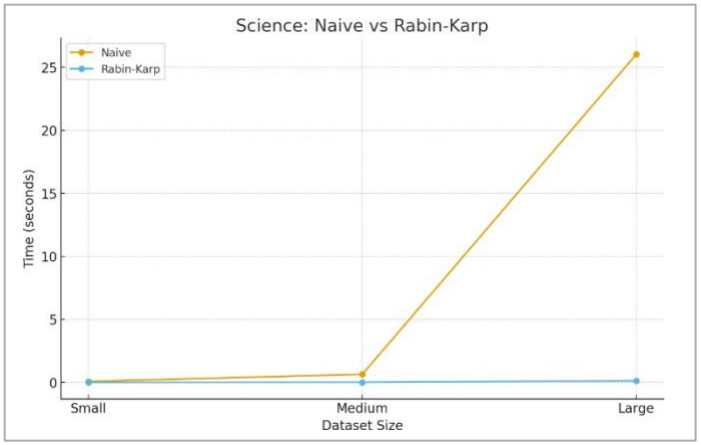
In contrast, the optimized algorithm stays fast, even with larger inputs because it handles the text more efficiently, the optimized algorithm performs better while still giving the same similarity result, below are the result:



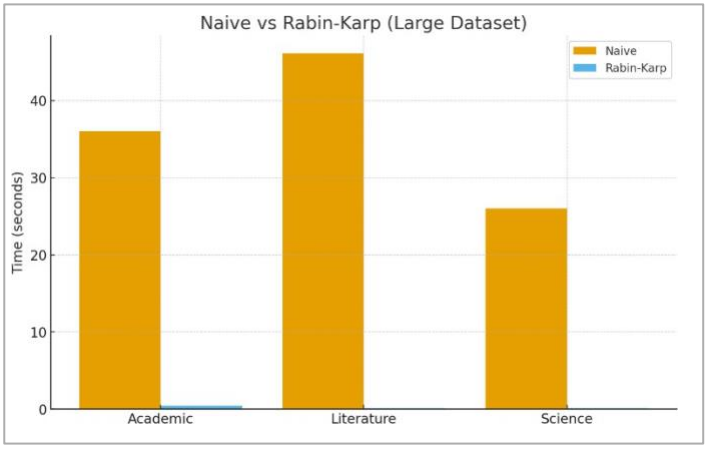
Execution Time Comparison (Line Graph) 1



Execution Time Comparison (Line Graph) 2



Execution Time Comparison (Line Graph) 3



Execution Time Comparison (Bar Chart)

7. Results Comparison

7.1 Comparison Between Theoretical and Empirical Analysis

The theoretical analysis predicted that the Naive algorithm would have a time complexity of O(n*m *w), while the Rabin-Karp optimized algorithm would have a time complexity of O((n+m) * w), And this was strongly confirmed by the empirical results.

For example, in the Academic dataset:

· Small Size: Naive took 0.1297s, Rabin-Karp took 0.0183s.

· Large Size: Naive took 36.0208s, Rabin-Karp took 0.1722s.

The difference in speed gets much larger as the input size increases, confirming the theoretical expectations. The Rabin-Karp algorithm scales runtime nearly linearly, making it far more efficient for large documents.

7.2 Discrepancies

There are some discrepancies between the theoretical and empirical results, although they are relatively minor:

1- Hashing overhead in Rabin-Karp:

While Rabin-Karp is theoretically faster, the hashing function adds small extra amout of work, This can cause the optimized approach to take longer than expected on very small inputs, even though it performs better than the Naive method overall

2- Data preprocessing and string operations:

In both algorithms, converting text to lowercase, splitting into words, window construction overhead adds a small amount of time. This overhead is not considered in the theoretical analysis

8. Conclusion & Reference

8.1 Conclusion:

The Naive algorithm, despite its simplicity, was chosen to demonstrate the fundamental idea of plagiarism detection. It is ideal for educational purposes or small datasets where clarity and correctness matter more than speed. The Rabin-Karp optimized algorithm was selected for scenarios where efficiency is crucial. By using hashing, it significantly reduces unnecessary comparisons while maintaining accuracy. This makes it suitable for large datasets, real-time plagiarism checking in academic or professional environments, and automated content verification systems.

In essence, the Naive method serves as a learning tool to understand the mechanics of window-based matching, while Rabin-Karp is a practical solution for real-world applications where performance and scalability are key.

8.2 References:

1. GeeksforGeeks. (2025). *Rabin–Karp algorithm for pattern searching*. https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/
2. Karp, R. M., & Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development, 31*(2), 249–260. https://doi.org/10.1147/rd.312.0249
3. Giometti, H. (2020, September 16). *Solving popular algorithms: String matching*. Medium. https://hopegiometti.medium.com/solving-popular-algorithms-string-matching-57eca7ee737f