

**Analysis and Design of Algorithms**

**CSCE 2202-02 - Spring 2021**

# **Final Project Report**

**Simple Plagiarism Detection Utility using String  
Matching**

Ahmed Essam Abdelaleem 900193476

Iman Ahmed Attia 900192510

Mariam Ramadan Ali 900191779

Rawan Sameh Hamad 900192388

## Abstract

Plagiarism has become a serious ethical violation, frequently committed by some students and researchers. It is a crime in the sense of academic integrity. Thus, several online tools and softwares are now available to detect such a violation. These softwares depend mainly on string matching algorithms that detect the occurrences of some patterns inside other external texts or resources. As students, we usually deal with some plagiarism detectors such as turnitin.com to check the plagiarism percentage in our papers. Therefore, we decided to utilize the knowledge that we have gained in the Analysis and Design of Algorithm course to simulate the plagiarism detection process.

**Keywords:** Plagiarism - Brute Force - Hamming Distance - Rabin Karp Algorithm - KMP Algorithm - BoyerMoore Algorithm - Time Complexity - String Matching

## Introduction

In our project, we are required to detect the plagiarism in a specific pattern from a given document, through different approaches. The input will be a determined number of patterns, with existing documents which can include some sentences from the pattern or not. The project can execute the following tasks: detecting the plagiarized patterns, highlighting the documents that the plagiarism takes place from, and calculating the plagiarism percentage of the pattern from each document and the total percentage of plagiarism. The project is executing its function using four different algorithms to compare between the time complexity of each algorithm and choose the better one. The project's performance is determined by the accuracy of the plagiarism percentage detected on a specific pattern on each document besides the total percentage.

## Problem Definition

The project depends on four main algorithms to detect the string matching between the pattern and the document: Hamming Distance, Rabin Karp Algorithm, KMP Algorithm, and Boyer-Moore Algorithm. Generally, we take the input pattern as a set of sentences which are separated by dots, while taking the input documents as a set of strings; each string stores one of the documents as a whole. The strategy is to loop on every sentence in the pattern and detect if it was plagiarized or not by looping again on every document in our database. At this point, the output can be represented by the following style: (assuming we have 2 patterns, each pattern has 3 sentences, and we have 3 documents)

	Document 1	Document 2	Document 3
Pattern 1	(true, false, false)	(false, false, false)	(true, true, false)
Pattern 2	(false, false, false)	(true, true, true)	(false, false, true)

In the table above, we can see that when Pattern 1 is compared with Document 1, the output is (true, false, false), which means that the 1st sentence in the pattern is detected as plagiarism

while the two remaining sentences are not. Consequently, we apply several approaches to improve the output and increase the project efficiency. Further details will be discussed in the methodology section.

## Methodology

Here we will present the methodology that we have followed throughout our project in order to solve the given problem and detect the plagiarism percentage from each text document and even detect the overall plagiarism percentage:

### I. Initialization stage

First of all, we calculate the number of characters stored in the pattern document during the initialization of the file in the function named *taking\_input\_pattern(string pat\_file\_name)* and store the total number of characters in a global variable named *pattern\_char\_total*. Also, inside the same function, the *pattern* vector was initialized, containing all the sentences of the file where each sentence (*pattern[idx]* for example) is treated as a potential pattern. Besides, we use the function *taking\_input\_text(string txt\_file\_name, int idx)* to initialize the vector *text* of type string and size of *num\_text\_files* to contain the text documents, where each document is stored as a single string.

### II. Testing Algorithms

Then, inside the *test\_algorithms(int algorithm\_num)* function, we loop over both the text documents (outer loop) and the patterns (inner loop) using a nested for loop then based on the number of the specified algorithm, we call one of the four string matching algorithm to detect whether the elements of the pattern exist in the text documents or not. We store the results of the matching or mismatching in a boolean vector of the same size as the pattern vector, named *palagrized\_or\_not*. This vector was initially set to false. Further, the length of the total number of characters plagiarized from each text is stored in a *palagrized\_length* vector whose size is equal to the number of the text documents (*num\_text\_files*). If the string matching algorithm returned true, i.e. a match occurred, the value stored in the index of the boolean vector will be changed to true, detecting a plagiarism of the corresponding sentence in the *pattern* vector, and the *palagrized\_length[i]* is incremented by the length of the plagiarized sentence in the pattern (i.e. *palagrized\_length[i] += pattern[j].size()*; noticing that *i* is the outer for loop counter that loops over the text files and *j* is the inner for loop counter that loops over the elements of the pattern vector).

### III. Plagiarism percentage from each text document (using the length approach)

Whenever, we exit the inner for loop, in other words, we finish comparing the sentences of pattern with one of the text documents, we calculate the plagiarism percentage detected from this single text file by dividing the length of the plagiarized sentences obtained from this file by the total length of the pattern characters that was calculated at the very beginning. Thus, we update the variable *palagrisim\_percent* that was declared as double and initialized to be zero as follows:

```
palagrisim_percent = (palagrized_length[i] / pattern_char_total) * 100;
```

And then we display its value on the screen.

#### IV. Total plagiarism percentage from all the text documents

To calculate the overall plagiarism percentage for a single pattern document, we loop over the boolean vector *palagirized\_or\_not* and check if *palagirized\_or\_not[i]* is true (i.e. the sentence is plagiarized), we increment a variable of type double, named *total\_percent* by the value of *pattern[i].length* divide by the value of *pattern\_char\_total* and making sure both are of data type double. Finally, after the execution of the for loop we multiply the value of *total\_percent* by 100 to obtain the total plagiarism in percentage.

## V. Threshold of plagiarism percentage

Moreover, we have a plagiarism threshold; up to a value of 50% for the *total\_percent* is not considered to be a plagiarism case; however, exceeding a 50% is regarded as a plagiarism and in this case a message is displayed to the user pointing out the plagiarism and the occurrence of academic dishonesty.

## VI. Capitalization

In our program, we put in our consideration the capitalization of the sentences and the fact that c++ is a case sensitive language which considers an “algorithm” different from an “ALGORITHM” for example. Thus, we have written the function *make\_lowercase(string in)* to change the inputs we read from the files to lowercase, and thus overcoming the case sensitivity obstacle.

### Specification of Algorithms to be used

In the project, we used the four mentioned algorithms for string matching to implement our approach for the seek of detecting whether the pattern is taken from the text or not. We will discuss the mechanism of each algorithm in details:

### 1. The Brute Force Algorithm using hamming distance

**Pseudo code:**

```

n ← length [T]
m ← length [P]
for i ← 0 to n - m do
  if P[1 .. m] = T[i + 1 .. i + m]
  then return valid
shift i

```

Text : A A B A A C A A D A A B A A B A A  
Pattern : A A B A

A A B A                      A A B A

A A B A A C A A D A A B A A B A A  
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

A A B A

Pattern Found at 0, 9 and 12

The brute force approach, or the naive approach, is simply taking the first character of the pattern and comparing it to the first one of the text. If there is a match, it will compare the next pattern characters. If a mismatch happens, the pattern will slide over the text by one index and

repeat the process again. The plagiarism is detected when all of the pattern characters are matched in some part in the text.

## 2. Rabin-Karp Algorithm

**Pseudo code:** (consider hashT & hashP are hash structures for Text & Pattern)

*hash\_p = hash value of pattern*

*hash\_t = hash value of first M letters in body of text*

*do*

*if (hash\_p == hash\_t)*

*brute force comparison of pattern and selected section of text*

*hash\_t = hash value of next section of text, one character over*

*while (end of text or brute force comparison == true)*

The Rabin-Karp approach depends on using the hashing technique for comparing two strings. As shown in the pseudo code, the algorithm calculates the hash values of the pattern and the current substring of the text as a pre-step. If the hashing matches, then it starts to compare between each character. This algorithm will do the same work of naive approach, but with quicker steps.

## 3. Knuth-Morris-Pratt Algorithm

**Pseudo code:**

*f ← compute failure function of Pattern P*

*// The failure function f(j) equals the length of the longest prefix of P that is a suffix of P[i . . j].*

*i ← 0*

*j ← 0*

*while i < length[T] do*

*if j ← m-1 then*

*return i - m + 1 // we have a match*

*i ← i + 1*

*j ← j + 1*

*else if j > 0*

*j ← f(j - 1)*

*else*

*i ← i + 1*

The KMP algorithm uses so-called degenerating property which means detecting equal sub-patterns appearing in the pattern to use it when a mismatch happens. The idea behind this algorithm is taking advantage of the sub-strings that we already determine through our search, and do not return back to the first character of the pattern when there is a mismatch.

#### 4. Boyer-Moore Algorithm

**Pseudo code:**

```
i ← m - 1
j ← m - 1
Repeat
  If P[j] = T[i] then
    if j = 0 then
      return i // we have a match
    else
      i ← i - 1
      j ← j - 1
    else
      i ← i + m - Min(j, 1 + last[T[i]])
      j ← m - 1
until i > n - 1
Return "no match"
```

The Boyer-Moore algorithm is considered better than the naive algorithm since it does preprocessing over the pattern so that the pattern can be shifted by more than once like the KMP algorithm. The distinct technique in this algorithm is comparing between two strings starting from the last character of the pattern. When there is a mismatch in some character, the pattern will slide to the last occurrence of this character. Hence, it ignores many unimportant steps while searching.

### Input Data Specifications

**Input details:**

- We have created 5 text documents of different sizes and different contents. The content of the 5 documents is the lyrics of 5 different songs as follows:
  - a. "text\_0.txt": Let her go - passenger
  - b. "text\_1.txt": Shape of you - Ed Sheeran
  - c. "text\_2.txt": The nights - Avicii
  - d. "text\_3.txt": Astronaut In The Ocean - Masked Wolf
  - e. "text\_4.txt": Windmills of Your Mind - Noel Harrison and Those Were The Days (Remastered 1991) - Mary Hopkin
- The sizes of the documents are ascendingly increasing in size; "text\_0.txt" has the least size where "text\_4.txt" has the largest size.
- We have created 2 pattern documents: "pattern\_0.txt" and "pattern\_1.txt". We intentionally built the first from a collection of sentences taken from the first 4 text documents to see a detected plagiarism case. On the other hand, the second pattern document was all unique; it was a plagiarism-free case.

### Process of reading the files:

- The text documents are read through the function *taking\_input\_text(string txt\_file\_name, int idx)*. The function takes the name of the text file to be tested and the index in which the input of the named text file will be stored later in the *text* vector. The function loops over the lines of the text file and concatenates all of them in a single string in *text[idx]*.
- The pattern documents are read through the function *taking\_input\_pattern(string pat\_file\_name)* which takes the name of the pattern file to be tested and then goes through the lines of the file and stores the sentences separated by dots ('.') by pushing it back in the *pattern* vector.
- The results of our program may have a percentage error of +0.2% represented in the dots, because the program may accidentally consider it as a pattern itself.

## Experimental Results

### I. Results of Plagiarism Detection

The following table shows the results of the 4 plagiarism-detection algorithms:

Used Algorithm	Pattern files names	Text files names					Total plagiarism Percent
		text_0	text_1	text_2	text_3	text_4	
Brute Force using Hamming Distance	pattern_0	13.96 %	14.19 %	36.38 %	10.3 %	0 %	74.83 %
	pattren_1	0.1773 %	0.1773 %	0.1773 %	0.1773 %	0.1773 %	0.1773 %
Rabin Karp (RK)	pattern_0	13.96 %	14.19 %	36.38 %	10.3 %	0 %	74.83 %
	pattren_1	0.1773 %	0.1773 %	0.1773 %	0.1773 %	0.1773 %	0.1773 %
Knuth-Morris-Pratt (KMP)	pattern_0	13.96 %	14.19 %	36.38 %	10.3 %	0 %	74.83 %
	pattren_1	0.1773 %	0.1773 %	0.1773 %	0.1773 %	0.1773 %	0.1773 %
Boyer Moore	pattern_0	13.96 %	14.19 %	36.38 %	10.3 %	0 %	74.83 %
	pattren_1	0.1773 %	0.1773 %	0.1773 %	0.1773 %	0.1773 %	0.1773 %

**Comment on the results:** as we can notice from the data collected in the table above, the four algorithms produced the same percentage of plagiarism, which indicates that the four algorithms are working properly without any differentiation.

## II. Results of the performance in sense of the execution time (time taken in seconds)

Used Algorithm	Pattern files names	Text files names				
		text_0	text_1	text_2	text_3	text_4
Brute Force using Hamming Distance	pattern_0	0.002	0.003	0.003	0.009	0.023
	pattren_1	0.002	0.003	0.005	0.012	0.026
Rabin Karp (RK)	pattern_0	0.005	0.005	0.008	0.011	0.033
	pattren_1	0.004	0.006	0.005	0.013	0.03
Knuth-Morris-Pratt (KMP)	pattern_0	0.008	0.009	0.011	0.033	0.117
	pattren_1	0.012	0.017	0.024	0.047	0.118
Boyer Moore	pattern_0	0.001	0.001	0.001	0.003	0.008
	pattren_1	0.001	0.001	0.002	0.003	0.008

**Comment on the results:** we tried to compare the performance of the 4 algorithms by calculating the execution time of each algorithm and the results show a kind of convergence. However, to draw a concise conclusion, the algorithms needs ro be tested on a huge number of files with very large sizes.

## III. Results of the performance in sense of the number of comparisons

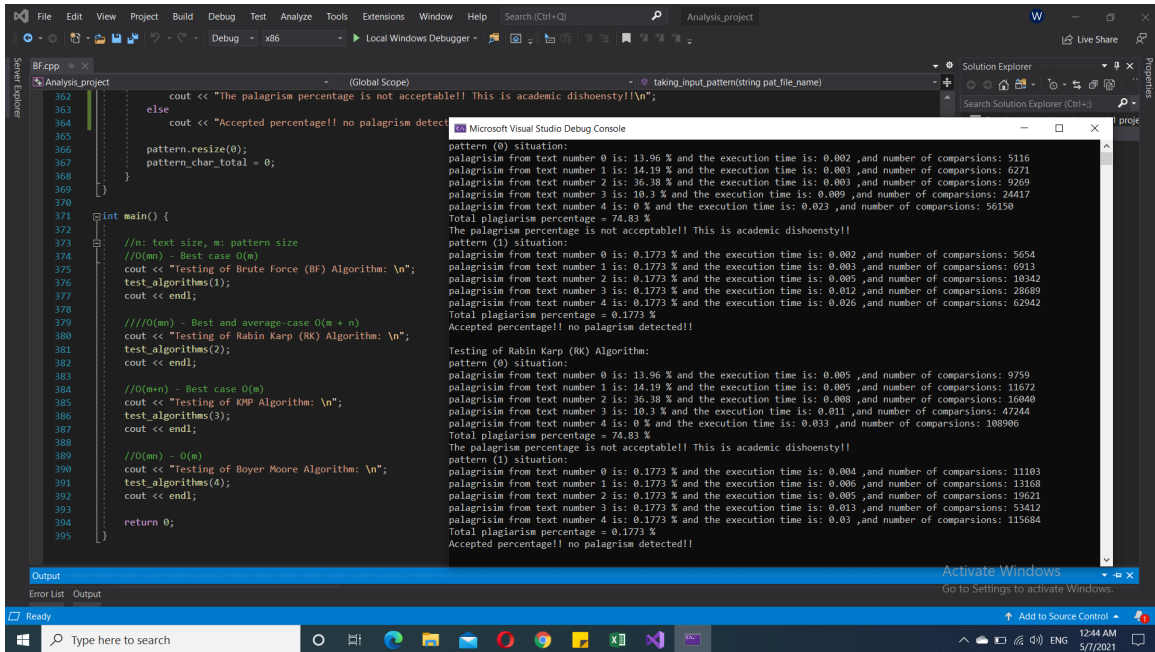
Used Algorithm	Pattern files names	Text files names				
		text_0	text_1	text_2	text_3	text_4
Brute Force using Hamming Distance	pattern_0	5116	6271	9269	24417	56150
	pattren_1	5654	6913	10342	28689	62942
Rabin Karp (RK)	pattern_0	9759	11672	16040	47244	108906
	pattren_1	11103	13168	19621	53412	115684
Knuth-Morris-Pratt (KMP)	pattern_0	29973	35384	46779	141843	324718
	pattren_1	35144	41167	60445	160711	345123
Boyer Moore	pattern_0	1817	2244	3227	8421	21247
	pattren_1	1861	2218	3298	8537	21660



**Comment on the results:** we used another method to compare the performance of the 4 algorithms which is by calculating the number of comparisons each algorithm takes. The results are consistent with the results obtained from calculating the execution time. Both show a very similar behavior.

**Screenshots for the outputs taken from the Debug Console of Microsoft Visual Studio:**

A) For Brute Force and Rabin Karp (RK) Algorithms:



```
362 cout << "The plagiarism percentage is not acceptable!! This is academic dishonesty!!\n";
363
364 else
365     cout << "Accepted percentage!! no plagiarism detected!!\n";
366
367     pattern.resize(0);
368     pattern_char_total = 0;
369
370
371 int main() {
372
373     //n: text size, m: pattern size
374     //O(m) - Best case O(m)
375     cout << "Testing of Brute Force (BF) Algorithm: \n";
376     test_algorithms(1);
377     cout << endl;
378
379     //O(mn) - Best and average-case O(m * n)
380     cout << "Testing of Rabin Karp (RK) Algorithm: \n";
381     test_algorithms(2);
382     cout << endl;
383
384     //O(mn) - Best case O(m)
385     cout << "Testing of KMP Algorithm: \n";
386     test_algorithms(3);
387     cout << endl;
388
389     //O(mn) - O(m)
390     cout << "Testing of Boyer Moore Algorithm: \n";
391     test_algorithms(4);
392     cout << endl;
393
394     return 0;
395 }
```

Microsoft Visual Studio Debug Console

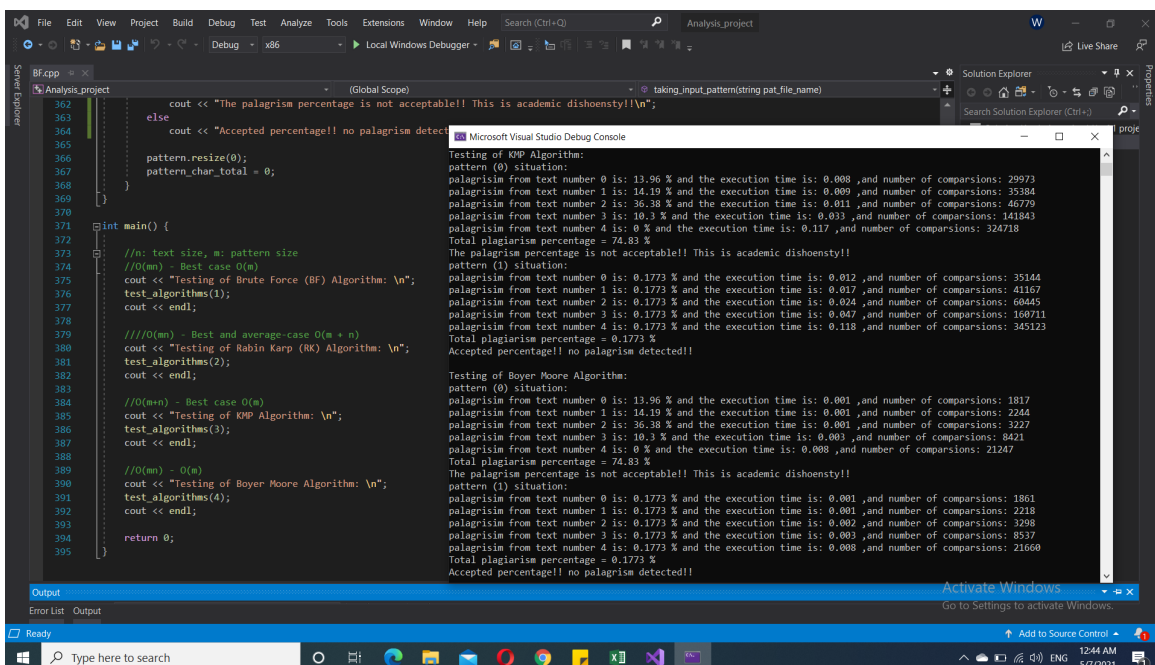
pattern (0) situation:  
palagrism from text number 0 is: 13.96 % and the execution time is: 0.002 ,and number of comparisons: 5116  
palagrism from text number 1 is: 14.19 % and the execution time is: 0.003 ,and number of comparisons: 6271  
palagrism from text number 2 is: 36.38 % and the execution time is: 0.003 ,and number of comparisons: 9269  
palagrism from text number 3 is: 10.3 % and the execution time is: 0.009 ,and number of comparisons: 24417  
palagrism from text number 4 is: 0 % and the execution time is: 0.023 ,and number of comparisons: 56150  
Total plagiarism percentage = 74.83 %  
The plagiarism percentage is not acceptable!! This is academic dishonesty!!

pattern (1) situation:  
palagrism from text number 0 is: 0.1773 % and the execution time is: 0.002 ,and number of comparisons: 5654  
palagrism from text number 1 is: 0.1773 % and the execution time is: 0.003 ,and number of comparisons: 6913  
palagrism from text number 2 is: 0.1773 % and the execution time is: 0.005 ,and number of comparisons: 10342  
palagrism from text number 3 is: 0.1773 % and the execution time is: 0.012 ,and number of comparisons: 28689  
palagrism from text number 4 is: 0.1773 % and the execution time is: 0.026 ,and number of comparisons: 62942  
Total plagiarism percentage = 0.1773 %  
Accepted percentage!! no plagiarism detected!!

Testing of Rabin Karp (RK) Algorithm:  
pattern (0) situation:  
palagrism from text number 0 is: 13.96 % and the execution time is: 0.005 ,and number of comparisons: 9759  
palagrism from text number 1 is: 14.19 % and the execution time is: 0.005 ,and number of comparisons: 11672  
palagrism from text number 2 is: 36.38 % and the execution time is: 0.008 ,and number of comparisons: 16040  
palagrism from text number 3 is: 10.3 % and the execution time is: 0.011 ,and number of comparisons: 47244  
palagrism from text number 4 is: 0 % and the execution time is: 0.033 ,and number of comparisons: 108906  
Total plagiarism percentage = 74.83 %  
The plagiarism percentage is not acceptable!! This is academic dishonesty!!

pattern (1) situation:  
palagrism from text number 0 is: 0.1773 % and the execution time is: 0.004 ,and number of comparisons: 11103  
palagrism from text number 1 is: 0.1773 % and the execution time is: 0.006 ,and number of comparisons: 13168  
palagrism from text number 2 is: 0.1773 % and the execution time is: 0.005 ,and number of comparisons: 19621  
palagrism from text number 3 is: 0.1773 % and the execution time is: 0.013 ,and number of comparisons: 53412  
palagrism from text number 4 is: 0.1773 % and the execution time is: 0.03 ,and number of comparisons: 115684  
Total plagiarism percentage = 0.1773 %  
Accepted percentage!! no plagiarism detected!!

B) For Knuth-Morris-Pratt (KMP) and Boyer Moore Algorithms:



```
362 cout << "The plagiarism percentage is not acceptable!! This is academic dishonesty!!\n";
363
364 else
365     cout << "Accepted percentage!! no plagiarism detected!!\n";
366
367     pattern.resize(0);
368     pattern_char_total = 0;
369
370
371 int main() {
372
373     //n: text size, m: pattern size
374     //O(m) - Best case O(m)
375     cout << "Testing of Brute Force (BF) Algorithm: \n";
376     test_algorithms(1);
377     cout << endl;
378
379     //O(mn) - Best and average-case O(m * n)
380     cout << "Testing of Rabin Karp (RK) Algorithm: \n";
381     test_algorithms(2);
382     cout << endl;
383
384     //O(mn) - Best case O(m)
385     cout << "Testing of KMP Algorithm: \n";
386     test_algorithms(3);
387     cout << endl;
388
389     //O(mn) - O(m)
390     cout << "Testing of Boyer Moore Algorithm: \n";
391     test_algorithms(4);
392     cout << endl;
393
394     return 0;
395 }
```

Microsoft Visual Studio Debug Console

Testing of KMP Algorithm:  
pattern (0) situation:  
palagrism from text number 0 is: 13.96 % and the execution time is: 0.008 ,and number of comparisons: 29973  
palagrism from text number 1 is: 14.19 % and the execution time is: 0.009 ,and number of comparisons: 35394  
palagrism from text number 2 is: 36.38 % and the execution time is: 0.011 ,and number of comparisons: 46770  
palagrism from text number 3 is: 10.3 % and the execution time is: 0.033 ,and number of comparisons: 141843  
palagrism from text number 4 is: 0 % and the execution time is: 0.117 ,and number of comparisons: 324718  
Total plagiarism percentage = 74.83 %  
The plagiarism percentage is not acceptable!! This is academic dishonesty!!

pattern (1) situation:  
palagrism from text number 0 is: 0.1773 % and the execution time is: 0.012 ,and number of comparisons: 35144  
palagrism from text number 1 is: 0.1773 % and the execution time is: 0.017 ,and number of comparisons: 41167  
palagrism from text number 2 is: 0.1773 % and the execution time is: 0.024 ,and number of comparisons: 60445  
palagrism from text number 3 is: 0.1773 % and the execution time is: 0.047 ,and number of comparisons: 160711  
palagrism from text number 4 is: 0.1773 % and the execution time is: 0.118 ,and number of comparisons: 345123  
Total plagiarism percentage = 0.1773 %  
Accepted percentage!! no plagiarism detected!!

Testing of Boyer Moore Algorithm:  
pattern (0) situation:  
palagrism from text number 0 is: 13.96 % and the execution time is: 0.001 ,and number of comparisons: 1817  
palagrism from text number 1 is: 14.19 % and the execution time is: 0.001 ,and number of comparisons: 2244  
palagrism from text number 2 is: 36.38 % and the execution time is: 0.001 ,and number of comparisons: 3227  
palagrism from text number 3 is: 10.3 % and the execution time is: 0.003 ,and number of comparisons: 8421  
palagrism from text number 4 is: 0 % and the execution time is: 0.008 ,and number of comparisons: 21247  
Total plagiarism percentage = 74.83 %  
The plagiarism percentage is not acceptable!! This is academic dishonesty!!

pattern (1) situation:  
palagrism from text number 0 is: 0.1773 % and the execution time is: 0.001 ,and number of comparisons: 1861  
palagrism from text number 1 is: 0.1773 % and the execution time is: 0.001 ,and number of comparisons: 2218  
palagrism from text number 2 is: 0.1773 % and the execution time is: 0.002 ,and number of comparisons: 3298  
palagrism from text number 3 is: 0.1773 % and the execution time is: 0.003 ,and number of comparisons: 8537  
palagrism from text number 4 is: 0.1773 % and the execution time is: 0.008 ,and number of comparisons: 21600  
Total plagiarism percentage = 0.1773 %  
Accepted percentage!! no plagiarism detected!!

## Analysis and Critique

As shown in the results' section, the resulting percentages are equal from the four algorithms but there is a slight difference in the execution time and the number of comparisons taken in each algorithm. To explain that in details, we will elaborate on the complexities of each algorithm in the best, and worst case.

- Suppose that “ $m$ ” is the length of the pattern, while “ $n$ ” is the length of the text

### 1- Brute Force Algorithm

The worst case scenario will be applied if there is a permanent mismatch in only the last character in the pattern. For example, the text is “AAAAAAAAAAAA”, and the pattern is : “AAAZ”. In this case, the naive approach will compare each character of the pattern with its adjacent character in the text. Since the first  $m-1$  character is always matching.  $m$  comparisons will take place. Consequently, the last comparison will detect a mismatch, and the pattern will slide over the text by one index and repeat the process. Hence,

***In worst case:***

- Time complexity:  $O(mn)$

The best case scenario will appear when the pattern is matching with the text from the first index' comparison. For example, the text is “ABCDEFGHIJ”, and the pattern is “ABCD”. In this case, the naive comparison will take place one time on the characters on the pattern.

***In best case:***

- Time complexity:  $\Omega(m)$

### 2- Rabin-Karp Algorithm

As explained above, the Rabin-Karp algorithm depends on the hashing as a pre-step before doing the brute force comparisons. Therefore, the worst case scenario will appear when the prime numbers used in the hashing function are small or not sufficient. Hence, there will be collisions results in increasing the time complexity to  $O(mn)$

***In worst case:***

- Time complexity:  $O(mn)$

On the other hand, if there is a sufficient number of prime numbers that's used for the hashing function, this will decrease the collision and make the hashing value of two different strings usually distinct. In this case, searching complexity can take  $\Omega(n)$ . Comparing this algorithm with naive approach, the best case complexity is greater than the one in naive approach, but it's easier to deal with as it depends on the used prime numbers not the pattern content.

***In best case:***

- Time complexity:  $\Omega(n)$

### **3- Knuth-Morris-Pratt algorithm**

The KMP algorithm, like Rabin-Karp algorithm, is processing the prefixes of substrings as a pre-step to improve the time complexity of searching. Processing the input pattern and the table creation will take  $O(m)$  time. After that, the comparisons take place. The worst case scenario appears when there are so many mismatches that ignore the improvements which the prefixes seek to achieve. In this case, the time complexity will be  $O(m) + O(n)$

#### ***In worst case:***

- Time complexity:  $O(m + n)$

On the other hand, the best case scenario appears when the pattern is matching with the first adjacent substring of the text just after creating the prefix table. In this case, the time taken to make an output can be just  $\Omega(m)$

#### ***In best case:***

- Time complexity:  $\Omega(m)$

### **4- Boyer Moore algorithm**

Analysing with the same style, the Boyer Moore algorithm depends on the extra advantage that allows the pattern sliding over the text by more than one index. So, the worst case scenario appears when the pattern content neglects this improvement and forces the pattern to slide by one index every time.

#### ***In worst case:***

- Time complexity:  $O(mn)$

Ofcourse, the best case will be the same as the naive approach. This scenario will happen when the pattern is matching with the first-index substring of the text. Starting from the first or last index of the pattern will not affect the time taking for computing the algorithm  $\Omega(m)$ . In conclusion, the results of naive approach and Boyer Moore algorithm may seem equal, but the average case in the Boyer algorithm is  $O(n)$ , which is smaller than the brute force.

#### ***In best case:***

- Time complexity:  $\Omega(m)$

## **Conclusions**

Plagiarism is a serious problem that's been commonly committed in many fields, such as research, study, and journalism. Our focus of this research was to investigate several string matching algorithms and see their potentials in solving such a problem. To accomplish this, we relied on 4 string matching algorithms which are the Brute Force algorithm, Rabin Karp algorithm, Knuth-Morris-Pratt (KMP) algorithm, and finally the Boyer Moore algorithm. Following a certain mythology, we managed to calculate the plagiarism percentage from each file and we also calculated the total plagiarism percentage. The results showed the same plagiarism percentages for the 4 algorithms which indicates a successful implementation of all 4 algorithms. Last but not least, in this project, we utilized the knowledge we have learned in the CSCE 2202 course to perform a time complexity analysis for the used algorithms as well as comparing the execution time and the number of comparisons of the four string matching algorithms.

## Acknowledgements

Undoubtedly, we would like to thank god for giving us patience and the blessing of completing this project successfully.

In recognition of all his efforts with us throughout the semester and the benefit we gained from his website and materials, we would like to thank Dr. Amr Goneid for all of his support and encouragement.

In acknowledgement of his assistant and continuous support, we want to thank the graduate teacher assistant of this course, Ahmed Abdel Kareem as he always showed redinnes for answering all of our questions and provided us with valuable guidance and encouragement throughout the journey of working on this project.

Also, we are thankful for our parents and friends who have always been our first and last supporters of all times, the people who inspire us to persist and do our best.

Lastly, we are grateful for anyone who supported us in any respect until we reached a successful end of this project.

## References

- <https://www.cs.auckland.ac.nz/courses/compsci369s1c/lectures/GG-notes/CS369-StringAlgs.pdf>
- <https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap11.pdf>
- <https://brilliant.org/wiki/rabin-karp-algorithm/>
- <https://brilliant.org/wiki/knuth-morris-pratt-algorithm/>
- <https://brilliant.org/wiki/boyer-moore-algorithm/>

## Appendix

//Here is the full C++ code of our project:

```
#include <iostream>
#include <string>
#include <fstream>
#include <vector>
#include <sstream>
#include <iomanip>
#include <algorithm>
#include <bits/stdc++.h>
# define NO_OF_CHARS 256
using namespace std;

int num_pat_files = 2;
int num_text_files = 5;
vector <string> pattern;
vector <string> text(num_text_files);
int pattern_char_total = 0;
vector <int> counter(4, 0);

//BF algorithm for string matching - complexity O(nm)
bool BF(string pattern, string text) {
    int n = text.length();
    int m = pattern.length();

    for (int i = 0; (i < n - m + 1) || (m == n); i++) { // we faced a problem when the pattern and
the text are the same

        // i.e. n = m, e.g. ptrn = "Iman", txt = "Iman", so i added this condition
        int j = 0;

        while (j < m && counter[0]++ && text[i + j] == pattern[j]) {
            j++; //increment no. of matches
        }
        if (j == m) return true;
        //else return false;
    }
    return false;
}

bool RK(string pattern, string text)
{
```

```

int prime = 101; //the larger the prime number, the smaller the chance of collisions
int size_p = pattern.size();
int size_t = text.size();
int i, j;
int hash_p = 0; // hash value for pattern
int hash_t = 0; // hash value for text
int h = 1;

//  $h = d^{(m-1)}$ 
for (i = 0; i < size_p - 1; i++)
    h = (h * 256) % prime;

//calculate hash values
for (i = 0; i < size_p; i++)
{
    hash_p = (256 * hash_p + pattern[i]) % prime;
    hash_t = (256 * hash_t + text[i]) % prime;
}

// Sliding
for (i = 0; i <= size_t - size_p; i++)
{
    //if the hash values of the string and text match, then compare characters

    counter[1]++;
    if (hash_p == hash_t)
    {
        for (j = 0; j < size_p; j++)
        {
            counter[1]++;
            if (text[i + j] != pattern[j])
                break;
        }
        // if hash_p == hash_t and pattern[0...M-1] = text[i, ..., M-1]
        if (j == size_p)
            return true;
    }

    //calculate the hash value of the next window from the previous one
    if (i < size_t - size_p)
    {
        hash_t = (256 * (hash_t - text[i] * h) + text[i + size_p]) % prime;
        //in case hash_t is negative, we get its positive
        if (hash_t < 0)
            hash_t = (hash_t + prime);
    }
}

```

```

        }

    }
    return false;
}
//KMP
void computeLPSArray(string pat, int M, vector <int> lps)
{
    int len = 0;
    lps[0] = 0;
    int i = 1;
    while (i < M) {
        counter[2]++;
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        }
        else
        {
            if (len != 0) {
                len = lps[len - 1];
            }
            else
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}

```

```

bool KMPSearch(string pat, string txt)
{
    int M = pat.size();
    int N = txt.size();

    // create lps[] that will hold the longest prefix suffix
    // values for pattern
    vector <int> lps(M);
    // Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps);

```

```

int i = 0; // index for txt[]
int j = 0; // index for pat[]
while (i < N) {
    counter[2]++;
    if (pat[j] == txt[i]) {
        j++;
        i++;
    }
    if (j == M) {
        //printf("Found pattern at index %d ", i - j);

        j = lps[j - 1];
        return true;
    }
    // mismatch after j matches
    else if (i < N && counter[2]++ && pat[j] != txt[i]) {
        // Do not match lps[0..lps[j]-1] characters,
        // they will match anyway
        if (j != 0)
            j = lps[j - 1];
        //return true;}
        else
            i = i + 1;
        // return false;}
    }
}
return false;
}

//Boyer Moore algorithm for string matching - complexity O(nm)
void badCharHeuristic(string str, int size, int badchar[NO_OF_CHARS])
{
    int i;
    // Initialize all occurrences as -1
    for (i = 0; i < NO_OF_CHARS; i++)
        badchar[i] = -1;
    // Fill the actual value of last occurrence
    // of a character
    for (i = 0; i < size; i++)
        badchar[(int)str[i]] = i;
}

bool BoyerMoore(string pat, string txt)
{
    int m = pat.size();
    int n = txt.size();
    int badchar[NO_OF_CHARS];

```



```

badCharHeuristic(pat, m, badchar);

int s = 0;
while (s <= (n - m))
{
    int j = m - 1;
    while (j >= 0 && counter[3]++ && pat[j] == txt[s + j])
        j--;

    if (j < 0)
    {
        return true;
        //cout << "pattern occurs at shift = " << s << endl;
        s += (s + m < n) ? m - badchar[txt[s + m]] : 1;
    }

    else {
        s += max(1, j - badchar[txt[s + j]]);
    }
}
return false;
}

string make_lowercase(string in)
{
    string out;
    transform(in.begin(), in.end(), back_inserter(out), tolower);
    return out;
}

void taking_input_text(string txt_file_name, int idx) {
    ifstream file_text("C:/Users/Dell/Desktop/Spring 2021/Algorithms/Project/" +
txt_file_name + ".txt");
    string line;
    if (!file_text.is_open())
    {
        cout << "error opening the file" << endl;
        return;
    }
    else
    {
        while (getline(file_text, line, '\n')) {
            text[idx] = text[idx] + " " + (line);
        }
        text[idx] = make_lowercase(text[idx]);
        //cout << text[idx];
        //cout << text[idx].size();
    }
}

```

```

        file_text.close();
    }
}

void taking_input_pattern(string pat_file_name) {
    ifstream file_pattern("C:/Users/Dell/Desktop/Spring 2021/Algorithms/Project/" +
        pat_file_name + ".txt");
    string line;

    if (!file_pattern.is_open())
    {
        cout << "error opening the file" << endl;
        return;
    }
    else
    {
        int i = 0;
        while (getline(file_pattern, line, '\n')) {
            stringstream data(line);
            while (getline(data, line, '.'))
            {
                pattern.push_back(line);
                pattern_char_total += pattern[i].size();

                pattern[i] = make_lowercase(pattern[i]);

                //cout << pattern[i] << endl;
                i++;
            }
        }
        //cout << pattern.size();
        file_pattern.close();
    }
}

```

```

void test_algorithms(int algorithm_num) {

    string pat_files[] = { "pattern_0", "pattern_1" };
    string text_files[] = { "text_0", "text_1", "text_2", "text_3", "text_4" };

    for (int i = 0; i < num_text_files; i++) {
        taking_input_text(text_files[i], i);
    }
    for (int i = 0; i < num_pat_files; i++) {
        taking_input_pattern(pat_files[i]);
        cout << "pattern (" << i << ") situation: \n";
    }
}

```

```

vector <bool> palagirized_or_not(pattern.size(), false);
vector <double> palagrized_length(num_text_files);
double palagrisim_percent = 0;
for (int i = 0; i < num_text_files; i++) {
    double start_time = clock();
    for (int j = 0; j < pattern.size(); j++) {
        switch (algorithm_num) {
            case 1:
                if (BF(pattern[j], text[i]))
                {
                    palagrized_length[i] += pattern[j].size();
                    palagirized_or_not[j] = true;
                }
                break;
            case 2:
                if (RK(pattern[j], text[i]))
                {
                    palagrized_length[i] += pattern[j].size();
                    palagirized_or_not[j] = true;
                }
                break;
            case 3:
                if (KMPSearch(pattern[j], text[i]))
                {
                    palagrized_length[i] += pattern[j].size();
                    palagirized_or_not[j] = true;
                }
                break;
            case 4:
                if (BoyerMoore(pattern[j], text[i]))
                {
                    palagrized_length[i] += pattern[j].size();
                    palagirized_or_not[j] = true;
                }
                break;
            default:
                break;
        }
    }
    double end_time = clock();
    double execution_time = (double)(end_time - start_time) /
CLOCKS_PER_SEC;
    palagrisim_percent = (palagrized_length[i] / pattern_char_total) * 100;
    cout << "palagrisim from text number " << i << " is: " << setprecision(4) <<
palagrisim_percent << " % and the execution time is: "

```

```

        << execution_time << " ,and number of comparsions: "<<
counter[algorithm_num-1] << endl;
        for (int i = 0; i < counter.size(); i++) counter[i] = 0;
    }
    double total_percent = 0;
    for (int i = 0; i < pattern.size(); i++)
        if (palagirized_or_not[i])
            total_percent += (double(pattern[i].size()) /
double(pattern_char_total));
    total_percent *= 100;
    cout << "Total plagiarism percentage = " << total_percent << " %" << endl;
    if (total_percent > 50)
        cout << "The palagrism percentage is not acceptable!! This is academic
dishoensty!!\n";
    else
        cout << "Accepted percentage!! no palagrism detected!!\n";
    pattern.resize(0);
    pattern_char_total = 0;
}
}

int main() {
    //n: text size, m: pattern size
    //O(mn) - Best case O(m)
    cout << "Testing of Brute Force (BF) Algorithm: \n";
    test_algorithms(1);
    cout << endl;

    ///O(mn) - Best and average-case O(m + n)
    cout << "Testing of Rabin Karp (RK) Algorithm: \n";
    test_algorithms(2);
    cout << endl;

    //O(m+n) - Best case O(m)
    cout << "Testing of KMP Algorithm: \n";
    test_algorithms(3);
    cout << endl;

    //O(mn) - O(m)
    cout << "Testing of Boyer Moore Algorithm: \n";
    test_algorithms(4);
    cout << endl;

    return 0;}

```