**Report: Plagiarism Detection Utility Using String Matching Algorithms**

Essam ElKholy, Yahia Kilany, Yousef Sayed

Department of Computer Science and Engineering, AUC

CSCE 110/1101 : Fundamentals of Computing II, section: 02

Dr.Howaida Ismael

May, 14 2023

[**Abstract 3**](#_b75g2649lnlf)

**Introduction………………………………………………………………………………………………4**

[**Problem Definition 5**](#_a0jw4fy36n1y)

[**Methodology 6**](#_nllbtegikept)

[Classes 6](#_bimdi3699tee)

[Document 6](#_7agbwstx03t1)

[Corpus 6](#_hgcidvxnl0kk)

[Matcher 6](#_v7etnl1i01f8)

[Algorithm Classes 7](#_p724u9smnjr7)

[Usage inside program 7](#_uyxaars4jkd6)

[**Specification of Algorithms to be used 8**](#_qrregteaswyq)

[Brute Force Matcher using Hamming Distance 8](#_ih0jfshi4w89)

[Rabin Karp Algorithm 8](#_35inrng00wsm)

[KMP algorithm 9](#_d2w00tdeswrz)

[BoyeerMoore algorithm 9](#_wnuu7if208ho)

[**Experimental Results 10**](#_1l9urt8k9vqn)

[**Analysis and Critique 14**](#_wta3zo6d4ny6)

[**Conclusion 15**](#_xzqd9qp4efa1)

[**Acknowledgements 16**](#_wefp0ylwb0am)

[**References 17**](#_b9fj5bkwxs12)

[**Appendix A 18**](#_t168peza1agj)

[**Appendix B 21**](#_vfy1j5wmm90o)

# Abstract

Today's academic and professional contexts struggle with plagiarism. Different plagiarism checking programs have been created to prevent plagiarism, however how effective they are relies heavily on the algorithms they employ. The goal of this C++ programming project is to create a plagiarism detector that compares a test document to a corpus using four distinct algorithms. The brute force matcher employing Hamming distance, Boyer-Moore, Knuth-Morris-Pratt (KMP), and Rabin-Karp are the four algorithms employed in this research. The program will assess the time it takes for each algorithm to check for plagiarism and compare the quality of the findings that each algorithm produces. To identify which algorithm is the most successful and efficient in detecting plagiarism, the results will be analyzed and compared. The program will allow the comparison of two documents—a test document and a corpus—with the test document being the document that was possibly plagiarized from the corpus document. In order to detect plagiarism, the program will compare the two documents. The final product will be a plagiarism detector that can identify plagiarized content while providing useful information on the performance of various plagiarism detection algorithms. The project's overall goal is to support the development of effective techniques for detecting plagiarism that can support maintaining academic and professional integrity.

Keywords:Brute Force, Rabin karp, KMP, Boyer Moore, time complexity , space complexity, arthmetric hashing , preprocess , goodsuffix heuristic, corpus , plagiarism detection, Inheritance , polymorphism , abstraction

**Report: Plagiarism Detection Utility Using String Matching Algorithms**

String matching is a well known issue in the world of computer science and it includes finding instances of a pattern string inside a larger text string; the pattern string is usually called the sub string (Cormen et al., 2009). Data compression, information retrieval, and plagiarism detection are just a few of its practical applications (Eshwitha, 2022). Numerous algorithms have been created to effectively tackle the string matching issue. Some of the algorithms used to tackle this issue are Brute Force, KMP, Rabin Karp, Boyer Moore, Automaton Matcher, and Aho Corasick (Eshwitha, 2022). The purpose of string matching is to, basically, determine if a specific pattern appears in a text and the location of this pattern (Cormen et al., 2009). Characters from both the pattern and the text are compared during the procedure, often one at a time. The algorithms that we will be talking about are the BruteForce, KMP, Rabin Karb, and Boyer Moore.The brute-force algorithm using hamming distance, which includes comparing the pattern with every possible substring of the text, is the simplest method. However, the time complexity exponentially rises as the amount of the text and pattern increases, making it unsuitable for bigger datasets. As a result, various more effective algorithms, including the Boyer-Moore, KMP, and Rabin-Karp algorithms, have been created to address this issue. The Boyer Moore algorithm works by performing a preprocessing step when a mismatch occurs to skip ahead in the text string (Cormen et al., 2009). The KMP algorithm does the same thing to prevent meaningless comparisons by maintaining a partial match table (Cormen et al., 2009). Finally, Rabin Karp uses hashing to minimize the amount of character comparisons required (Cormen et al., 2009). Moreover, which algorithm is the most effective and efficient highly depends on the time complexity and space complexity of the algorithm used. According to the case that the algorithms are used in, their time and space complexities will vary and which one would be considered the best will be according to the case.

# Problem Definition

A significant problem that compromises the integrity of academic and professional work is plagiarism. Plagiarism is the act of using someone else's work or ideas without proper attribution. People may now steal ideas without giving credit far more easily than before because of the increased availability of electronic content. The efficiency of the numerous plagiarism detection methods that have been created to prevent plagiarism mostly depends on the algorithms employed. As a result, the goal of this project is to create a plagiarism detector that compares a sample test document to various corpus documents using four distinct approaches. The brute force matcher employing Hamming distance, Boyer-Moore, Knuth-Morris-Pratt (KMP), and Rabin-Karp are the four algorithms employed in this program. Each algorithm has particular benefits and drawbacks and may be used to identify plagiarism in various ways. The program will evaluate the efficiency and effectiveness of each algorithm, comparing the time taken to perform the plagiarism check and the quality of the results obtained.

# Methodology

The program contains 7 classes: *Document, Corpus, Matcher, BruteForceMatcher, RabinKarbMatcher, kmpMatcher, BoyerMooreMatcher*. The program is heavily depends on demonstrate the Object oriented programming and it’s four principles ( Abstraction , Inheritance , Polymorphism , Encapsulation) and the program tries to utilize these functionalities as much as possible.In addition, the program makes heavy use of the *String* class and *STL* datatypes such as: *Vectors* and *Maps*. We will first investigate these classes and move on to their usage inside the program.

## Classes

### Document

The *Document* class includes two private string variables ( *Title* and *Content*), the function contains setters and getters for these variables in addition to the *createFromFile(string)* which reads the contents and the name of a text file and assigns it to the member variables of the class.

### Corpus

The *Corpus* class includs a *vector* of *Document,* which results in a relationship of aggregation between the classes *Document* and *Corpus*. The class includes setters and getters and *addDocument(Document)* to add *Document* object into the *vector.*

### Matcher

An abstract class that is used as a superclass for the four algorithm class: *BruteForceMatcher, RabinKarbMatcher, kmpMatcher, BoyerMooreMatcher* which all inherent from the Matcher class publicly. The class contains two pure virtual functions *getMemoryUsage()* which returns the size of an object and fucntion *matcher* which is overridden in all of the child classes so that it can execute their respective algorithms and returns a *map* that contains the titles of documents that were plagarised and the similarity percentage. *splitIntoSentences* is a protected member that takes a *Document* and returns a vector that includes individual sentences in the file based on the

### Algorithm Classes

The four algorithm classes:*BruteForceMatcher, RabinKarbMatcher, kmpMatcher, BoyerMooreMatcher* ,inheret from the *Matcher* class. they implement their respective algorithms inside the *matcher* function. Some of these classes contain functions that serve as an abstraction of some functionalities of the main algorithm such as the *computeLBS()* algorithm in *kmpMatcher* class.

## Usage inside program

The program starts by converting text files into *Document* objects that then either added to a corpus, or used as a test document. Then objects of each algorithm class are initialized and the match function is called and the time that the function takes is measured in nanoseconds using the *Chrono* library

*\*Important Notes : The Rabin Karp class implements an arithmetic hashing algorithm and Boyer Moore class uses the good suffix heuristic . Refer to Appendix for the C++ implementations*

# Specification of Algorithms to be used

## Brute Force Matcher using Hamming Distance

This algorithm uses the simple brute force algorithm in addition to calculating the Hamming distance between two strings. The number of positions where the corresponding characters in the two strings differ from one another is known as the Hamming distance. Every potential sub-string of the test document will be iterated through by the

Add Headings (Format > Paragraph styles) and they will appear in your table of contents.

method, and the Hamming distance between each sub-string and every sub-string in the corpus will be determined. Any substrings with a low enough Hamming distance will be flagged by the algorithm as potentially plagiarized. How low can the hamming distance get is set by us and stored in a variable called threshold. In this case, the threshold is 0 so there must be no differences for the document to be not considered plagiarized.

## Rabin Karp Algorithm

The Rabin-Karp algorithm effectively finds a pattern inside a bigger text document by using hashing. The algorithm calculates the hash value of the test document's substrings and the hash value of the pattern taken. The algorithm will compare the sub-string and pattern character by character to determine whether a match exists if the hash values match. To identify probable plagiarism, the algorithm will compare every sub-string of the test document against the pattern or the corpus.

## KMP algorithm

To optimize the string matching process, the KMP algorithm employs the idea of a prefix function. The prefix function is a table that is constructed depending on the pattern, which aids in minimizing redundant comparisons by using previously matched pattern information. The prefix function is computed in linear time, hence the overall time complexity of the technique is O(m + n), where m is the pattern length and n is the text length.

## BoyeerMoore algorithm

The essential principle underlying the Boyer-Moore method is to begin matching the pattern at the end and move backwards towards the beginning. Mismatches near the end of the pattern are more probable to happen than mismatches at the start of the pattern.The bad character rule is used for skipping over characters in the search string that do not match the matching character in the pattern. The method looks for the character's rightmost occurrence in the pattern and changes it to match the rightmost incidence in the search string.Additionaly,The Good Suffix Rule: The good suffix rule is used to handle scenarios where the pattern's suffix matches a substring in the search string. The technique finds the longest feasible suffix that matches a substring in the search string and moves the pattern to match the start of the matching substring.

**Data Specifications**

The input data used in the work is 1 test document and 4 corpus documents. The test document will contain some sentences from the corpus documents so we can test it for plagiarism. Moreover, the test document and corpus documents contain randomized content/essays that were brought from sample papers.

# Experimental Results

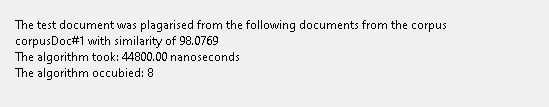
According to the results, Rabin-Karp is the quickest algorithm, lasting just 18,500 nanoseconds. Boyer-Moore comes in second with a time of 23,700 nanoseconds, followed by KMP with a time of 34,700 nanoseconds, and lastly Brute Force with a time of 44,800 nanoseconds.

The underlying computational approaches utilized by each of these algorithms account for the speed disparity. Brute Force compares every symbol of two strings individually and computes the Hamming distance. The time complexity of this strategy is O(n), where n is the number of characters of the string. This is the slowest of the methods and is only useful for short strings.Rabin-Karp compares the two strings using hashing. It generates a hash value for every substring of the target string and compares it to the search string's hash value. This method is substantially quicker than Brute Force since it has a time complexity of O(n), where n is the length of the string. Both KMP and Boyer-Moore are pattern matching algorithms that function by processing the search string to generate a database of potential matches. The time complexity of the KMP technique is O(m + n), where m is the total length of the pattern and n is the length of the search string, which makes it quicker than Brute Force but slower than Rabin-Karp.

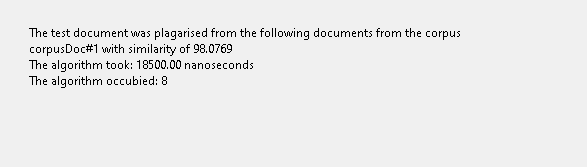
In conclusion, the Rabin-Karp method is executed the fastest because it employs hashing to optimize the comparison process, whereas the other algorithms use preprocessing techniques to optimize pattern matching. Brute Force is the slowest approach and is only useful for short strings.

**Findings while using QT after comparing the first sentence of the test document with all of the corpus:**

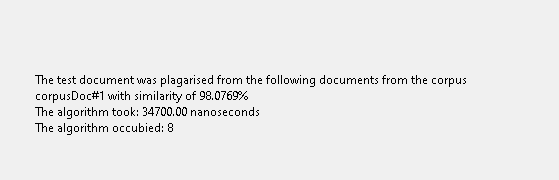
Bruteforce:



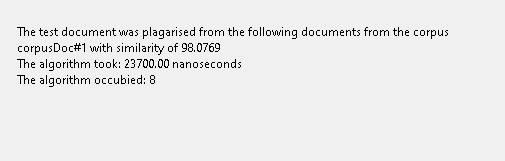
RabinKarp:

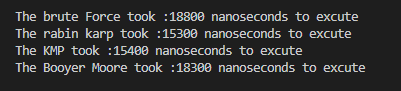


KMP:

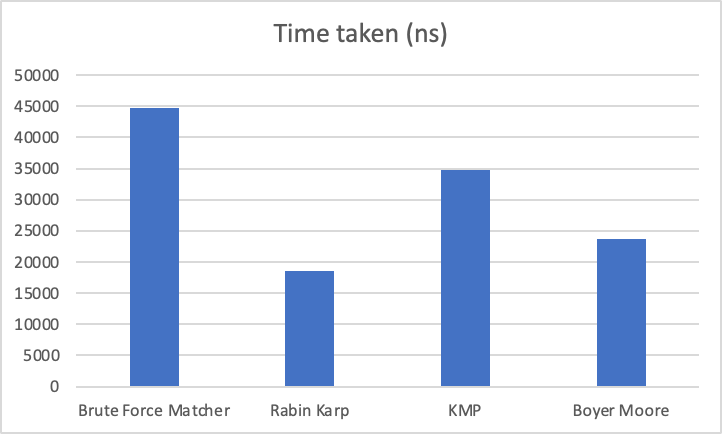


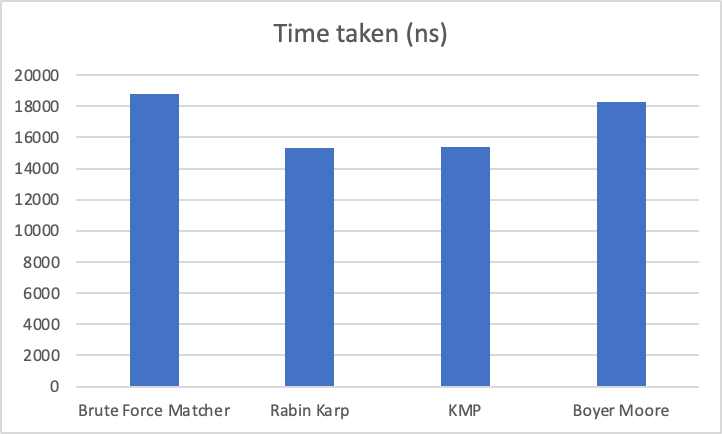
BoyeerMoore:



**Findings while using the main program after comparing the first sentence of the test document with all of the corpus:**

**Graph representation of time complexity of the algorithms while using QT after comparing the first sentence of the test document with all of the corpus**

****

**Graph representation of time complexity of the algorithms while using the main program after comparing the first sentence of the test document with all of the corpus**

# **Analysis and Critique**

After analyzing the results that we included in our findings, it was evident that the Rabin-Karp algorithm was clearly the fastest algorithm for string matching. It is significantly faster than the other algorithms, especially for longer texts, because it uses hashing to compare two strings. The brute-force approach is the slowest algorithm, whereas the Boyer-Moore and KMP are the next two quickest algorithms with the Boyer Moore algorithm taking a slight edge over the KMP algorithm. Moreover, The fact that the brute-force algorithm analyzes each character of two strings separately and has an O(n) time complexity for a string of length n is one of its main limitations. Because it takes so long to process larger strings, this approach is only practical for small strings. As a result, it is not a practical approach for bigger datasets. On the other hand, the Rabin Karp algorithm gives each substring in the text a hash value, which then compares to the hash value of the pattern. This algorithm finds matches quicker than brute force and all of the other algorithms since it requires fewer comparisons thanks to the use of hash values. Nevertheless, a prominent drawback that has to be stated is that hash collisions can occur due to the usage of hashing, which may result in false positives or ,in other words, false matches. Furthermore, KMP and Boyer Moore algorithms use preprocessing techniques to enhance pattern matching. The worst-case time complexity of the Boyer-Moore method is O(mn), whereas the KMP approach has a time complexity of O(m+n), where m is the length of the pattern and n is the length of the text. Moreover, the advantage that gives Boyer-Moore the edge over KMP is that it skips over unnecessary parts of the text using the good suffix table.Overall, while all four algorithms are useful for string matching, Rabin-Karp is the most effective choice, followed by Boyer-Moore and KMP in terms of effectiveness. The brute-force approach is only useful for short strings and is the least effective of the four. Lastly, it is notable to provide critique about our own implementation of the algorithms too and what were the limitations that occured due to this implementation. Moreover, our implementation caused us to fail to calculate the space complexity of each of the algorithms because the classes of the algorithms had implementations of functions only causing all the classes of the algorithms to have the same space complexity of 8 since none of them have member variables and the space complexities of classes depend on the existence of member variables. Moreover, this is an issue that we acknowledge in our code and hope to fix and make it better in the future.

# **Conclusion**

Our implementation of the algorithms included shortcomings such as the inability to determine the space complexity of each method owing to the implementation of the classes. It is critical to recognise these limitations in order to enhance the accuracy of future investigations.

To summarise, picking the most successful method is dependent on the application and dataset. Rabin-Karp is the favoured solution for bigger datasets owing to its speed, although Brute Force may be useful for smaller datasets. Boyer-Moore and KMP are both suitable for medium-sized datasets, with Boyer-Moore being more effective due to its ability to skip through redundant text. When choosing a string matching algorithm for a certain application, one may make an educated selection by analysing the strengths and limits of each approach.

# **Acknowledgements**

- Chat GPT: We used it as a debugging tool and used it to create the random samples of the test document files and corpus files, and to explain the algorithms and provide small scale examples for them to have a better understanding.

-Seif Sallam CS Senior at AUC (suggested the chrono library to measure the time of the algorithms)

-Omar Moawad CE Senior at AUC (suggested the implementation of the createfromfile function in the document class as we faced some difficulties when we used the getline at first. Moreover, the getline function did not get the string that was needed as is and to prove that we experimented with it on a small scale example where we wrote a line in a file and used the getline function to get the line and saved it in a string variable; then, we wrote the same line in the program as a string and compared it using the == with the string variable that stored the getline content and it turned out to not be equal to each other. Therefore, we asked omar for his help and he suggested the iterator approach)

# 

# 

# References

-Cormen, T. H., Leiserson, C. E., Rivest, R. L., &#38; Stein, C. (2009). Introduction to Algorithms, third edition (pp. 985–1013). MIT Press.

-Eshwitha, R. (2022, November 7). *Applications of string matching algorithms*. GeeksforGeeks. <https://www.geeksforgeeks.org/applications-of-string-matching-algorithms/>

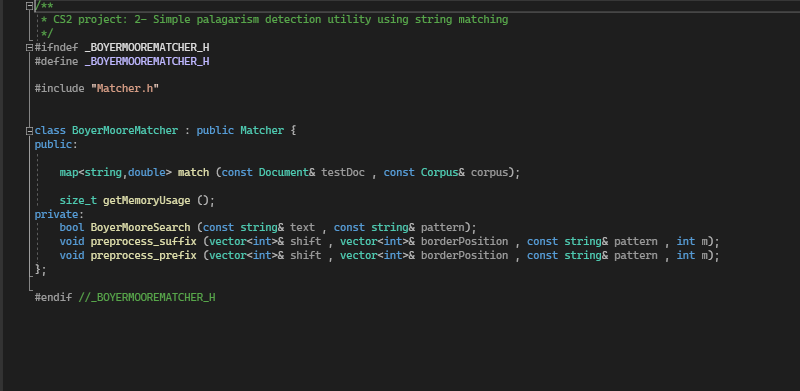
-Gaurav, S. (2022, November 3). *Boyer Moore Algorithm*. Scaler Topics. https://www.scaler.com/topics/boyer-moore-algorithm/

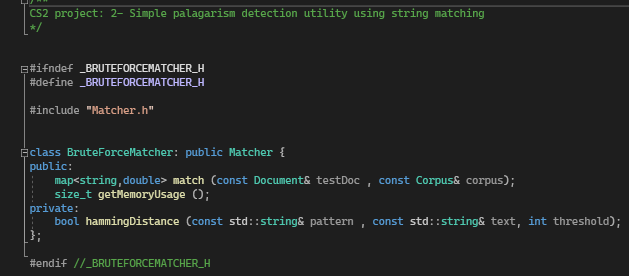
-GeeksforGeeks. (2022, September 23). *Rabin-Karp algorithm for pattern searching*. GeeksforGeeks.https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/

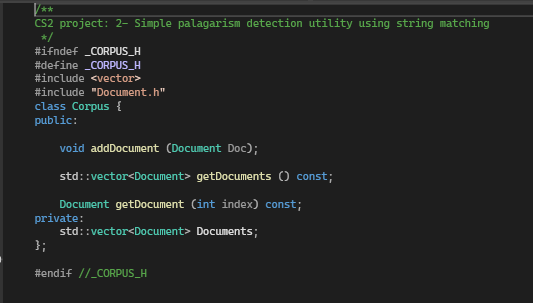
-GeeksforGeeks. (2023, May 4). *KMP algorithm for pattern searching*. GeeksforGeeks. https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

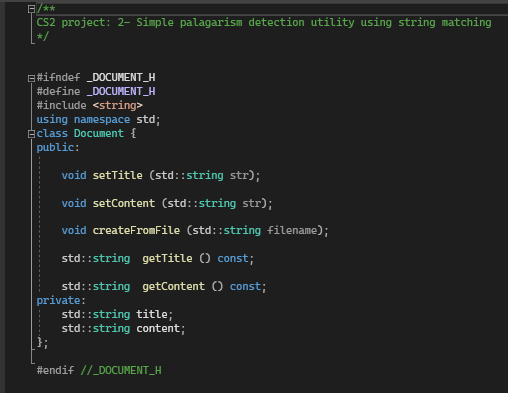
-GeeksforGeeks. (2023, February 24). *Naive algorithm for pattern searching*. GeeksforGeeks. https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/

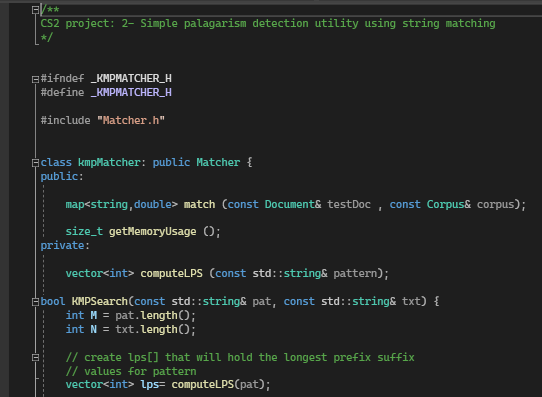
# Appendix A

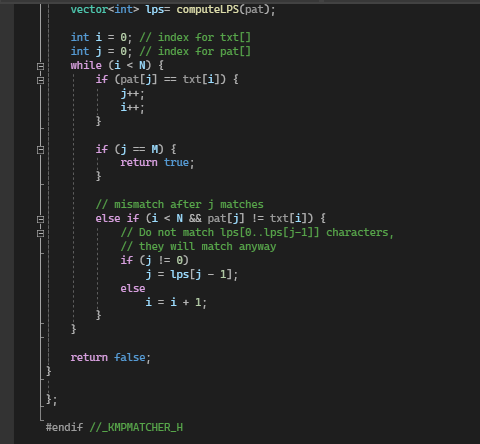
****

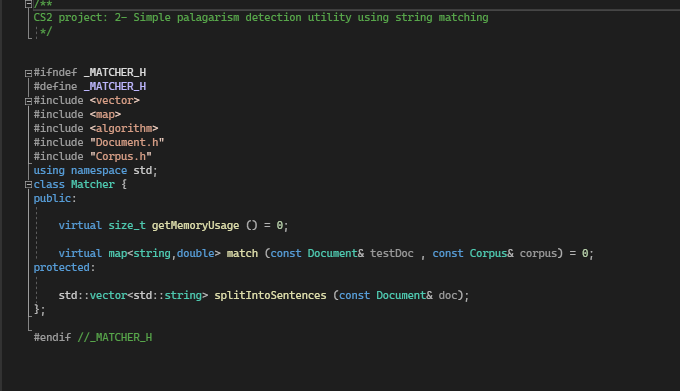
****

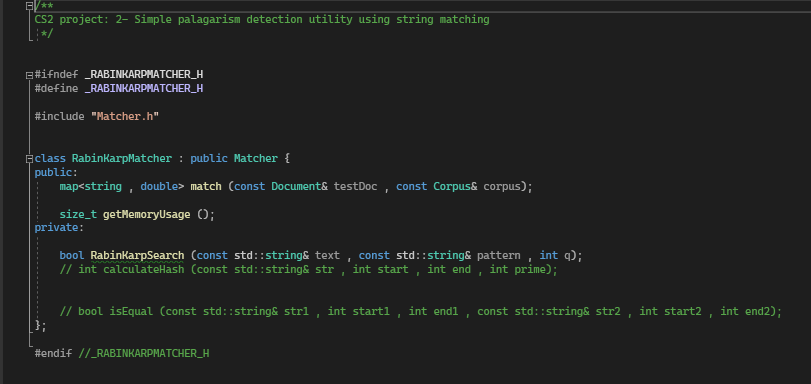
****

****

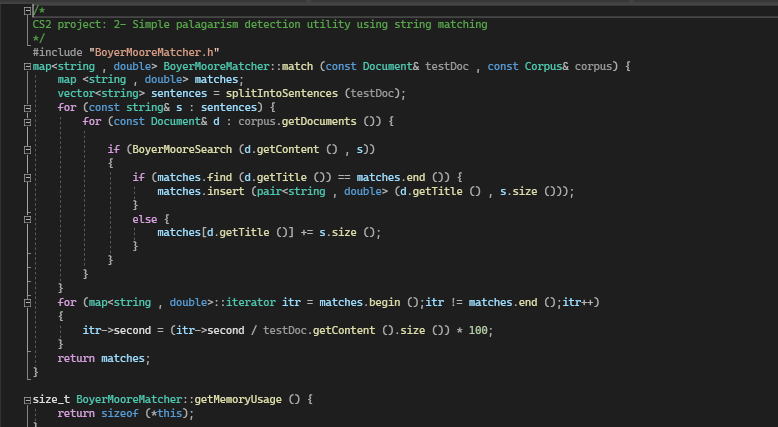
****

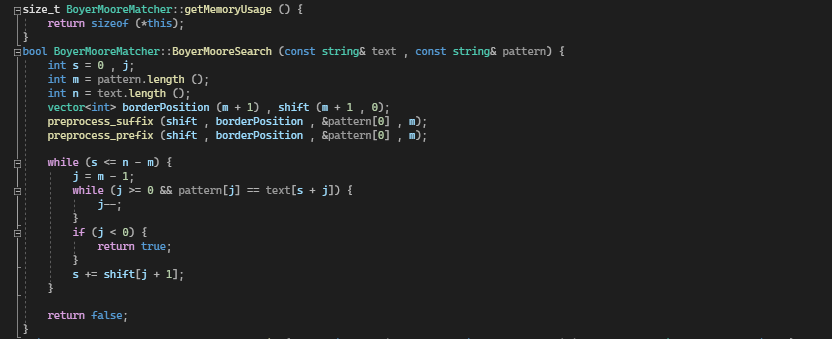
****

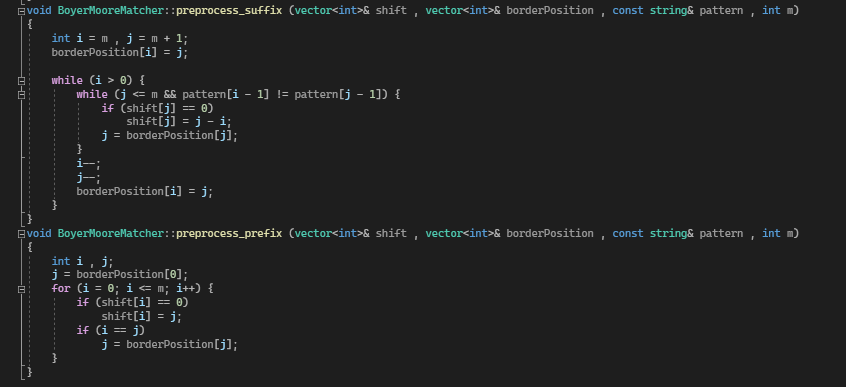
****

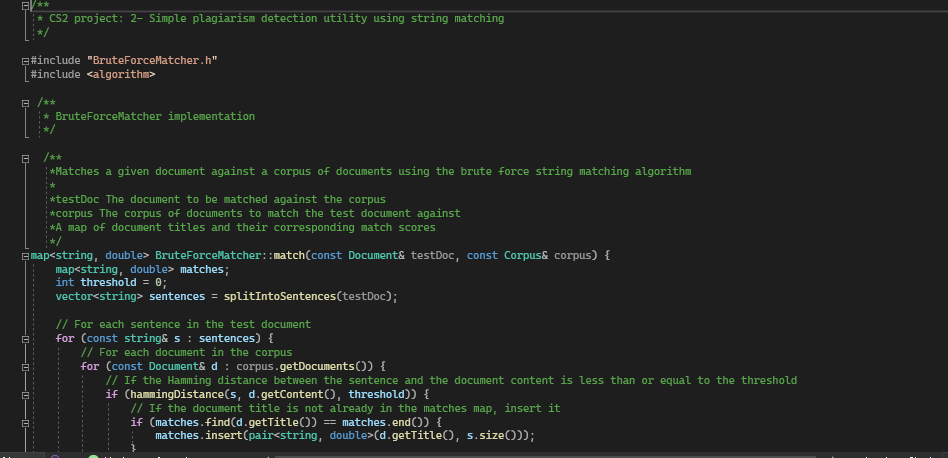
****

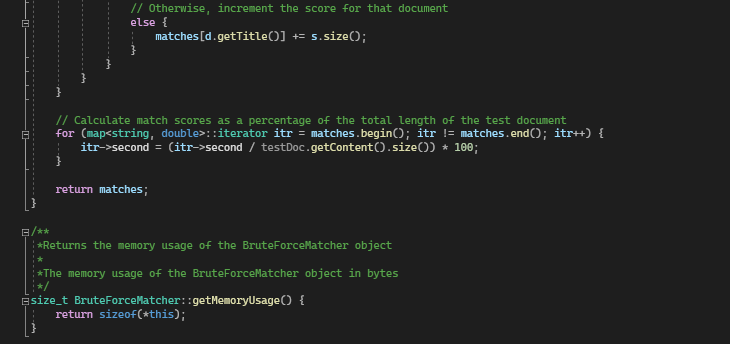
# *Appendix B*

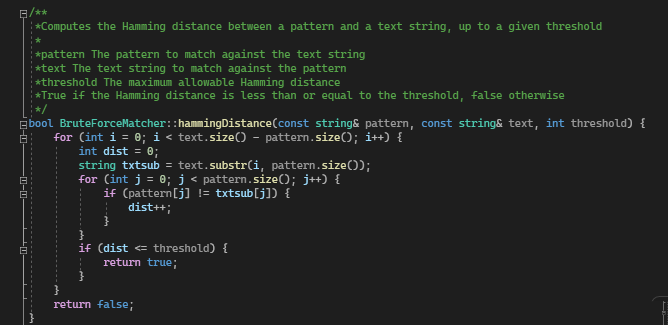
****

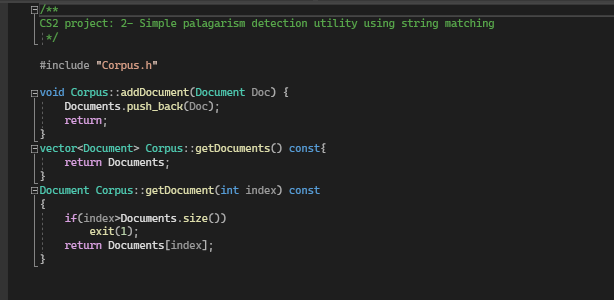
****

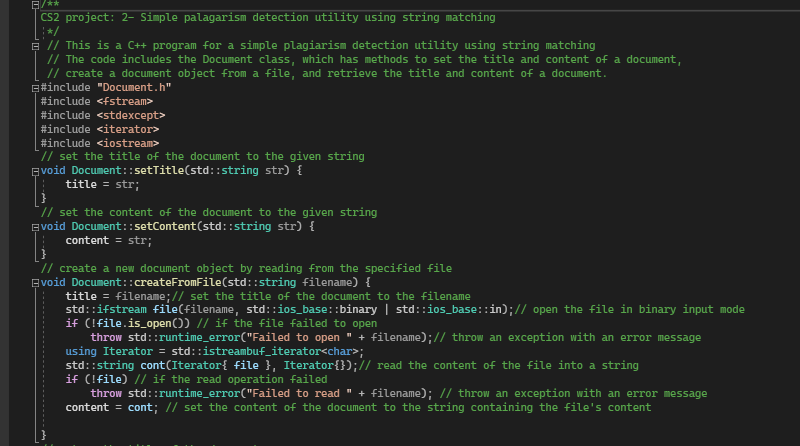
****

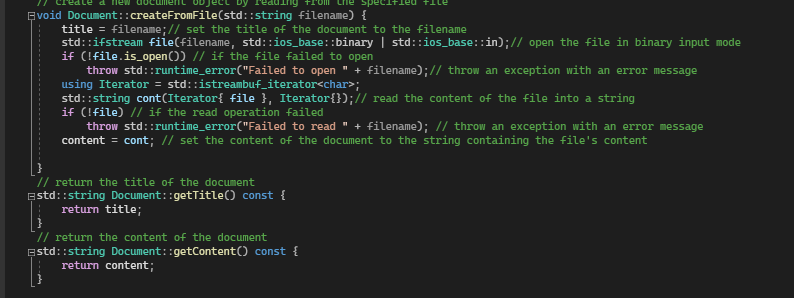
****

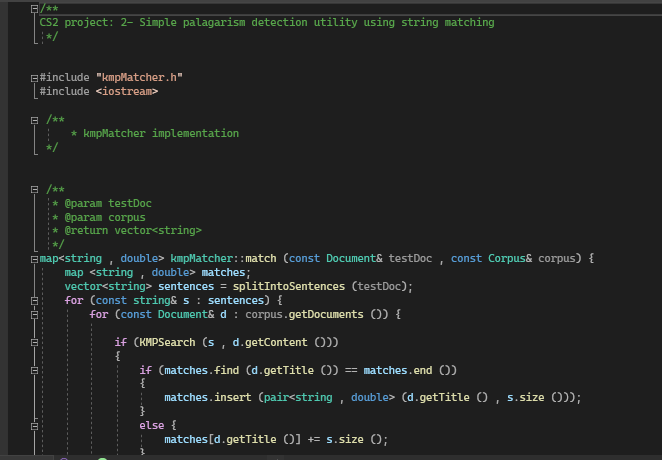
****

****

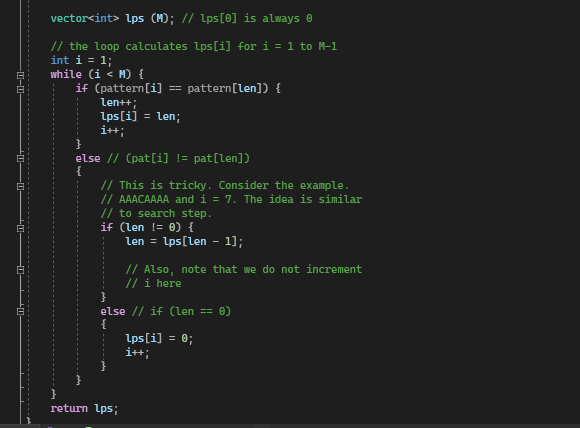
****

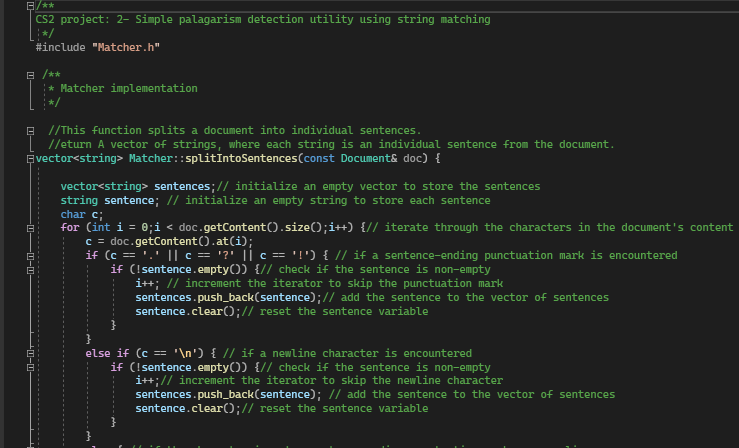
****

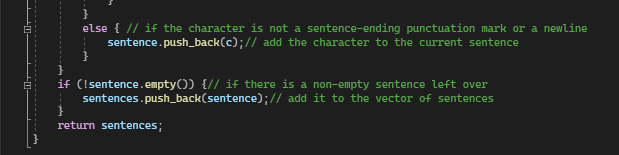
****

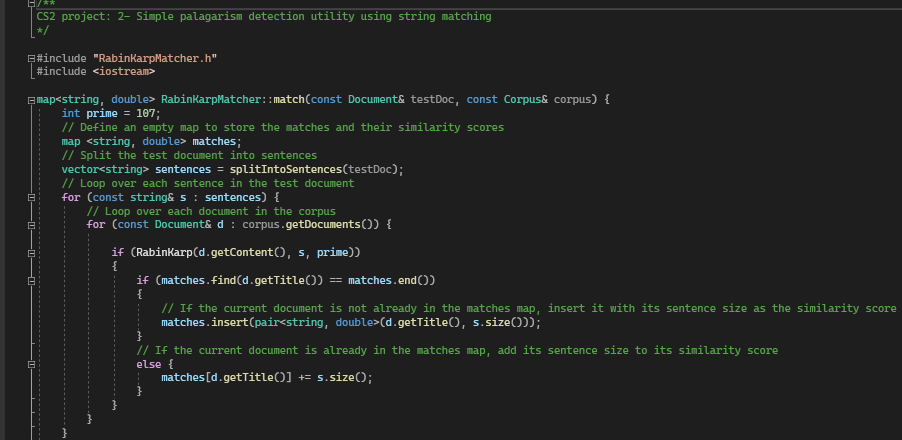
****

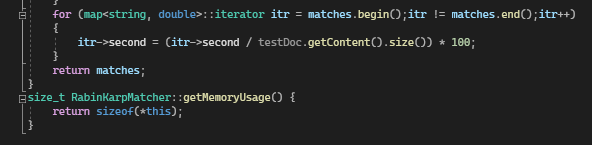
****

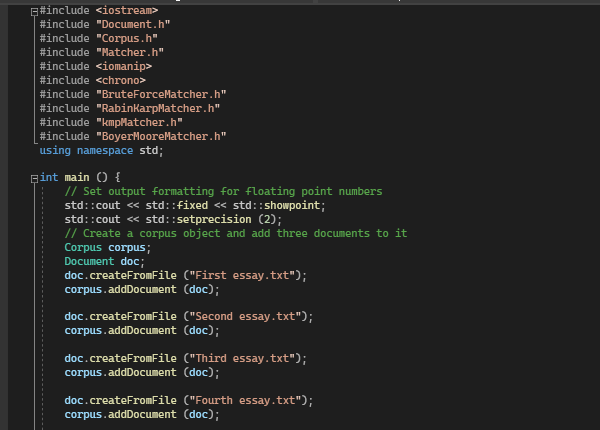
****

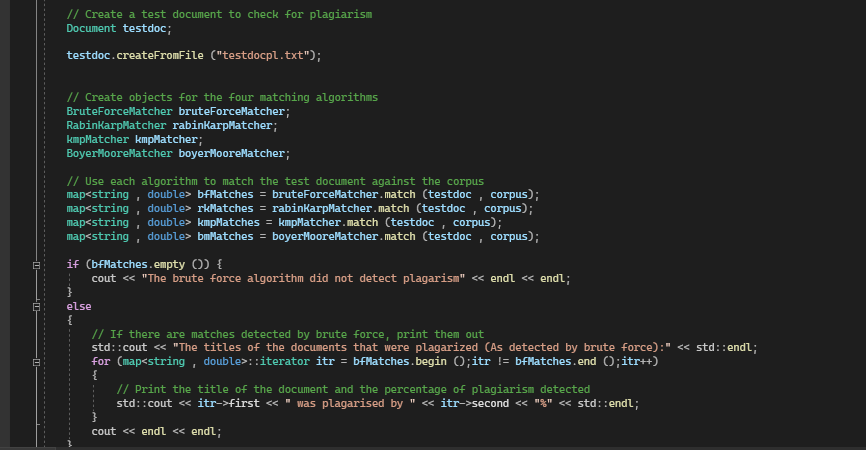
****

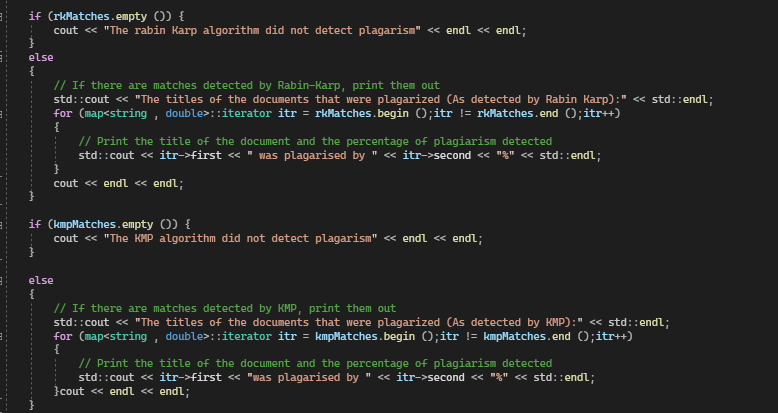
****

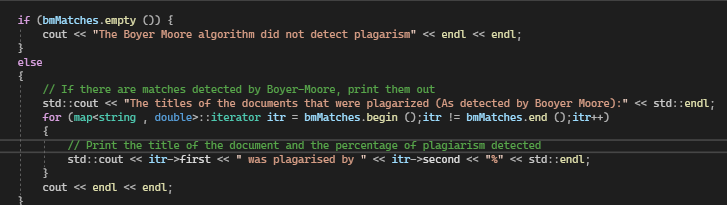
****

****

****

****

****

****

****