Project Report: Simple Plagiarism Detection Utility using String Matching

Ebram Thabet ID:, Adham Ali ID: 900223243, Mahmoud Sharaby ID: 900223616.

Department of Computer Science and Engineering, The American University in Cairo

CSCE 1101-05: Fundamentals of Comp II

Dr.Howaida Ismaeel

Abstract:

This project explores various string-matching algorithms and compares their performance in time complexity and accuracy phrases. The examination focuses on 4 algorithms: brute force, the usage of Hamming distance, Knuth-Morris-Pratt (KMP), Rabin-Karp, and Boyer-Moore. The goal is to determine which rules better suit specific facts units, especially high-plagiarist, mid-plagiarist, and non-plagiarist documents. The research methodology entails implementing these algorithms in a plagiarism detection utility, where each file sentence is treated as a capacity sample to be checked. The comparative evaluation is performed by means of jogging each algorithm at the equivalent document from each statistics institution 3 times and recording their standard execution time in nanoseconds. The experimental consequences suggest the time taken by using every algorithm for one-of-a-kind scenarios, offering insights into their efficiency. The findings contribute to the know-how of the strengths and weaknesses of each set of rules and may guide builders in selecting the maximum suitable algorithm for particular string-matching tasks in C++. Overall, this research enhances the know-how of string matching in C++ and allows efficient and powerful string manipulation and evaluation.

Keywords: Brute Force using hamming distance, KMP algorthim, Boyer-Moore, Rabin Karp, Plagiarism

1. Introduction:

String matching is essential in computer technological know-how and vital in numerous programs—one famous programming language for implementing string-matching algorithms is C++. C++ offers a practical set of equipment and libraries that permit green and effective string manipulation and comparison. We can discover the idea of string matching and its importance, highlighting the numerous strategies and algorithms used in C++ for this reason. String matching involves locating one or different occurrences of a selected pattern inside a given string. This sample may be unmarried, a sequence of characters, or a regular expression. The capacity to successfully carry out string matching is vital in various fields, including textual content processing, statistics mining, records retrieval, bioinformatics, and natural language processing. C++ gives numerous built-in functions and libraries that facilitate string-matching operations. One of the maximum commonly used techniques for string matching in C++ is brute force. In this approach, we sequentially examine each character of the pattern of the textual content. If a mismatch occurs, we move the sample one role to the right and retain the contrast until either a match is observed or the end of the textual content is reached. To enhance the efficiency of string matching, C++ presents superior algorithms. One popular set of rules is the Knuth-Morris-Pratt (KMP) algorithm, which exploits the understanding won from preceding character comparisons to avoid useless re-comparisons by precomputing the longest proper prefix. Another top-notch string-matching algorithm in C++ is the Boyer-Moore algorithm. This algorithm leverages number-one techniques. The awful man or woman rule enables skipping comparisons via shifting the sample to more than one position when a mismatch happens, based totally on the final incidence of the mismatched person within the model. The desirable suffix rule, on the other hand, uses data about the matched suffixes of the pattern to decide the subsequent feasible shift. The Boyer-Moore set of rules is typically famous for superior performance in most realistic instances. Besides those well-known algorithms, C++ also gives different string-matching strategies, including the Rabin-Karp algorithm. These algorithms deal with unique eventualities and often perform better in positive contexts, including sample matching or searching in a large collection of texts. With its extensive tools, libraries, and algorithms, C++ empowers programmers to perform green and practical string-matching duties. Whether using brute force, KMP, Boyer-Moore, or different advanced algorithms, C++ presents a versatile platform for tackling string-matching demanding situations and unlocking the full capability of string manipulation and contrast.

2. Problem definition:

As the world gets more advanced, it becomes crucial to have time efficient processes. At the same time, there is a high demand for programs that detect plagiarism. Therefore, it was important to juxtapose some renowned string matching techniques, knowing which is better. This leads to the main aim of this project, which is exploring various string-matching algorithms and determining which is better in terms of time complexity, and accuracy.

3. Methodology

This paper explores and compares four algorithms that use string matching to find a match for a pattern in a large text. We have used those algorithms to make a plagiarism Detection Utility, which looks for a potential pattern that matches the program's database. Basically, a document with some sentences, and each sentence will be dealt with as a possible pattern to be checked. In addition, we will conduct a comparative analysis of those four algorithms: Brute force using hamming distance, KMP, Rabin-Karp, and Boyer-Moore. This comparison aims to check which of the four has the best time complexity in different data groups. The groups will be a high Plagiarist, a mid-plagiarist, and a non-plagiarist document. By that, we try to identify which of the algorithms is suited for each data set in this research. We will perform with one document from the same group that runs three times, taking their time in nanoseconds. Then, take the average of the three runs, equal to the time for the algorithm result.

4. Specification of Algorithms to be used:

Brute Force Using Hamming Distance Algorithm:

Brute Force is a technique for fixing trouble by trying all possible answers. In computing, it usually involves iterating via every likely mixture of inputs to find the ideal output. The Hamming distance algorithm measures the difference between two strings of equal periods. It calculates the number of positions at which the corresponding symbols are specific. This set of rules may be beneficial in imposing a brute pressure method for positive issues. To put into effect the Hamming distance algorithm in C++, you could first define strings of identical length, then use a loop to evaluate every individual in both strings. For every person, this is extraordinary. You could increment a counter. At the cease of the circle, the counter will represent the Hamming distance between the two strings. Next, to use this set of rules in a brute force technique, you could create a loop that iterates through all feasible mixtures of inputs. For instance, if you are attempting to find a password, you may iterate through all possible character combinations till you locate the ideal password. You may calculate the Hamming distance between the aggregate and the goal password for each aggregate. Once you discover the mixture with a Hamming distance of 0, you have found the precise password. The worst-case complexity of the brute force approach with the Hamming distance is O(n*m), in which n is the length of the text and m is the length of the pattern. This occurs whilst the sample must be compared with every possible substring of the identical period inside the textual content. In the worst case, the set of rules desires to compute the Hamming distance for each comparison, resulting in quadratic time complexity. The average complexity of the brute force approach with the Hamming distance is also O(n*m) in maximum instances. Since the algorithm compares the pattern with each possible text substring, the variety of comparisons stays identical. However, the real-time taken can range depending on the sample's distribution in the textual content and the properties of the entered facts. The best-case complexity of the brute pressure approach with the Hamming distance is O(n - m + 1). This happens while the sample isn't observed within the text, and the set of rules handiest desires to compare the model with every substring. However, it's essential to be aware that brute force is frequently not the most efficient approach for fixing a problem, as large entry areas could take a long time. In a few cases, more sophisticated algorithms can be used to lessen the range of possible solutions that need to be checked.

Rabin Karp Algorithm:

The Rabin-Karp algorithm uses a rolling hash feature to compare the pattern with the substrings of the textual content. To start, we calculate the hash value of the sample and the first substring of the identical length from the text. If the hash values are healthy, we carry out a character-with the aid of-character assessment to affirm the shape. However, if the hash values don't match, we move the window by using one individual and update the hash cost by putting off the first individual and adding the subsequent person in the window. This rolling hash computation allows us to evaluate the following substrings in consistent time. To similarly improve performance, we pick out a hash characteristic that supports rolling hashing and minimizes collisions. The set of rules continues this system until it unearths a matching sample or reaches the quit of the textual content. The worst-case complexity of the Rabin-Karp algorithm is O((n - m + 1) * m), in which n is the length of the textual content and m is the duration of the sample. This occurs while there are many spurious hash collisions, which means exclusive styles have an equal hash cost. In such cases, the set of rules wishes to perform extra character with the aid of-character comparisons to affirm the matches, assuming a probably quadratic time complexity. The expected complexity of the Rabin-Karp algorithm depends on the first-rate of hash characteristic used and the distribution of the styles in the textual content. With an outstanding hash characteristic and random sample occurrences, the average complexity is considered O(n + m). This is because the set of rules commonly fits steady-time hash comparisons and only calls for character-by-person comparisons when there are hash collisions. The satisfactory-case complexity of the Rabin-Karp set of regulations is O(n + m), which happens while the sample is not in the textual content. In this situation, the location of rules plays a complete search however does not discover any matching occurrences, ensuing in a linear time complexity.

KMP Algorithm:

The KMP set of rules is carried out through an inventive approach that avoids needless comparisons by using facts about formerly matched characters. To begin, we pre-system the pattern and construct an auxiliary array, known as the LPS (Longest Proper Prefix, also a Suffix) array, which stores the duration of the longest proper prefix. This is also a suffix for every prefix of the pattern. This LPS array allows us to decide where to renew the quest within the textual content when a mismatch happens. Armed with these records, we iterate through the textual content and pattern, comparing characters and adjusting our roles thus. Whenever a mismatch occurs, we consult the LPS array to find the most critical possible suffix of the pattern. This is also a prefix of the textual content substring examined thus far, permitting us to bypass needless comparisons. By leveraging this approach, the worst-case complexity of the KMP set of rules is O(n + m), in which n is the duration of the text and m is the period of the pattern. This occurs when the sample is discovered outside the textual content or when the example only occurs at the cease of the text. In such cases, the algorithm desires to compare all characters of each text and the pattern, resulting in a linear time complexity. The average complexity of the KMP set of rules is also O(n + m) in maximum instances. This is because the rules usually call for a complete contrast of the sample with the text. However, the KMP rules reveal higher overall performance than naive string-matching algorithms in eventualities wherein repeated characters are in the pattern or textual content. The first rate-case complexity of the KMP set of rules is O(n), while the sample isn't observed in the textual content. In this example, the algorithm only wishes to iterate via the text as soon as without acting on any different comparisons.

Boyer-Moore Algorithm:

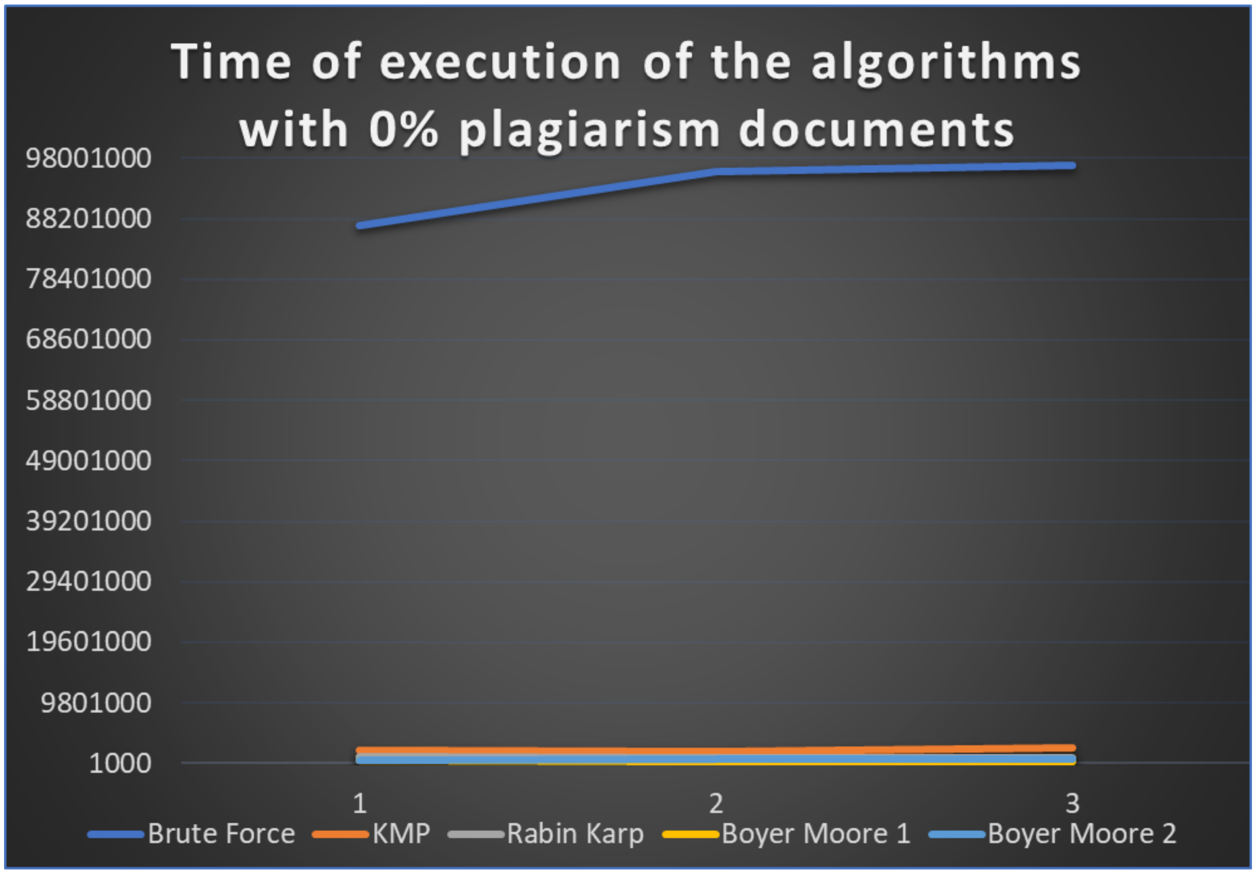
The Boyer-Moore algorithm evolved with a preprocessing phase to create vital research tables. The "bad character" desk, represented using an array of length 256, stores the rightmost occurrence of each character in the pattern. It facilitates determining the maximum possible shift in case of a mismatch. The "right suffix" desk, additionally an array, stores facts about the longest suffix of the pattern that fits a prefix. Once the tables are built, the algorithm moves to the looking segment. It begins by aligning the pattern's last character with the corresponding function in the textual content. Then, it scans the textual content from right to left, comparing characters with the sample. In a mismatch, the set of rules consults the bad character table to discover the most shift. If the mismatched character exists inside the pattern, the shift is calculated primarily based on the difference between the modern index and the rightmost occurrence in the horrific character desk. If the mismatched character no longer exists, the shift is about to the period of the pattern. Additionally, the set of rules considers the excellent suffix table to decide any additional shifts. It calculates the longest matching suffix of the sample with a prefix and uses this fact to shift the sample for this reason. The looking section maintains till a healthy is located, or the pattern is not inside the ultimate textual content. The algorithm then returns the location(s) wherein the way occurs in the textual content. The worst-case complexity of the Boyer-Moore set of rules is O(n * m), where n is the length of the textual content and m is the duration of the pattern. This occurs when the pattern has no repeated characters and no longer appears within the text until the last individual. In such instances, the algorithm may also want to compare all characters of each textual content and the pattern for every ability suit, resulting in quadratic time complexity. The expected complexity of the Boyer-Moore algorithm is commonly considered sublinear or close to linear, making it more excellent and efficient than the worst-case complexity shows. In practice, the Boyer-Moore algorithm regularly shows first-rate common-case overall performance because it uses heuristics that permit skipping several characters at a time for the duration of the matching system. The nice-case complexity of the Boyer-Moore algorithm is O(n/m), which occurs when the sample is discovered at the beginning of the text. In this situation, the algorithm handiest desires to conduct an unmarried comparison among the primary character of the pattern and the corresponding person within the text. The Boyer-Moore algorithm's performance lies in its potential to bypass comparisons by making shrewd shifts based totally on the terrible character and precise suffix rules, making it reasonably powerful in exercises for string-looking tasks.

5. Experimental results:

1. When entering a zero-plagiarized document with fixed size.

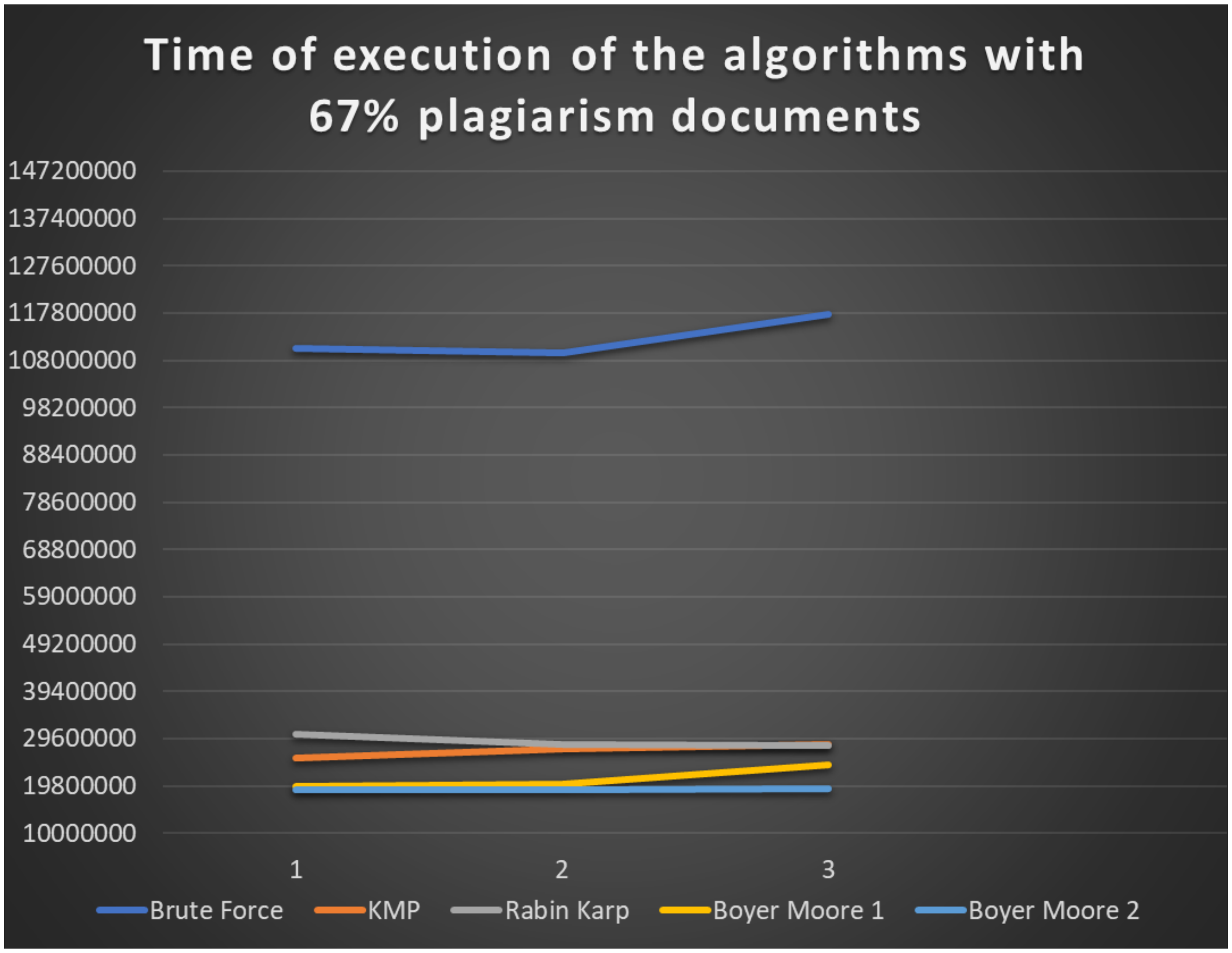
Note: time is measured in nanoseconds

Trial number	Brute Force	KMP	Rabin Karp	Boyer Moore 1	Boyer Moore 2
1	87112800	2174500	1041000	399000	534900
2	95947400	1985700	897100	335300	697800
3	96879900	2575000	907500	251200	629800



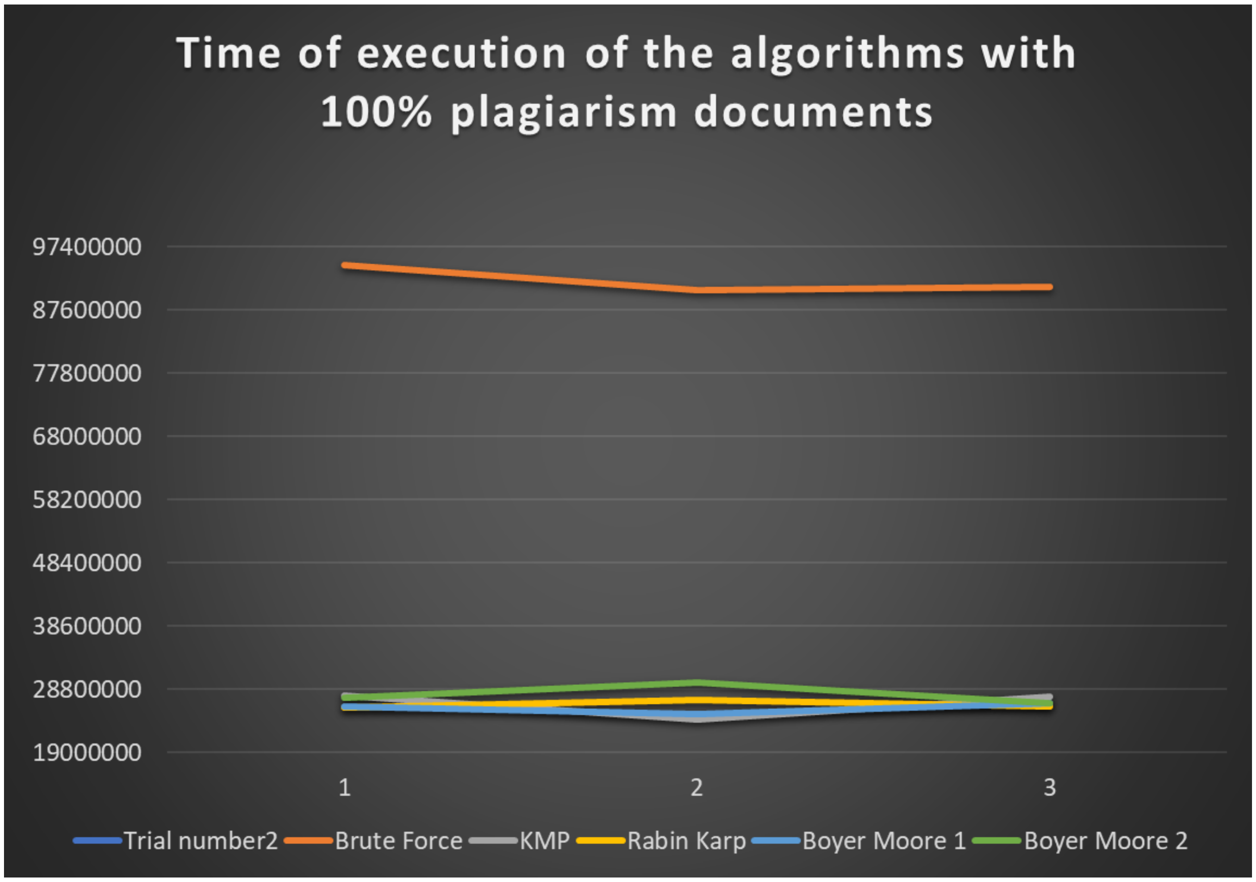
2. When entering a semi-plagiarized document with fixed size as the zero-plagiarized.

Trial number	Brute Force	KMP	Rabin Karp	Boyer Moore 1	Boyer Moore 2
1	110542300	25608500	30547800	19787700	19093500
2	109651800	27561500	28351000	20124400	19044800
3	117458800	28449200	28154500	24129900	19193800



3. When entering a fully plagiarized document with fixed size as the previous ones.

Trial number	Brute Force	KMP	Rabin Karp	Boyer Moore 1	Boyer Moore 2
1	94536700	27802500	25980100	26083200	27439200
2	90612800	24020300	27209200	25023600	29776300
3	91192900	27673500	26120600	26683300	26665300



Now, we will take the totally plagiarized document and change the size of the patterns document.

1. Size from 3 to 5 sentences.

Trial number	Brute Force	KMP	Rabin Karp	Boyer Moore 1	Boyer Moore 2
1	30373400	11344600	7662500	12281700	12278800
2	30270200	11280100	8340300	11238100	11448900
3	29415300	13112100	9034500	11628800	10953700

2. Size from 6 to 8 sentences.

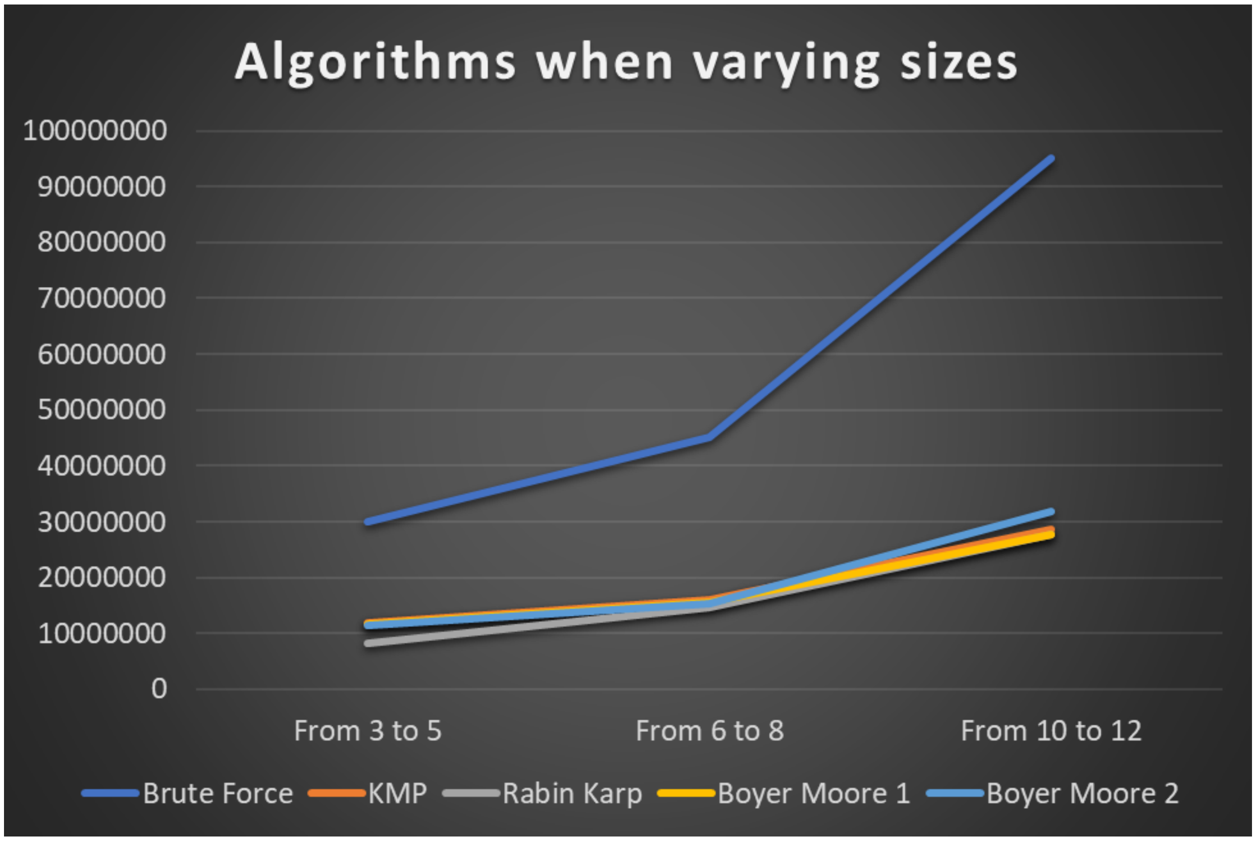
Trial number	Brute Force	KMP	Rabin Karp	Boyer Moore 1	Boyer Moore 2
1	44614000	15110100	12626800	15763000	15618400
2	45487700	17942900	15940000	16529900	14974700
3	45363300	15032900	15139300	14588400	15635000

3. Size from 10 to 12 sentences.

Trial number	Brute Force	KMP	Rabin Karp	Boyer Moore 1	Boyer Moore 2
1	94709400	29002900	26240700	28079600	27822700
2	95275100	28498900	27942100	27388500	26914800
3	95434000	28718500	28828300	27475500	40704000

Table of averages:

Size	Brute Force	KMP	Rabin Karp	Boyer Moore 1	Boyer Moore 2
3-5	30019633.33	11912266.67	8345766.667	11716200	11560466.67
6-8	45155000	16028633.33	14568700	15627100	15409366.67
10-12	95139500	28740100	27670366.67	27647866.67	31813833.33



6. Analysis:

1- The brute force algorithm has the least performance in all cases.Therefore, it is not preferable.

2- When trying documents with a zero-plagiarized document, the other algorithms have almost the same execution time. Slightly different from the other algorithms, by trying a document with 67%-plagiarized document, Boyer Moore algorithms had the best performance, followed by Rabin Karp and KMP. The case is not significantly different in the totally plagiarized document; excluding the brute force algorithm, all the others showed the same results with a minor advantage for KMP.

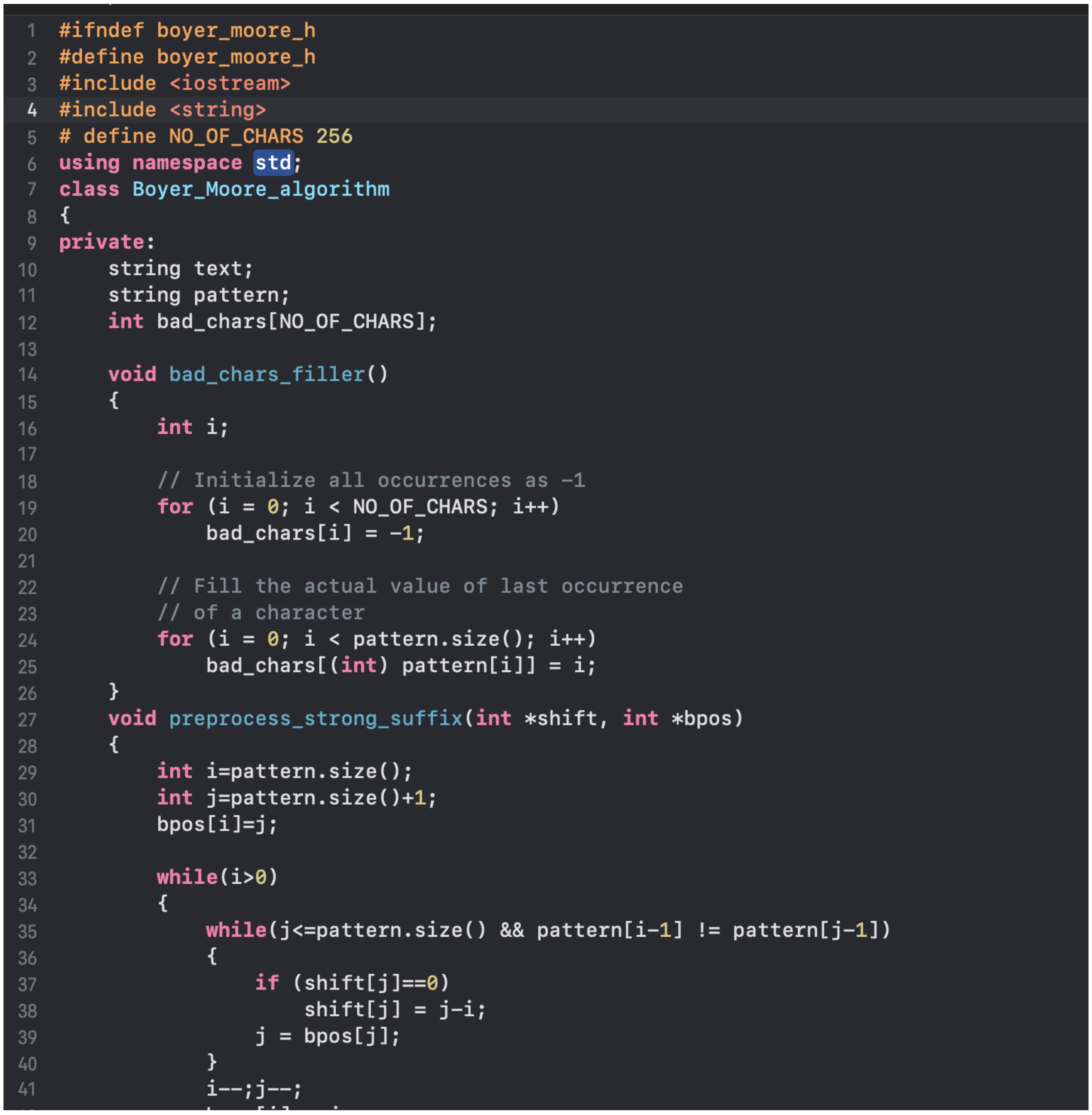
3- In the case of varying sizes, KMP takes the lead in small documents, joined by Boyer Moore's first algorithm in large documents.

7. Conclusion:

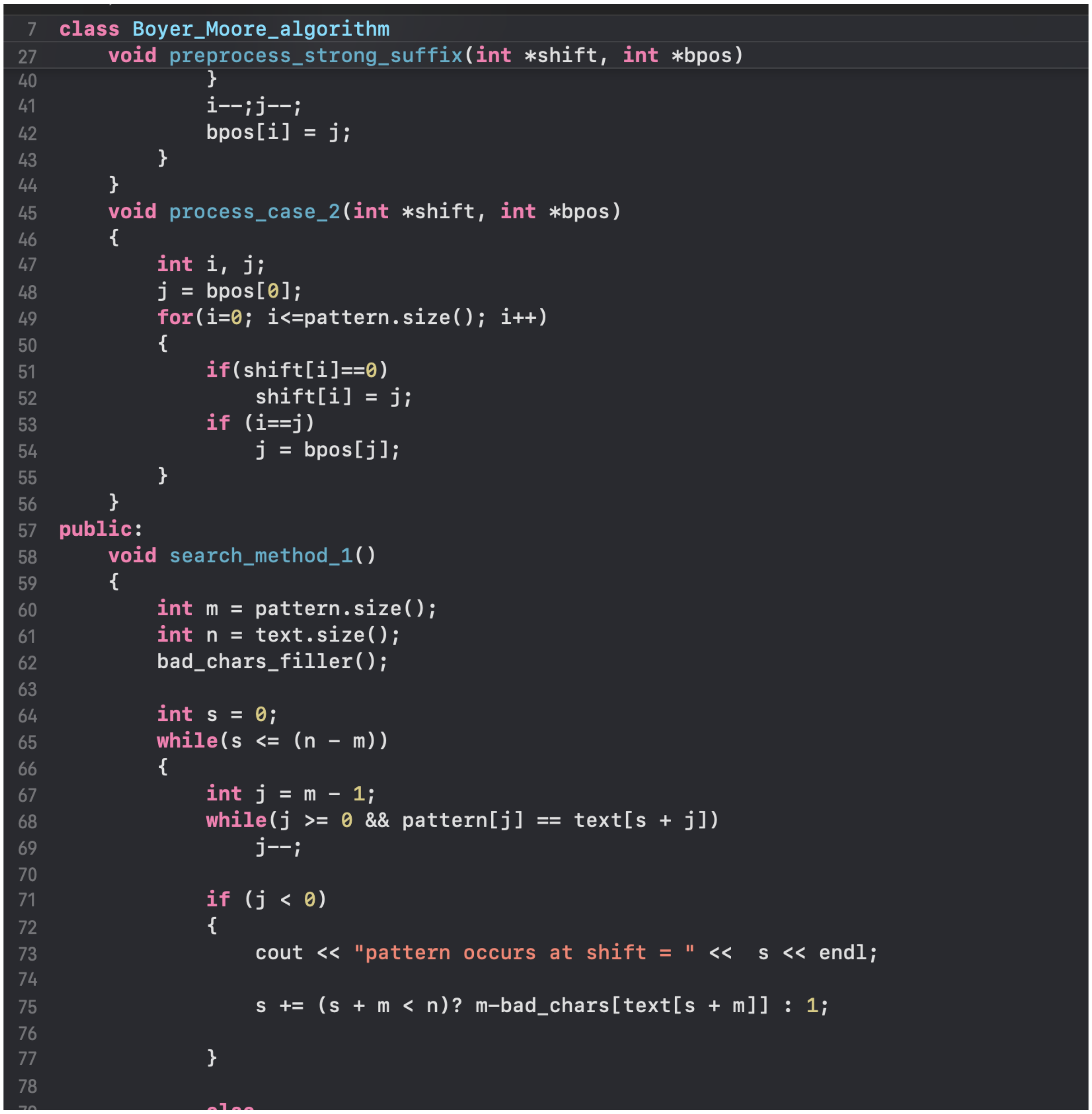
String matching is an important topic in the contemporary world. It has many applications and usages especially in plagiarism detection, which ensures integrity and credibility of written papers. When analyzed, all algorithms but brute force showed nearly close performances. Brute Force is the worst in terms of time among all algorithms. This is owing to the fact that it tries all the possibilities, working naively. This means when attempting to implement a string matching algorithm, it is advisable to avoid using brute force. In the future, it would be great if other powerful algorithms were implemented with better time complexities.

Appendix:

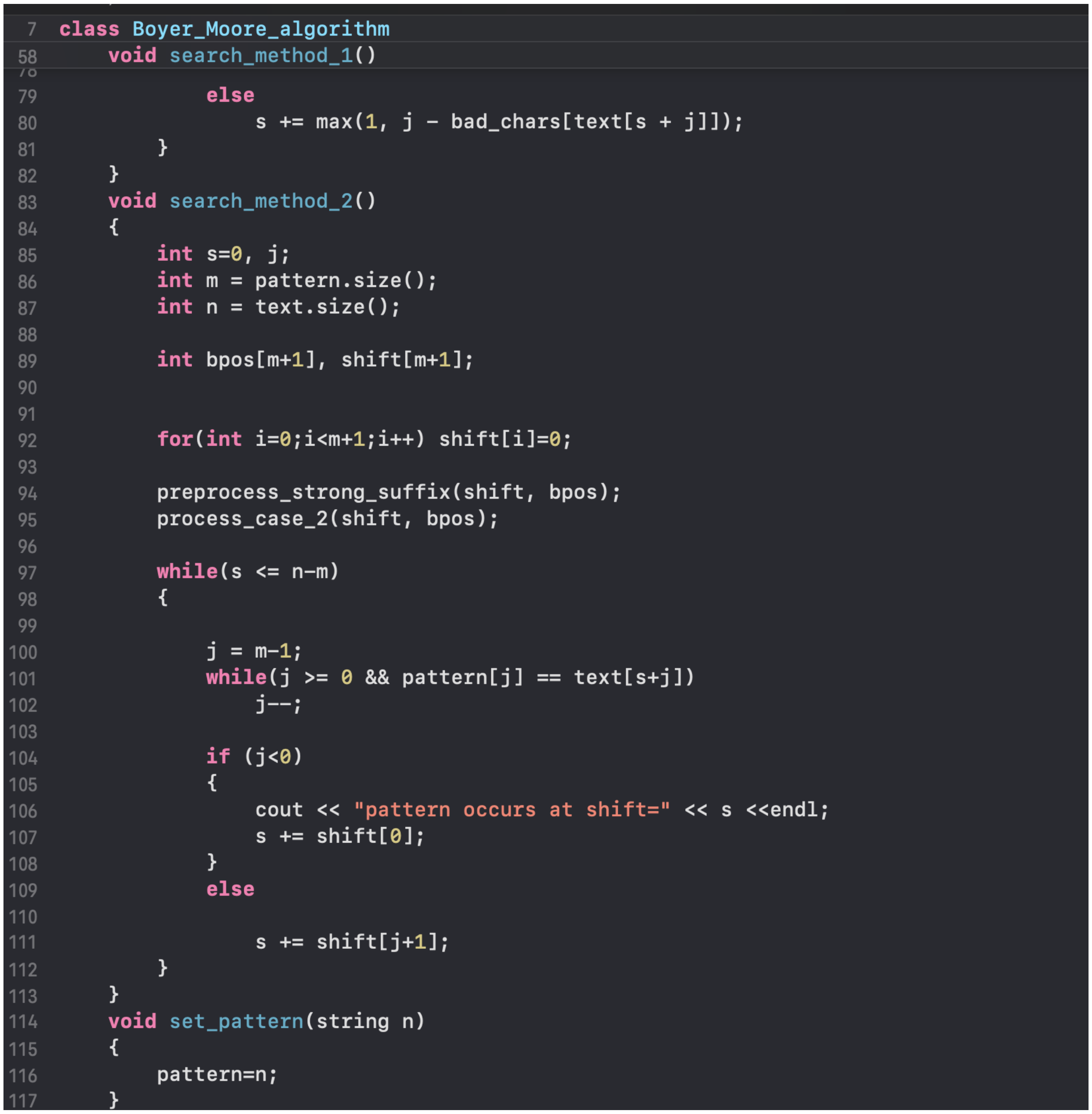
Boyer-moore Algorithm:



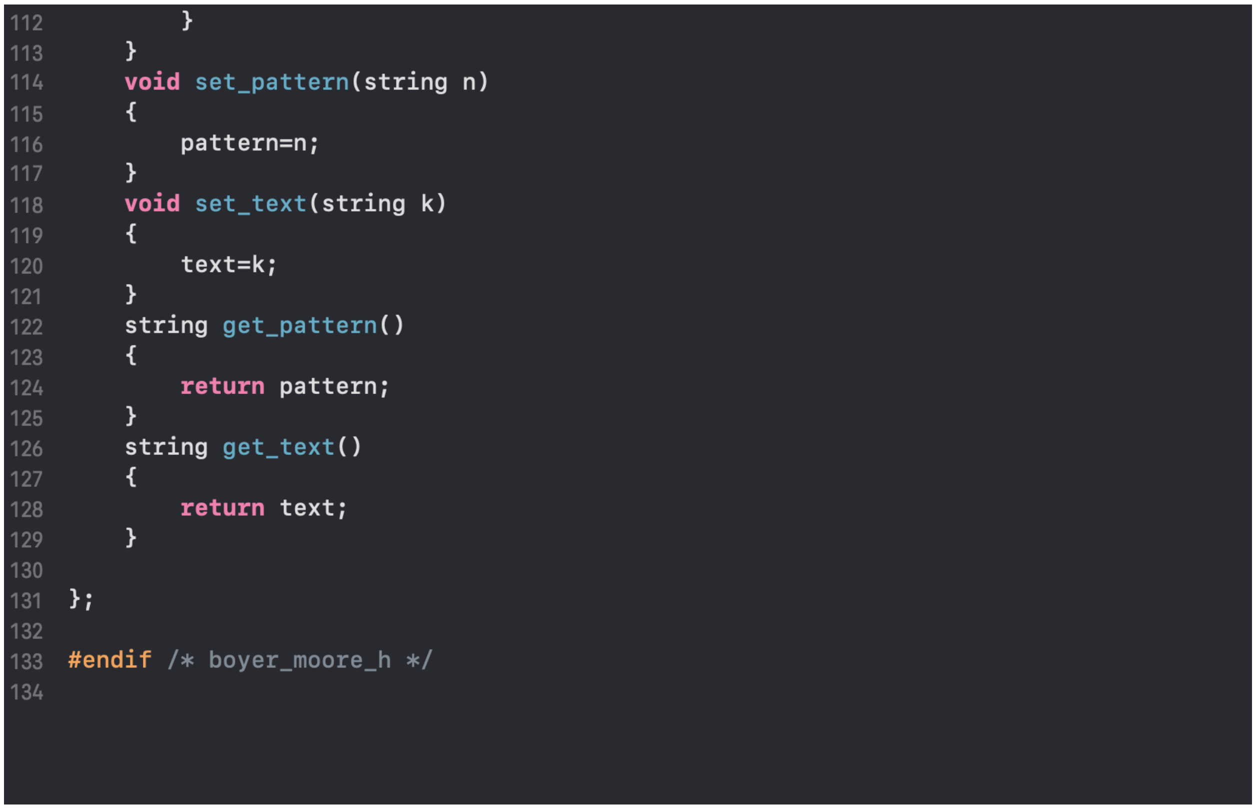
```
#ifndef boyer_moore_h
#define boyer_moore_h
#include <iostream>
#include <string>
# define NO_OF_CHARS 256
using namespace std;
class Boyer_Moore_algorithm
{
private:
    string text;
    string pattern;
    int bad_chars[NO_OF_CHARS];

    void bad_chars_filler()
    {
        int i;

        // Initialize all occurrences as -1
        for (i = 0; i < NO_OF_CHARS; i++)
            bad_chars[i] = -1;

        // Fill the actual value of last occurrence
        // of a character
        for (i = 0; i < pattern.size(); i++)
            bad_chars[(int) pattern[i]] = i;
    }
    void preprocess_strong_suffix(int *shift, int *bpos)
    {
        int i=pattern.size();
        int j=pattern.size()+1;
        bpos[i]=j;

        while(i>0)
        {
            while(j<=pattern.size() && pattern[i-1] != pattern[j-1])
            {
                if (shift[j]==0)
                    shift[j] = j-i;
                j = bpos[j];
            }
            i--;j--;
```



```
class Boyer_Moore_algorithm
    void preprocess_strong_suffix(int *shift, int *bpos)
            }
            i--;j--;
            bpos[i] = j;
        }
    }
    void process_case_2(int *shift, int *bpos)
    {
        int i, j;
        j = bpos[0];
        for(i=0; i<=pattern.size(); i++)
        {
            if(shift[i]==0)
                shift[i] = j;
            if (i==j)
                j = bpos[j];
        }
    }
public:
    void search_method_1()
    {
        int m = pattern.size();
        int n = text.size();
        bad_chars_filler();

        int s = 0;
        while(s <= (n - m))
        {
            int j = m - 1;
            while(j >= 0 && pattern[j] == text[s + j])
                j--;

            if (j < 0)
            {
                cout << "pattern occurs at shift = " <<  s << endl;

                s += (s + m < n)? m-bad_chars[text[s + m]] : 1;

            }

            else
```

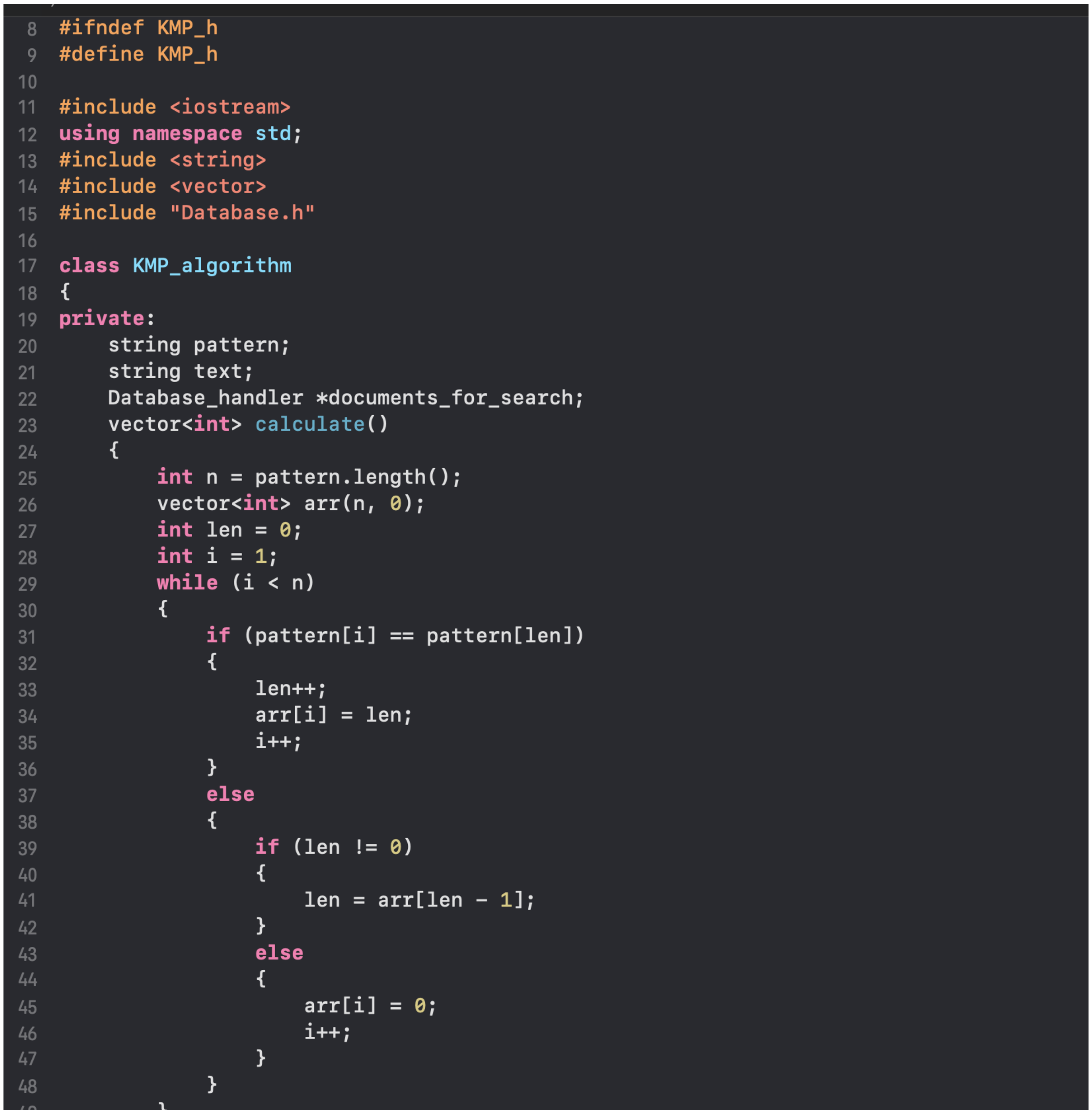


```
class Boyer_Moore_algorithm
    void search_method_1()
            else
                s += max(1, j - bad_chars[text[s + j]]);
        }
    }
    void search_method_2()
    {
        int s=0, j;
        int m = pattern.size();
        int n = text.size();

        int bpos[m+1], shift[m+1];


        for(int i=0;i<m+1;i++) shift[i]=0;

        preprocess_strong_suffix(shift, bpos);
        process_case_2(shift, bpos);

        while(s <= n-m)
        {

            j = m-1;
            while(j >= 0 && pattern[j] == text[s+j])
                j--;

            if (j<0)
            {
                cout << "pattern occurs at shift=" << s <<endl;
                s += shift[0];
            }
            else

                s += shift[j+1];
        }
    }
    void set_pattern(string n)
    {
        pattern=n;
    }
```

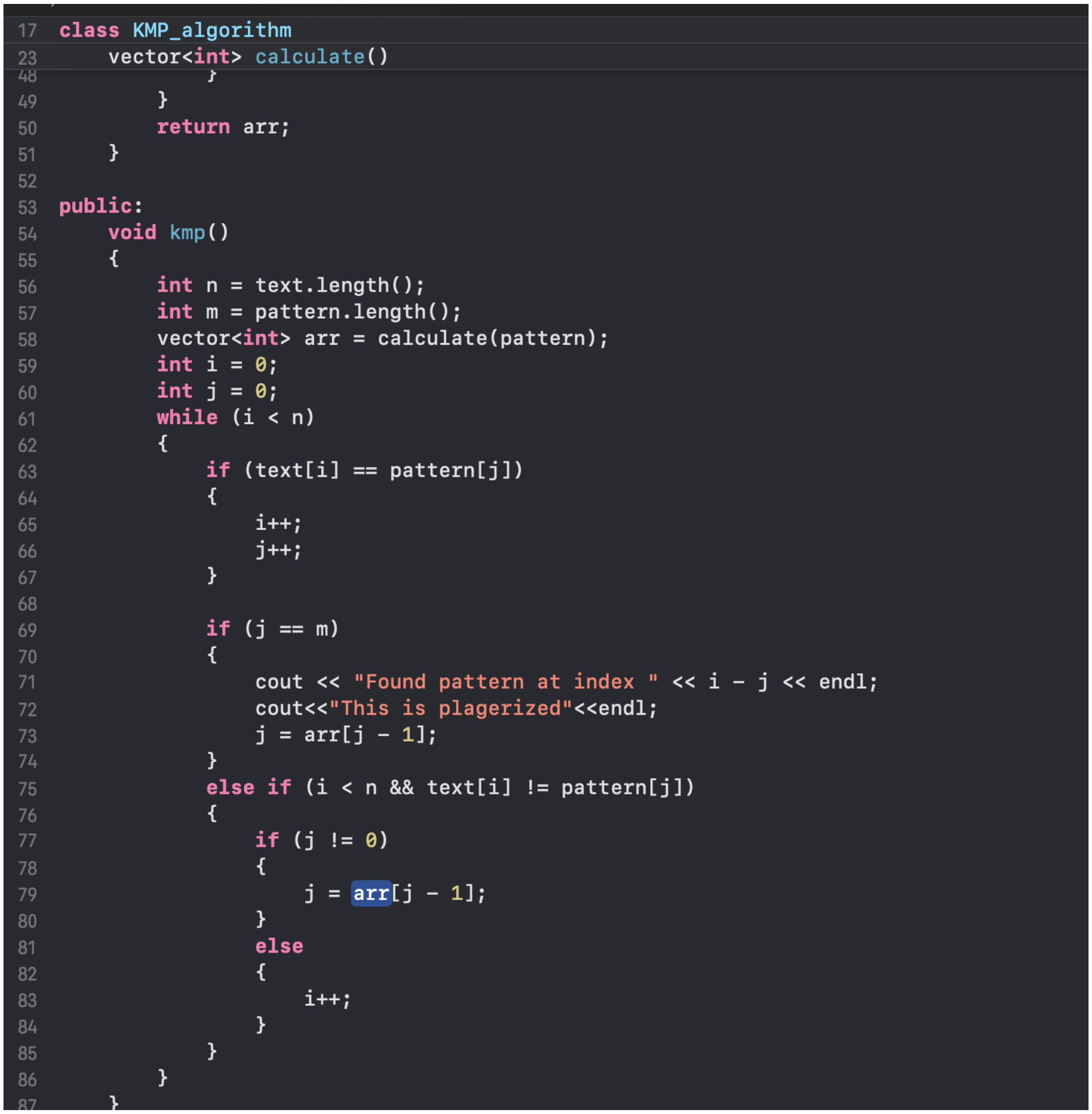


```
        }
    }
    void set_pattern(string n)
    {
        pattern=n;
    }
    void set_text(string k)
    {
        text=k;
    }
    string get_pattern()
    {
        return pattern;
    }
    string get_text()
    {
        return text;
    }

};

#endif /* boyer_moore_h */
```

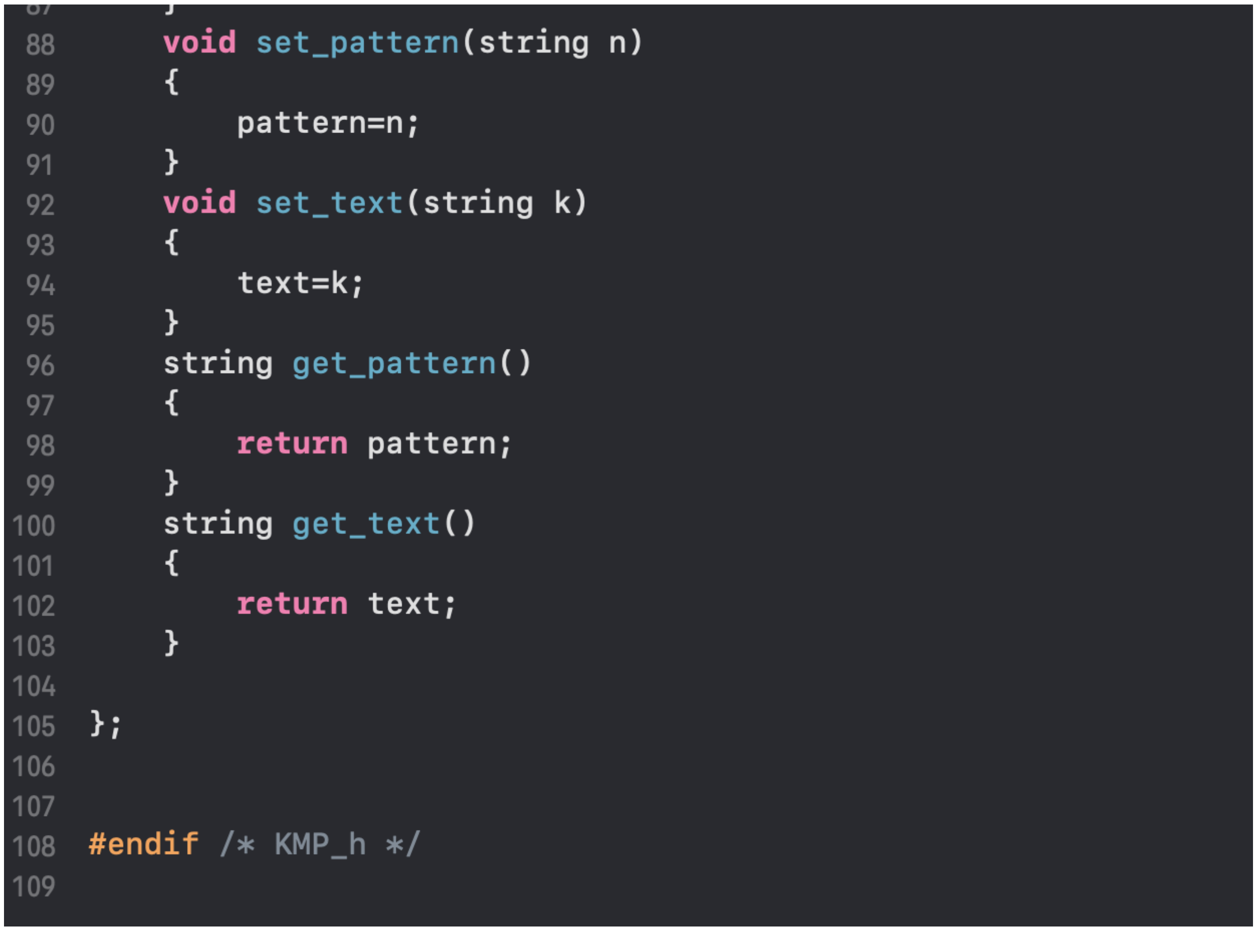
KMP Algorithm:



```cpp
#ifndef KMP_h
#define KMP_h

#include <iostream>
using namespace std;
#include <string>
#include <vector>
#include "Database.h"

class KMP_algorithm
{
private:
    string pattern;
    string text;
    Database_handler *documents_for_search;
    vector<int> calculate()
    {
        int n = pattern.length();
        vector<int> arr(n, 0);
        int len = 0;
        int i = 1;
        while (i < n)
        {
            if (pattern[i] == pattern[len])
            {
                len++;
                arr[i] = len;
                i++;
            }
            else
            {
                if (len != 0)
                {
                    len = arr[len - 1];
                }
                else
                {
                    arr[i] = 0;
                    i++;
                }
            }
```

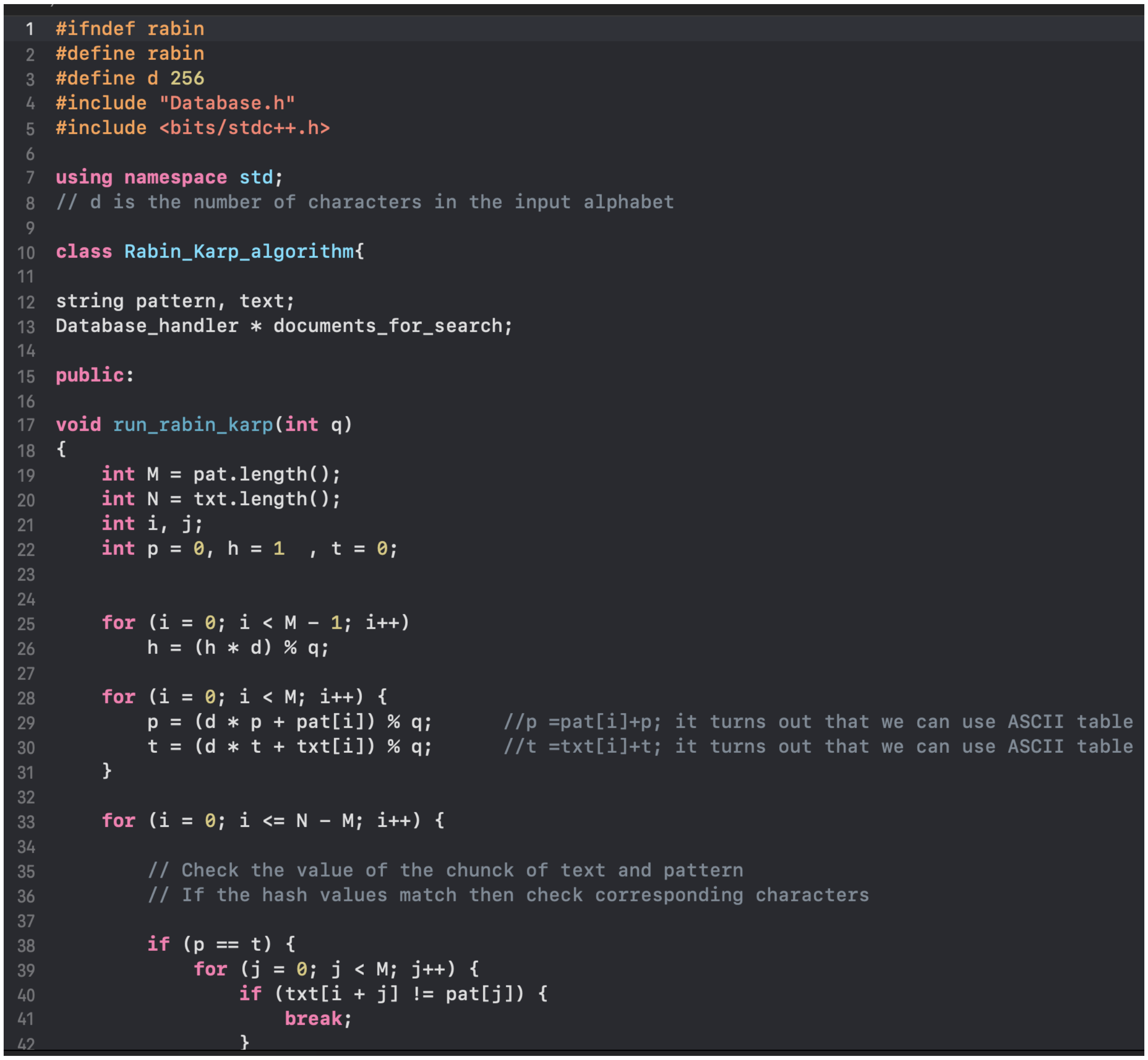


```cpp
class KMP_algorithm
    vector<int> calculate()
            }
        }
        return arr;
    }

public:
    void kmp()
    {
        int n = text.length();
        int m = pattern.length();
        vector<int> arr = calculate(pattern);
        int i = 0;
        int j = 0;
        while (i < n)
        {
            if (text[i] == pattern[j])
            {
                i++;
                j++;
            }

            if (j == m)
            {
                cout << "Found pattern at index " << i - j << endl;
                cout<<"This is plagerized"<<endl;
                j = arr[j - 1];
            }
            else if (i < n && text[i] != pattern[j])
            {
                if (j != 0)
                {
                    j = arr[j - 1];
                }
                else
                {
                    i++;
                }
            }
        }
    }
```

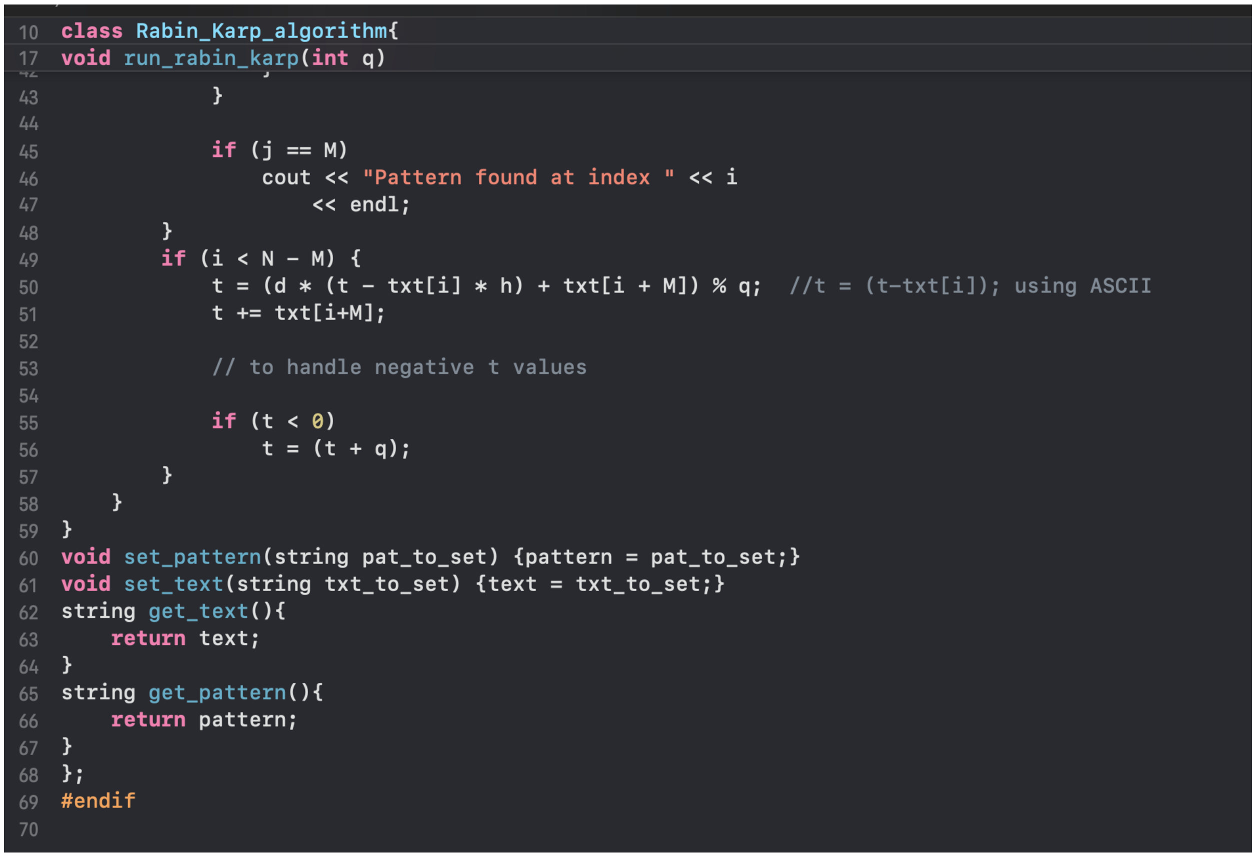


```
    }
    void set_pattern(string n)
    {
        pattern=n;
    }
    void set_text(string k)
    {
        text=k;
    }
    string get_pattern()
    {
        return pattern;
    }
    string get_text()
    {
        return text;
    }

};


#endif /* KMP_h */

```

Rabin Karp Algorithm:

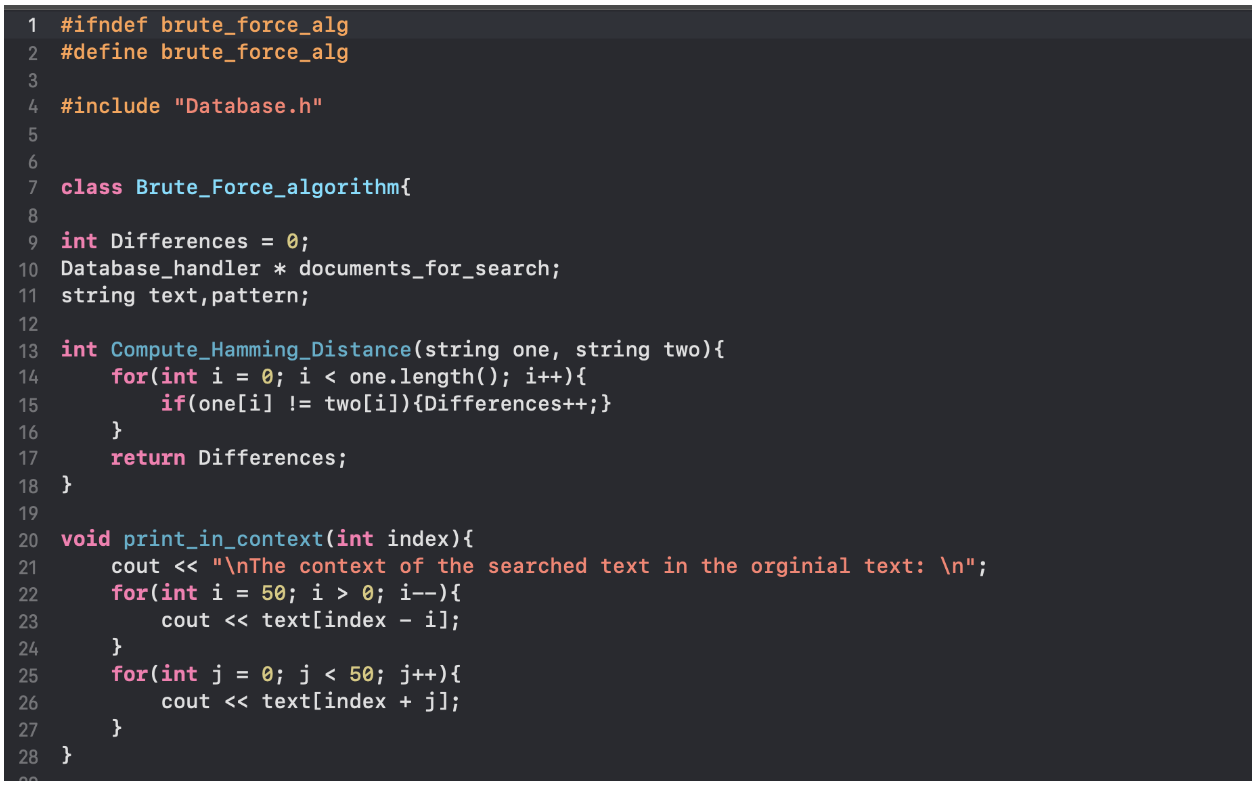


```
#ifndef rabin
#define rabin
#define d 256
#include "Database.h"
#include <bits/stdc++.h>

using namespace std;
// d is the number of characters in the input alphabet

class Rabin_Karp_algorithm{

string pattern, text;
Database_handler * documents_for_search;

public:

void run_rabin_karp(int q)
{
    int M = pat.length();
    int N = txt.length();
    int i, j;
    int p = 0, h = 1  , t = 0;


    for (i = 0; i < M - 1; i++)
        h = (h * d) % q;

    for (i = 0; i < M; i++) {
        p = (d * p + pat[i]) % q;      //p =pat[i]+p; it turns out that we can use ASCII table
        t = (d * t + txt[i]) % q;      //t =txt[i]+t; it turns out that we can use ASCII table
    }

    for (i = 0; i <= N - M; i++) {

        // Check the value of the chunck of text and pattern
        // If the hash values match then check corresponding characters

        if (p == t) {
            for (j = 0; j < M; j++) {
                if (txt[i + j] != pat[j]) {
                    break;
                }
```

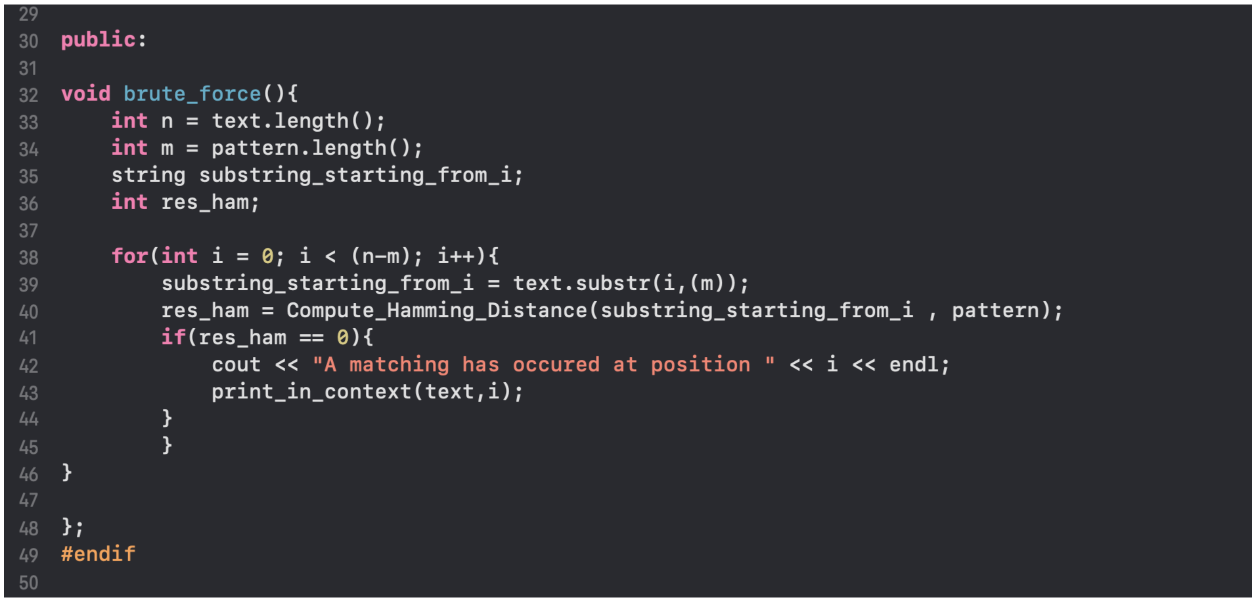


```
class Rabin_Karp_algorithm{
void run_rabin_karp(int q)
            }

            if (j == M)
                cout << "Pattern found at index " << i
                    << endl;
        }
        if (i < N - M) {
            t = (d * (t - txt[i] * h) + txt[i + M]) % q;  //t = (t-txt[i]); using ASCII
            t += txt[i+M];

            // to handle negative t values

            if (t < 0)
                t = (t + q);
        }
    }
}
void set_pattern(string pat_to_set) {pattern = pat_to_set;}
void set_text(string txt_to_set) {text = txt_to_set;}
string get_text(){
    return text;
}
string get_pattern(){
    return pattern;
}
};
#endif
```

Brute Force using hamming distance:



```cpp
#ifndef brute_force_alg
#define brute_force_alg

#include "Database.h"


class Brute_Force_algorithm{

int Differences = 0;
Database_handler * documents_for_search;
string text,pattern;

int Compute_Hamming_Distance(string one, string two){
    for(int i = 0; i < one.length(); i++){
        if(one[i] != two[i]){Differences++;}
    }
    return Differences;
}

void print_in_context(int index){
    cout << "\nThe context of the searched text in the orginial text: \n";
    for(int i = 50; i > 0; i--){
        cout << text[index - i];
    }
    for(int j = 0; j < 50; j++){
        cout << text[index + j];
    }
}
```



```cpp

public:

void brute_force(){
    int n = text.length();
    int m = pattern.length();
    string substring_starting_from_i;
    int res_ham;

    for(int i = 0; i < (n-m); i++){
        substring_starting_from_i = text.substr(i,(m));
        res_ham = Compute_Hamming_Distance(substring_starting_from_i , pattern);
        if(res_ham == 0){
            cout << "A matching has occured at position " << i << endl;
            print_in_context(text,i);
        }
        }
}

};
#endif

```