b

Department of Computer Science and Engineering, AUC

Spring 2021

CSCE 2202-01

Simple Plagiarism Detector

Project Report

Submitted to:

Dr. Amr Goneid

Submitted by:

Ahmed Fathy 900192723

Amr Sallam 900196011

Mohamed Ali 900192994

Mohamed Riad 900192833

11/05/2021

# Abstract:

This report demonstrates different string matching algorithms. It lists the implementations of four of the most well-known algorithms alongside a critique and analysis of their time and space complexities and the idea behind each of them. Also we are going to be illustrating the results of testing the four algorithms on 4 text files and another plagiarized file to conclude which of them is the most accurate and efficient when it comes to saving time.

# Introduction:

Strings are considered one of the most important building blocks for data. So it is logical to construct searching algorithm to aid us to understand and control them more. String searching algorithms is type of string class algorithms that play an essential role in real-life problems, i.e. plagiarism checker, search engines, etc. They are divided into two main categories, exact string matching algorithm and approximate string matching algorithm. In this report we will be trying 4 exact string matching algorithms which are Naïve Algorithm, Rabin Karp Algorithm, Knuth Morris Pratt (KMP) Algorithm, and Boyer-Moore Algorithm. We are going to analyze each one from time and space complexities perspective and implement them and the report would communicate the results from testing the 4 algorithms on 4 different files with a test text file.

# Problem Definition:

Our problem to be solved would be to search all occurrences of a defined string (pattern) in a large string (text or sequences) such that each matching is perfect. This is done by implementing four exact string matching algorithms, naïve algorithm, Rabin Karp Algorithm, KMP algorithm, and Boyer-Moore algorithm, to check for plagiarized text in a test text file from different 4 other text files.

# Methodology:

**Brute Force Matching**

This is the simplest and most straightforward solution to the problem. Simply, it uses the sliding window approach. Thus, it slide the pattern over the whole text and calculate the hamming distance between the pattern and the current alignment along the way. When the distance equal zero, this means the pattern matches this part of the text.

**Pseudocode**

function ApproximateStringMatching\_Search (string s[1..n], string pattern[1..m])

for i from 1 to n-m+1{

distance=0

for j from 1 to m+1

if s[i+j] != pattern[j]

distance += 1

if(distance=0)

return i}

return -1

}

**Boyer Moore**

Before starting to match, Boyer Moore Algorithm does pre-processing to allow for multiple shifts. And, using the approaches of bad character heuristic and good suffix heuristic, in each iteration, it slides the pattern by an amount equal to the maximum amount suggested by these two approaches. Contrary to other matching algorithms, Boyer Moore starts the matching process from the last element of the pattern. We used mainly Bad Character Heuristic.

Here is how it works:

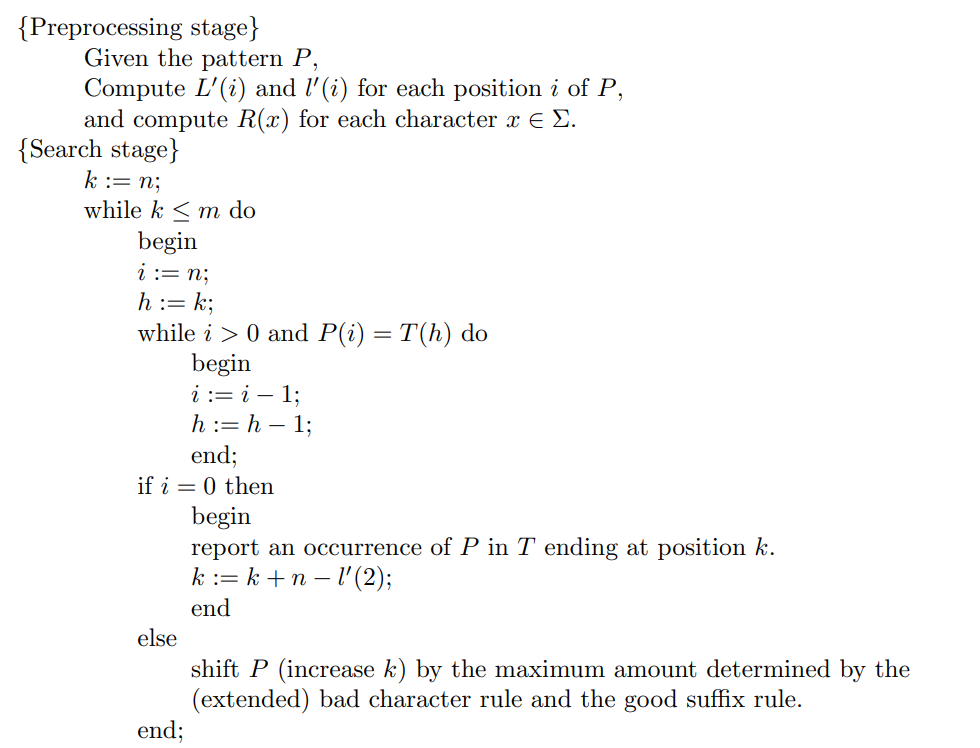
1. **Constructing bad Char Heuristic array (Pre-processing).**

Simply, after this function initializes all occurrences of the badChar array with -1s , then it set only the actual value for the last occurrence of a character.

1. **Matching**

The matching based on bad character heuristic is quite simple. We call the character of the text that mismatches with a character the pattern a bad character. Whenever we face this mismatch we start shifting the pattern until we reach on of two scenarios : (1) This mismatch becomes a match. (2) the pattern is completely moved past the bad character location.

**Pseudocode**

****

**KMP**

The basic idea behind KMP’s algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match.

Here is how it works:

1. **Constructing the auxiliary lps[].**

In this stage, we pre-process the pattern to construct the longest proper prefixes (lps) which is used to skip characters while matching.

In the pre-processing part, we calculate values in lps[]. To do that, we keep track of the length of the longest prefix suffix value (we use len variable for this purpose) for the previous index. We initialize lps[0] and len as 0. If pat[len] and pat[i] match, we increment len by 1 and assign the incremented value to lps[i]. If pat[i] and pat[len] do not match and len is not 0, we update len to lps[len-1]

Examples of lps[] construction:

For the pattern “AABAACAABAA”,

lps[] is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]

**Pseudocode**:

Begin

   length := 0

   prefArray[0] := 0

   for all character index ‘i’ of pattern, do

      if pattern[i] = pattern[length], then

         increase length by 1

         prefArray[i] := length

      else

         if length ≠ 0 then

            length := prefArray[length - 1]

            decrease i by 1

         else

            prefArray[i] := 0

   done

End

1. **Searching process.**

Unlike [Naive algorithm](https://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/), where we slide the pattern by one and compare all characters at each shift, we use a value from lps[] to decide the next characters to be matched. The idea is to not match a character that we know will anyway match.

So, how to know the position of the next shift:

We start comparison of pat[j] with j = 0 with characters of current window of text.

We keep matching characters txt[i] and pat[j] and keep incrementing i and j while pat[j] and txt[i] keep matching.

When we see a mismatch

We know that characters pat[0..j-1] match with txt[i-j…i-1] (Note that j starts with 0 and increment it only when there is a match).

We also know (from above definition) that lps[j-1] is count of characters of pat[0…j-1] that are both proper prefix and suffix.

From above two points, we can conclude that we do not need to match these lps[j-1] characters with txt[i-j…i-1] because we know that these characters will anyway match. Let us consider above example to understand this.

**Pseudocode:**

Begin

   n := size of text

   m := size of pattern

   call findPrefix(pattern, m, prefArray)

   while i < n, do

      if text[i] = pattern[j], then

         increase i and j by 1

      if j = m, then

         print the location (i-j) as there is the pattern

         j := prefArray[j-1]

      else if i < n AND pattern[j] ≠ text[i] then

         if j ≠ 0 then

            j := prefArray[j - 1]

         else

            increase i by 1

   done

End

**Rabin-Karp**

The main idea of the Rabin Karp algorithm is that it matches a calculated hash value of the pattern with a calculated hash value of the current substring of the text. If the hash values match, it starts matching individual characters.

1. **Calculating hash value of the pattern and a window of the text.**

Hashing function can be anything the programmer can choose, but as the different versions of Rabin-Karp are of different versions of the hashing function, the hashing function is the core of this algorithm. A good hashing function and the currently used is:

hash( txt[s+1 .. s+m] ) = ( d ( hash( txt[s .. s+m-1]) – txt[s]\*h ) + txt[s + m] ) mod q   
hash( txt[s .. s+m-1] ) : Hash value at shift s.   
hash( txt[s+1 .. s+m] ) : Hash value at next shift (or shift s+1)   
d: Number of characters in the alphabet   
q: A prime number   
h: d^(m-1)

1. **Search process.**

The process is as follows:

After calculating the hash value of the pattern.

Calculate the hash value of the first window of text.

If the hash values match, then check for characters one by one.

If it is a characters match, here we stop

If the hash values do not match, shift by one and calculate the hash value of the new window of text.

Repeat the process until the pattern is found or the text ends.

**Pseudocode**

function RabinKarp(string s[1..n], string pattern[1..m])

hpattern := hash(pattern[1..m]);

for i from 1 to n-m+1

hs := hash(s[i..i+m-1])

if hs = hpattern

if s[i..i+m-1] = pattern[1..m]

return i

return not found

# Data specifications:

The program takes from the user through command line input the following variables: number of files to be compared with, names of the files, name of desired algorithm. The program deals typically with .txt files. The program takes two set of files. The first set contains a user-defined number of original text files with which they want to compare the potentially plagiarized file. The second set contains only one file which is the potentially plagiarized file.

# Experimental results:

Input files:

We used four texts

**Text1:**

The Nile River is the longest river in the world, 6700 km long, which runs north through north-eastern Africa through ten countries: Egypt, Sudan, Ethiopia, Tanzania, Uganda, Rwanda, Congo, Eritrea, Burundi, and Kenya till it discharges in the Mediterranean Sea. The Nile has two main tributaries, the White Nile and the Blue Nile, that meet together at Khartoum in Sudan. The White Nile emerges from Burundi and its contribution is small to the flow of the Nile; however, the Blue Nile, which originates from the water tower highlands in Ethiopia, feeds the majority of the Nile.

**Text2:**

Since my childhood, I have been interested in education and learning. I started my journey as a student at Al-Azhar where I have completed my primary education. The only thing I remember clearly about this phase is that at my 4th grade I started depending on myself in learning. I remember my suffering at the beginning of the 4th grade as there were new subjects introduced such as science and social studies, but I managed to overcome this difficulty alone. After this turning point, I used to be a smart and studious student. I graduated from the primary level with the highest grades. After that, my father insisted on transferring me to a public school. He said that public schools would guarantee me more chances in the future, which later proved to be accurate. At my preparatory school, I began defining who exactly I want to be. I realized that I love learning and I want to pursue knowledge as much as I can. At that time, I joined many academic competitions in my school and I managed to be the highest scorer. I spent this 3-year phase coming first in all my exams every year and graduated with 99.5 percent.

**Text3:**

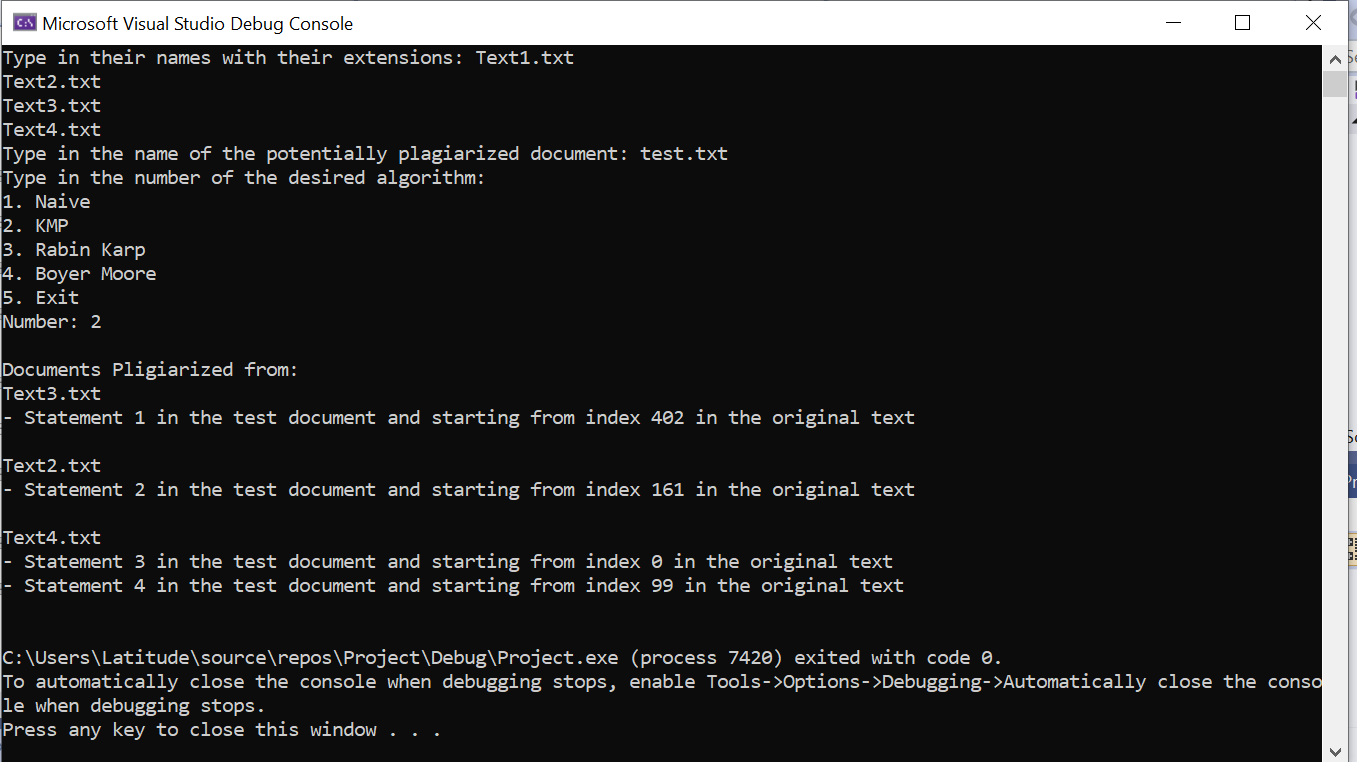
The most significant phase of my life was about to start, which is my secondary level. After I had graduated from my preparatory school, I heard about new schools for smart students called STEM schools. The school name, STEM, stands for Science, Technology, Engineering and Mathematics. I searched about these schools and discovered that they use different teaching techniques from which I am used to. In these schools, students learn how to research about the topic they want, how to think in a creative manner and how to solve the problems they encounter using an effective method called the EDP, Engineering Design Process.

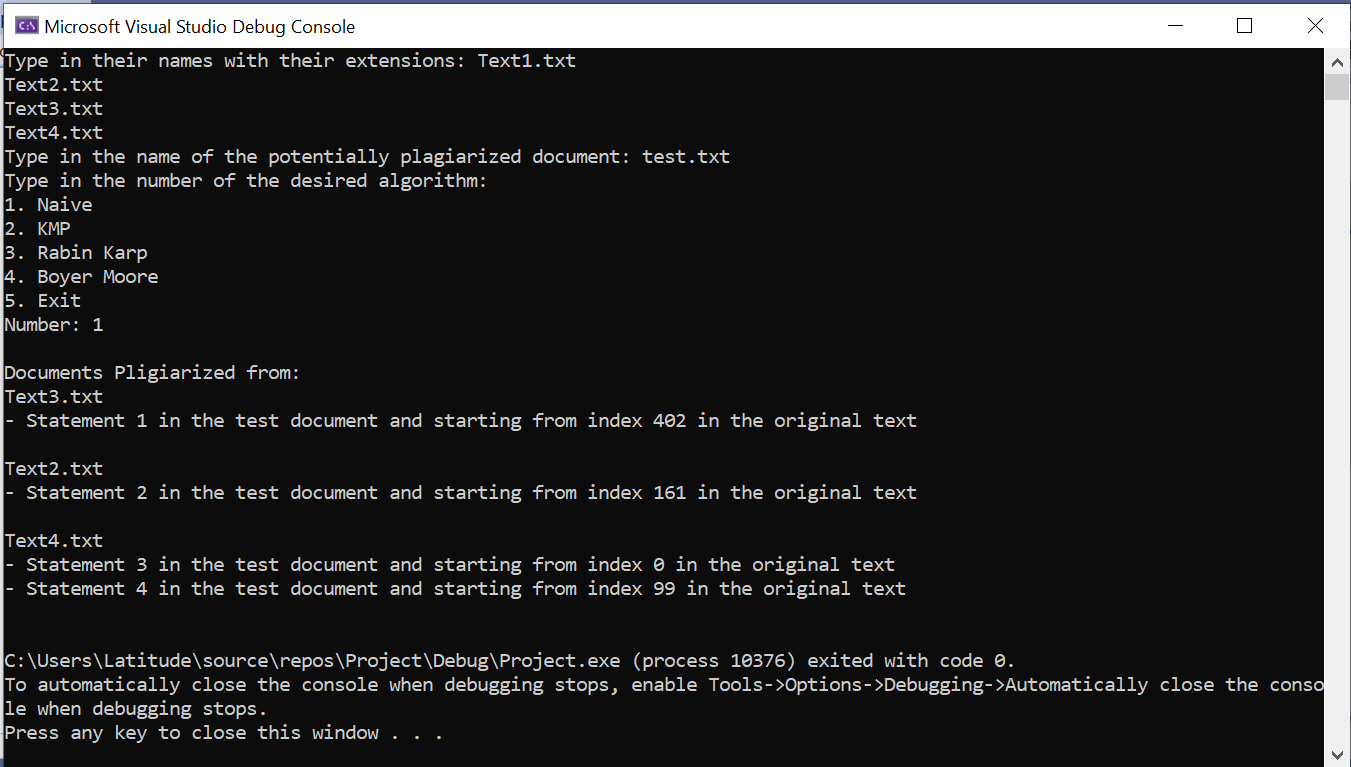
**Text4:**

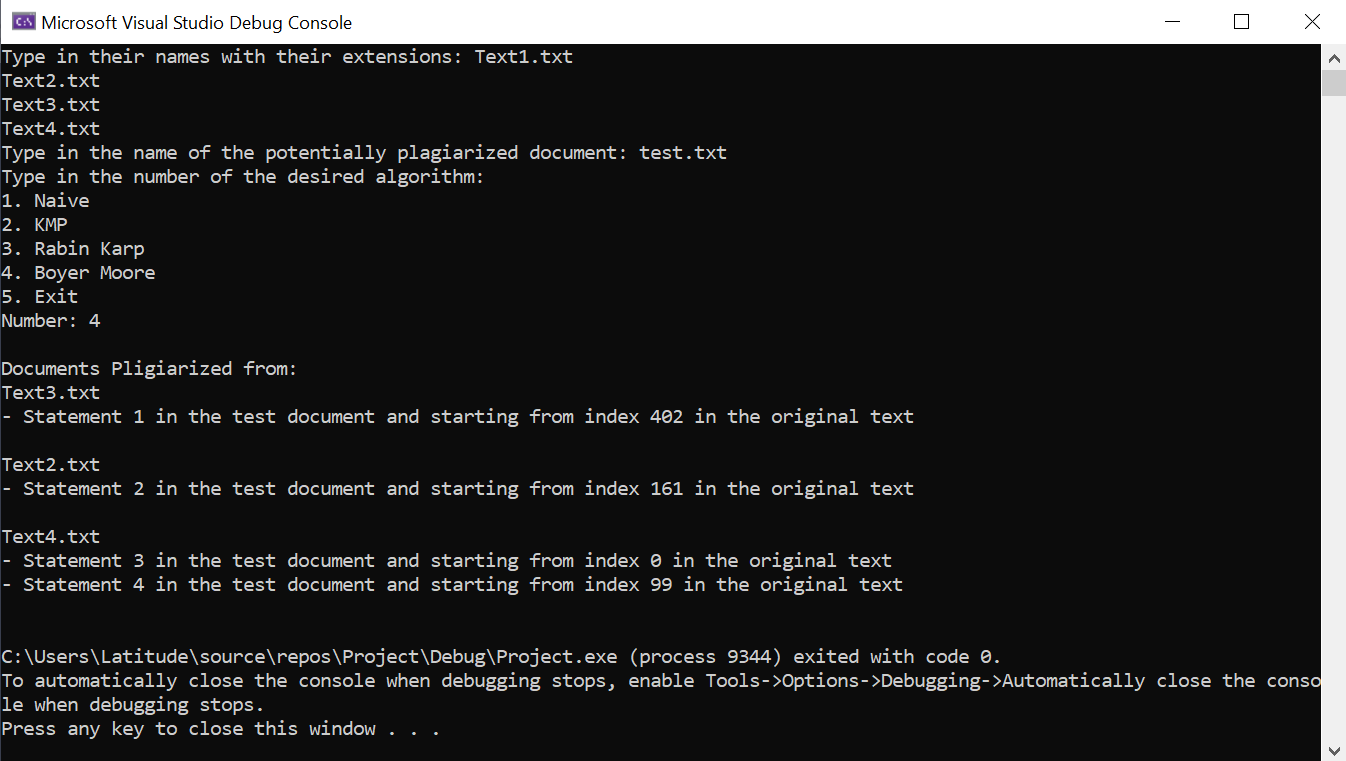
Disputes about water and allocating shares are prevalent acts that we hear frequently on the news. Africa is witnessing the most vigorous contention about water shares in the current time. Ethiopia’s decision to construct the Grand Ethiopian Renaissance Dam (GERD) has received an entire refusal from the Egyptian side. The reason behind the GERD build is that it will contribute substantially to the development of the upstream countries and provide them with the required electrical power towards that development.

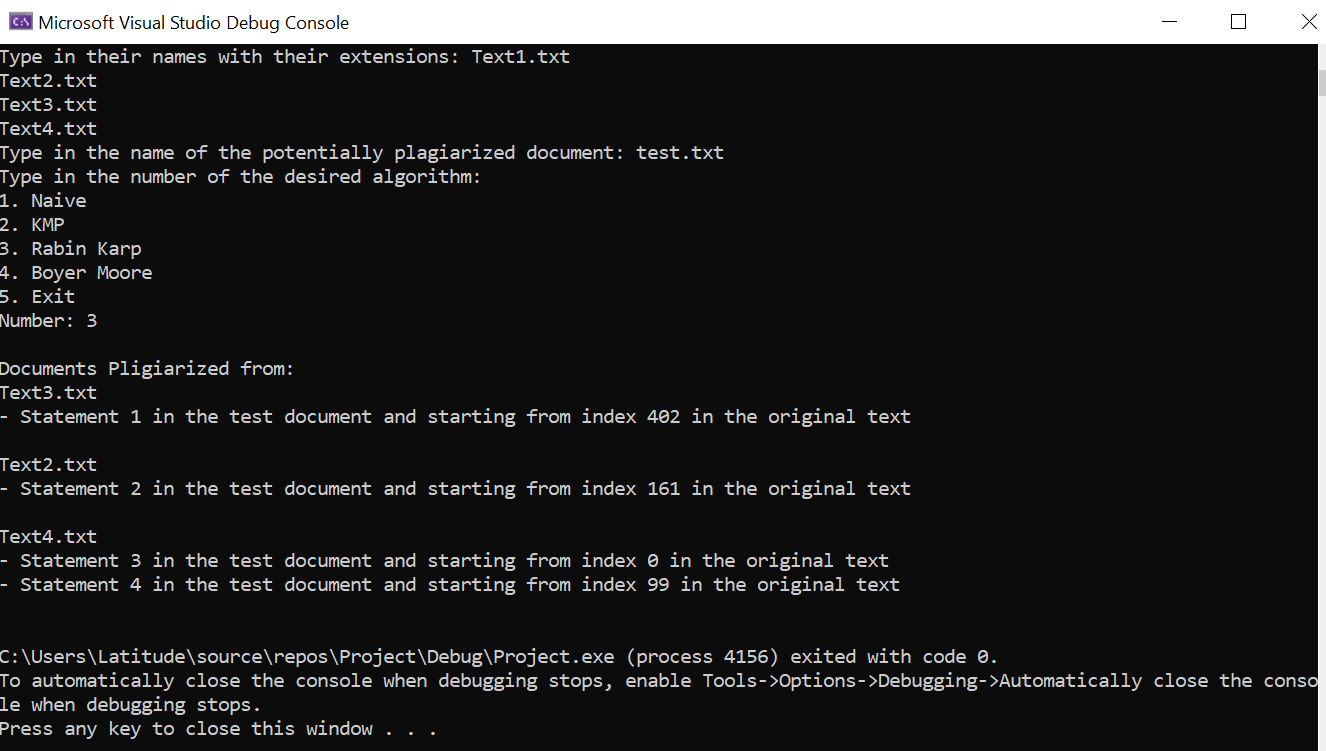
**Potentially plagiarized text:**

In these schools, students learn how to research about the topic they want, how to think in a creative manner and how to solve the problems they encounter using an effective method called the EDP, Engineering Design Process. The only thing I remember clearly about this phase is that at my 4th grade I started depending on myself in learning. Disputes about water and allocating shares are prevalent acts that we hear frequently on the news. Africa is witnessing the most vigorous contention about water shares in the current time.

**KMP**

**Naïve**

**Boyer Moore**

**Rabin Karp**

We can see that the output for every algorithm is the same.

# Analysis and critique:

**Brute Force Matching**

**Time:**

* We used an outer for loop from 0 to N-M to create the sliding window.
* We also used a loop from 0 to M to loop over the characters of the pattern.
* Thus, the total complexity is O (M(N-M))

**Space:**

* There are no vectors, arrays, or any containers. There are only variables of type int, so it is of O(1).

**Boyer Moore**

**Time:**

* The pre-processing typically takes O(M) in all cases.
* Given text size = N and pattern size = M, the search algorithm takes O(MN); this case occurs when the characters used in the pattern are identical to those used in the text. On the other hand, the algorithm take O(N/M) in best case; this occurs when none of the characters of the pattern is present in the text.

**Space:**

* We only used extra dynamic array of size M (the pattern’s size) in the pre-processing stage. So, space complexity is O(M).

**KMP**

**Time:**

* In the pre-processing stage, where we compute the lps[], we used a while loop that loops over the pattern string from 1 to M (the pattern’s size). What is inside the while loop of O(1), so it’s of O(M).
* In the search process, we used a while loop that loops over the text string from 0 to N (the text’s size). What is inside the while loop if of O(1), so it is of O(N).
* Total will be O(N+M)

**Space:**

* We only used extra dynamic array of size M (the pattern’s size) to store the longest pattern suffix, so it is of O(M).

**Rabin-Karp**

**Time:**

* We used a for loop from 0 to M (pattern’s size) to calculate the hash value of the pattern. What is inside the loop is of O(1), so it is of O(M).
* The average and best-case running time is O(N+M), but worst case is O(NM).
* There are many versions of Rabin-Karp algorithm depending on the hash function used. There is a hash function that makes it always O(N+M), even in the worst case and that is by avoiding the spurious hits (hash values matching, but not characters matching).

**Space:**

* There are no vectors, arrays, or any containers. There are only variables of type int, so it is of O(1).

# References:

* <https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/>
* <https://www.tutorialspoint.com/Knuth-Morris-Pratt-Algorithm>
* <https://www.youtube.com/watch?v=V5-7GzOfADQ&ab_channel=AbdulBari>
* <https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/>
* <https://www.programiz.com/dsa/rabin-karp-algorithm>
* <https://www.youtube.com/watch?v=qQ8vS2btsxI&ab_channel=AbdulBari>
* [Part 4 Brute Force Algorithms.pdf (aucegypt.edu)](http://www1.aucegypt.edu/faculty/cse/goneid/csce2202/Part%204%20Brute%20Force%20Algorithms.pdf)
* [Boyer Moore Algorithm for Pattern Searching - GeeksforGeeks](https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/)