Algorithms and Problem Solving Lab Project 2023

Submitted By:

Parth Ahuja 9921103024

Prachee Mohapatra 992110325

Chaaya Agarwal 9921103026

Akarshit Chauhan 9921103050

Submitted to:

Dr. Nitin Shukla

Ms. Laxmi Chaudhary

Project Title: String Matching Algorithms

Department of CSE/IT

Jaypee Institute of Information Technology University,

Noida

May, 2023

1. Problem Statement:

To find one or all occurrences of a given pattern string of length m in a text string of length n. The challenge is to design efficient algorithms that can handle large inputs in a reasonable amount of time. Traditional algorithms like naive string matching and brute-force search have high time complexity, making them unsuitable for large inputs. Therefore, efficient algorithms like Knuth-Morris-Pratt (KMP), Boyer-Moore (BM), and Rabin-Karp have been developed to tackle this problem. Hence, the problem of string matching algorithms is to find the best algorithm for a given set of input strings, which can efficiently find one or all occurrences of a pattern string in a text string.

2. Introduction

2.1. Motivation

String matching algorithms are critical for many applications, including text processing, bioinformatics, and network security. By developing efficient algorithms, we can improve the performance of these applications, reduce the processing time, and improve the user experience. String matching algorithms are also an excellent example of algorithm design, as they require careful analysis of the problem and creative solutions. By working on a string matching algorithm project, you can learn about algorithm design principles and techniques, which can be applied to other problems.

2.2 Objective

The main objective of this project is to implement one or more string matching algorithms, such as the naive algorithm, Knuth-Morris-Pratt algorithm, Boyer-Moore algorithm, or Rabin-Karp algorithm. The objective can be to understand the algorithm's working principles and implement them in code. The project can aim to compare the performance of different string matching algorithms under different conditions, such as varying pattern and text lengths, character sets, or input formats. The objective can be to identify the strengths and weaknesses of each algorithm and provide recommendations for selecting the most appropriate algorithm for specific scenarios.

Overall, the objective of a project on string matching algorithms can be to deepen the understanding of algorithm design and analysis, improve practical skills in programming and data analysis, and contribute to the development of efficient solutions for string matching problems.

2.3 Contribution

String matching algorithms are widely used in text processing applications such as search engines, plagiarism detection, and natural language processing. By developing more efficient algorithms, we can improve the accuracy and speed of these applications, making them more accessible and user-friendly for people worldwide.

String matching algorithms are essential for DNA sequencing and analysis, which have significant implications for healthcare, agriculture, and environmental studies. By improving the performance and accuracy of string matching algorithms, we can accelerate the progress of these fields and contribute to solving critical challenges such as disease diagnosis and drug development.

This can also help in detecting plagiarised work in workplaces, schools and other academic institutions and ensure that the word submitted is of original value.

3. Description of the Project:

String Matching Algorithms:

String matching algorithms are a set of techniques used to search for one or more occurrences of a pattern within a text or a larger string. These algorithms are widely used in a variety of applications, such as text editors, search engines, and bioinformatics. The performance of a string matching algorithm depends on various factors such as the size of the pattern, the size of the text, the character set used in the pattern and text, and the desired accuracy of the matches.

Algorithms/Approaches used:

1. Brute Force Approach
2. Knuth-Morris-Pratt Algorithm
3. Suffix Trie Approach
4. Rabin Karp Algorithm
5. Automaton Matcher Algorithm

Brute Force Approach:

The brute force approach, also known as the naive algorithm, is the simplest and most straightforward approach for string matching. It involves comparing each character of the pattern with each character of the text until a match is found or the end of the text is reached. This algorithm has a worst-case time complexity of O(mn), where m is the length of the pattern and n is the length of the text.

Knuth-Morris-Pratt Algorithm:

The Knuth-Morris-Pratt (KMP) algorithm is a more efficient string matching algorithm that uses a pre-processed table to avoid unnecessary comparisons. The table is computed from the pattern and is used to determine the maximum length of the proper prefix of the pattern that is also a suffix of a substring of the pattern. The algorithm then uses this table to skip over unnecessary comparisons in the text. The KMP algorithm has a worst-case time complexity of O(m+n), where m is the length of the pattern and n is the length of the text.

Suffix Trie Approach:

The Suffix Trie Approach is a more space-efficient version of the Automaton Matcher Algorithm that uses a trie to store all suffixes of the text. The algorithm then searches for the pattern in the trie, starting from the root and following the edges that match the pattern. The Suffix Trie Approach has a worst-case time complexity of O(mn), where m is the length of the pattern and n is the length of the text. However, it has a high space complexity of O(mn), which may be a limiting factor for large datasets.

Rabin Karp Algorithm:

The Rabin-Karp algorithm is a probabilistic string matching algorithm that uses hashing to compare the pattern and text. The algorithm hashes the pattern and slides it over the text while comparing the hash values. If the hash values match, the algorithm then compares the pattern and text character by character to avoid hash collisions. The Rabin-Karp algorithm has a worst-case time complexity of O(mn), but its average-case time complexity is better than that of the brute force algorithm.

Automaton Matcher Algorithm:

The Automaton Matcher Algorithm constructs an automaton that recognizes all the patterns simultaneously. The automaton is constructed by concatenating the patterns and adding suffix links to the nodes of the trie. The algorithm then follows the links to quickly find matches in the text. The Automaton Matcher Algorithm has a worst-case time complexity of O(m+n), where m is the length of the pattern and n is the length of the text.

Plagiarism Checker:

In our project, if a string match is found after running the chosen algorithm, it returns the percentage of strings that are plagiarized from the input text. In conclusion, a plagiarism checker that uses a more nuanced approach to string matching by limiting the matching to short strings within the text can provide more accurate and reliable results. By providing the percentage of matched strings only if the match occurs within a specific number of words, the checker can reduce the number of false positives and encourage original writing practices.

Choosing the best amongst the given:

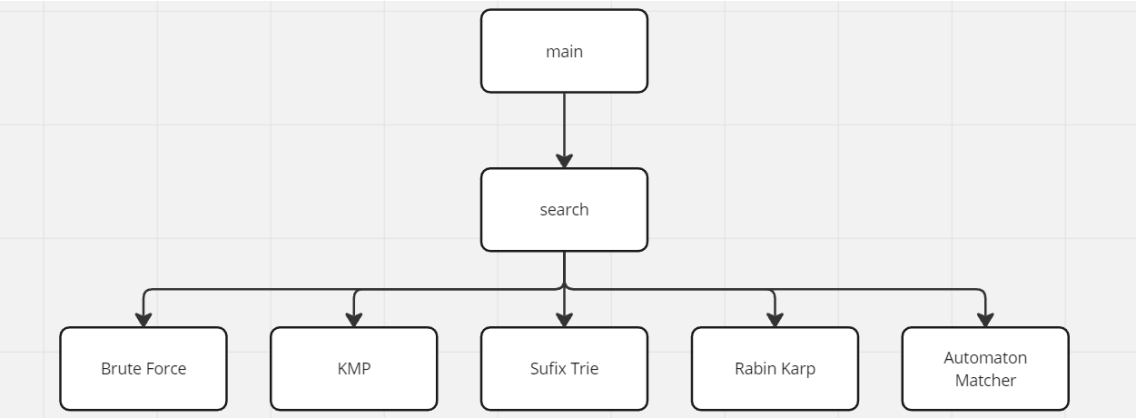
The best approach for string matching among the above-given approaches depends on the specific requirements and constraints of the problem at hand.

- If the pattern is small, then the Brute Force approach can be used as it has a simple implementation and does not require pre-processing.
- If the pattern is large, then the Rabin-Karp algorithm may be a good choice as it has good average-case time complexity and does not require much pre-processing.
- If the pattern changes frequently, then the Suffix Trie Approach may be useful as it allows for fast updates to the trie.
- If multiple patterns need to be matched, then the Automaton Matcher Algorithm may be a good choice as it can be used to construct an automaton that recognizes all the patterns simultaneously.
- If the focus is on the average-case time complexity, then the KMP algorithm may be the best choice.

Therefore, it is important to analyse the requirements of the specific problem before choosing the best approach for string matching.

4. Implementation:

4.1. Workflow diagram



4.2. Program Code

Main:

```
#include <bits/stdc++.h>
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <string.h>
#include <windows.h>

using namespace std;

int cou = 0;

//---------------------------------------------------------------------------//

// brute-force approach
void search(char *pat, char *txt)
{
   int M = strlen(pat);
   int N = strlen(txt);

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

```
        {
            cou++;
        }
    }
}

//-------------------------------------------------------------------------------//

// KMP approach
void computeLPSArray(char *pat, int M, int *lps)
{
    int len = 0;
    lps[0] = 0;

    int i = 1;
    while (i < M)
    {
        if (pat[i] == pat[len])
        {
            len++;
            lps[i] = len;
            i++;
        }
        else
        {
            if (len != 0)
            {
                len = lps[len - 1];
            }
            else
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}
void KMPSearch(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int lps[M];
    computeLPSArray(pat, M, lps);
    int i = 0;
    int j = 0;
    while (i < N)
    {
        if (pat[j] == txt[i])
        {
            j++;
            i++;
```

```
        }
        if (j == M)
        {
            cou++;
            j = lps[j - 1];
        }
        else if (i < N && pat[j] != txt[i])
        {
            if (j != 0)
            {
                j = lps[j - 1];
            }
            else
            {
                i = i + 1;
            }
        }
    }
}

//-------------------------------------------------------------------------------//

// Suffix-trie approach

#define MAX_CHAR 256

class SuffixTrieNode
{
private:
    SuffixTrieNode *children[MAX_CHAR];
    list<int> *indexes;

public:
    SuffixTrieNode()
    {

        indexes = new list<int>;

        for (int i = 0; i < MAX_CHAR; i++)
        {
            children[i] = NULL;
        }
    }

    void insertSuffix(string suffix, int index);

    list<int> *search(string pat);
};

class SuffixTrie
{
```

```
private:
   SuffixTrieNode root;

public:
   SuffixTrie(string txt)
   {

      for (int i = 0; i < txt.length(); i++)
      {
         root.insertSuffix(txt.substr(i), i);
      }
   }

   void search(string pat);
};

void SuffixTrieNode::insertSuffix(string s, int index)
{

   indexes->push_back(index);

   if (s.length() > 0)
   {

      char cIndex = s.at(0);

      if (children[cIndex] == NULL)
      {
         children[cIndex] = new SuffixTrieNode();
      }

      children[cIndex]->insertSuffix(s.substr(1), index + 1);
   }
}

list<int> *SuffixTrieNode::search(string s)
{

   if (s.length() == 0)
   {
      return indexes;
   }

   if (children[s.at(0)] != NULL)
   {
      return (children[s.at(0)])->search(s.substr(1));
   }

   else
      return NULL;
}
```

```
void SuffixTrie::search(string pat)
{
   list<int> *result = root.search(pat);

   if (result == NULL)
   {
      cout << "Pattern not found" << endl;
   }
   else
   {
      list<int>::iterator i;
      int patLen = pat.length();
      for (i = result->begin(); i != result->end(); ++i)
      {
         // cout << "Pattern found at position " << *i - patLen<< endl;
         cou++;
      }
   }
}

//--------------------------------------------------------------------------------//

// Ranbin-Karp approach

#define d 256

void rksearch(char pat[], char txt[], int q)
{
   int M = strlen(pat);
   int N = strlen(txt);
   int i, j;
   int p = 0;
   int t = 0;
   int h = 1;
   for (i = 0; i < M - 1; i++)
   {
      h = (h * d) % q;
   }
   for (i = 0; i < M; i++)
   {
      p = (d * p + pat[i]) % q;
      t = (d * t + txt[i]) % q;
   }
   for (i = 0; i <= N - M; i++)
   {
      if (p == t)
      {
         for (j = 0; j < M; j++)
         {
            if (txt[i + j] != pat[j])
```

```
            {
                break;
            }
        }
        if (j == M)
        {
            cou++;
        }
    }
    if (i < N - M)
    {
        t = (d * (t - txt[i] * h) + txt[i + M]) % q;
        if (t < 0)
        {
            t = (t + q);
        }
    }
  }
}

//--------------------------------------------------------------------------------//

// Automaton-Matcher approach

#define NO_OF_CHARS 256

int getNextState(char *pat, int M, int state, int x)
{
  if (state < M && x == pat[state])
  {
    return state + 1;
  }
  int ns, i;
  for (ns = state; ns > 0; ns--)
  {
    if (pat[ns - 1] == x)
    {
      for (i = 0; i < ns - 1; i++)
      {
        if (pat[i] != pat[state - ns + 1 + i])
        {
          break;
        }
      }
      if (i == ns - 1)
      {
        return ns;
      }
    }
  }
  return 0;
```

```
}
void computeTF(char *pat, int M, int TF[][NO_OF_CHARS])
{
   int state, x;
   for (state = 0; state <= M; ++state)
   {
      for (x = 0; x < NO_OF_CHARS; ++x)
      {
         TF[state][x] = getNextState(pat, M, state, x);
      }
   }
}
void amsearch(char *pat, char *txt)
{
   int M = strlen(pat);
   int N = strlen(txt);
   int TF[M + 1][NO_OF_CHARS];
   computeTF(pat, M, TF);
   int i, state = 0;
   for (i = 0; i < N; i++)
   {
      state = TF[state][txt[i]];
      if (state == M)
      {
         cou++;
      }
   }
}

//--------------------------------------------------------------------------------//

void dispalyAlgoName()
{

   cout << "\t\t\t\t\t      From the following list of String Matching Algorithms select one: " << endl;
   cout << "\t\t\t\t\t      1. Brutte Force Approach" << endl;
   cout << "\t\t\t\t\t      2. Knuth-Morris-Pratt Algorithm" << endl;
   cout << "\t\t\t\t\t      3. Sufix Trie Approach" << endl;
   cout << "\t\t\t\t\t      4. Rabin-Karp Algortihm" << endl;
   cout << "\t\t\t\t\t      5. Automaton-Matcher Approach \n\n" << endl;
}
int main()
{
   char txt[1000];
   char pat[1000];
   string rtxt;
   string rpat;
   system("CLS");
   cout << "\n\n\n\n\n\n\n\t\t\t\t\t      STRING MATCHING ALGORITHMS \n\n";
   dispalyAlgoName();
   int ch;
```

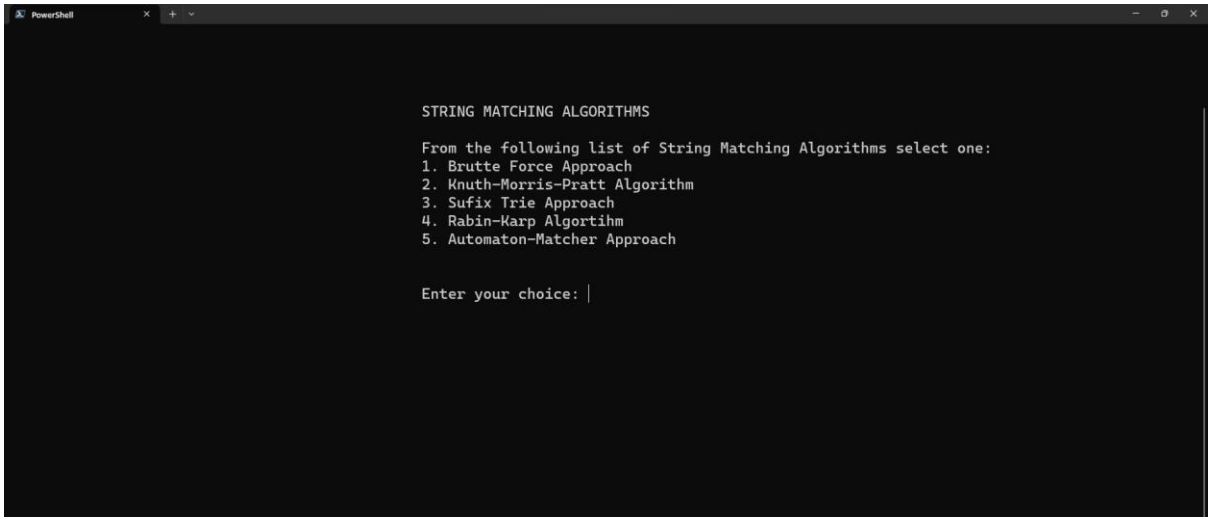
```
cout << "\t\t\t\t\t       Enter your choice: ";
cin >> ch;
system("CLS");

do
{
   getchar();
   cout << "\t\t\t\t\t       Enter or paste the source text: \n\n";
   getline(cin,rtxt);
   cout << "\t\t\t\t\t       Enter or paste the text to be checked: \n\n";
   getline(cin,rpat);
   system("CLS");

   int len1 = rtxt.length();
   int len2 = rpat.length();
   strcpy(txt, rtxt.c_str());
   strcpy(pat, rpat.c_str());

   cout << "\n\n       TEXT: " << txt << "\n\n";
   cout << "       PATTERN: " << pat << "\n\n";

   if (ch == 1)
   {
      search(pat, txt);
      cout << "       Number of matches of \"" << pat << "\" is " << cou << endl << endl;
      float a=0.0,b=0.0;
      a = cou*len2*100;
      b = a/len1;
      cout<< "       Percentage similar : " << b <<"%" << endl;
      system("PAUSE");
      system("CLS");
      cou = 0;
   }
   else if (ch == 2)
   {
      KMPSearch(pat, txt);
      cout << "       Number of matches of \"" << pat << "\" is " << cou << endl << endl;
      float a=0.0,b=0.0;
      a = cou*len2*100;
      b = a/len1;
      cout<< "       Percentage similar : " << b <<"%" << endl;
      system("PAUSE");
      system("CLS");
      cou = 0;
   }
   else if (ch == 3)
   {
      SuffixTrie S(txt);
      S.search(pat);
      cout << "       Number of matches of \"" << pat << "\" is " << cou << endl << endl;
      float a=0.0,b=0.0;
```

```
    a = cou*len2*100;
    b = a/len1;
    cout<< "      Percentage similar : " << b <<"%" << endl;
    system("PAUSE");
    system("CLS");
    cou = 0;
}
else if (ch == 4)
{
    int q = 13;
    rksearch(pat, txt, q);
    cout << "       Number of matches of \"" << pat << "\" is " << cou << endl << endl;
    float a=0.0,b=0.0;
    a = cou*len2*100;
    b = a/len1;
    cout<< "       Percentage similar : " << b <<"%" << endl;
    system("PAUSE");
    system("CLS");
    cou = 0;
}
else if (ch == 5)
{
    amsearch(pat, txt);
    cout << "       Number of matches of \"" << pat << "\" is " << cou << endl << endl;
    float a=0.0,b=0.0;
    a = cou*len2*100;
    b = a/len1;
    cout<< "       Percentage similar : " << b <<"%" << endl;
    system("PAUSE");
    system("CLS");
    cou = 0;
}
else
{
    cout << "\t\t\t\t\t       Invalid Choice!" << endl;
    system("PAUSE");
    system("CLS");
}
int a;
cout << "\n\n\n\n\n\n\n\n\t\t\t\t\t       Press 0 to continue or -1 to exit: ";
cin >> a;
system("CLS");
if (a == 0)
{
    cout << "\n\n\n\n\n\n\n\n";
    dispalyAlgoName();
    cout << "\t\t\t\t\t       Enter your choice: ";
    cin >> ch;
    system("CLS");
}
else
```

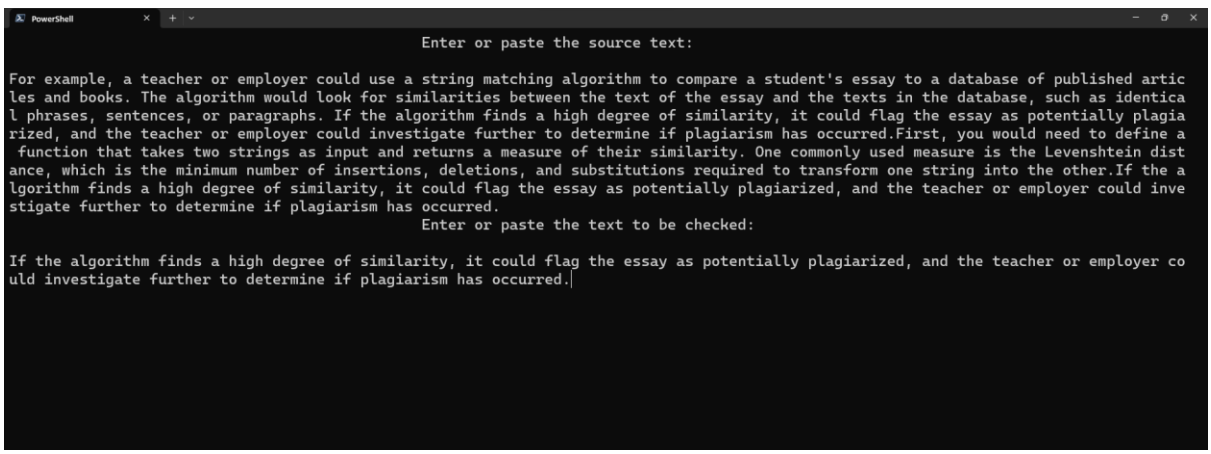
```
        {
            exit(0);
        }
    } while (ch > 0);

    return 0;
}
```

5. Results

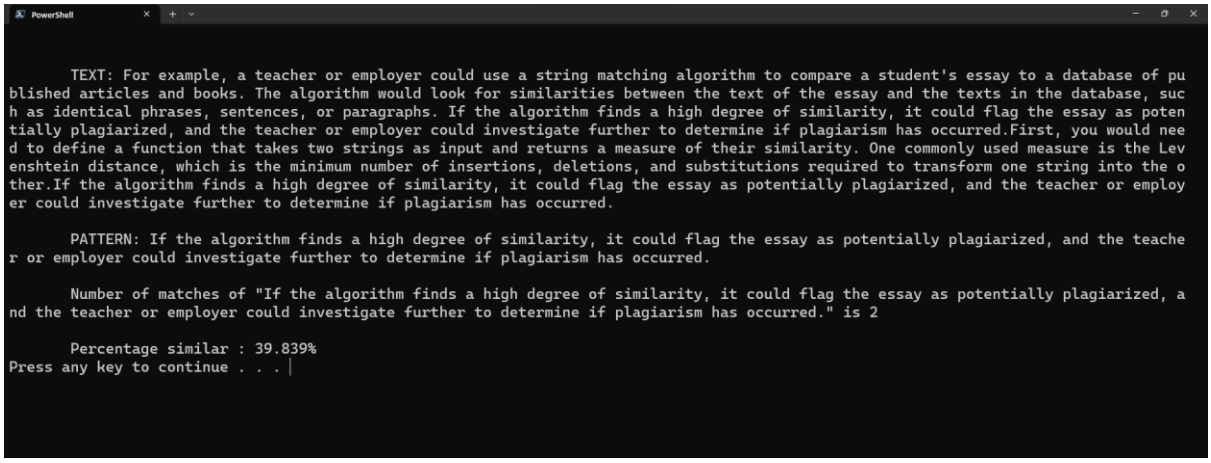
Main Window:

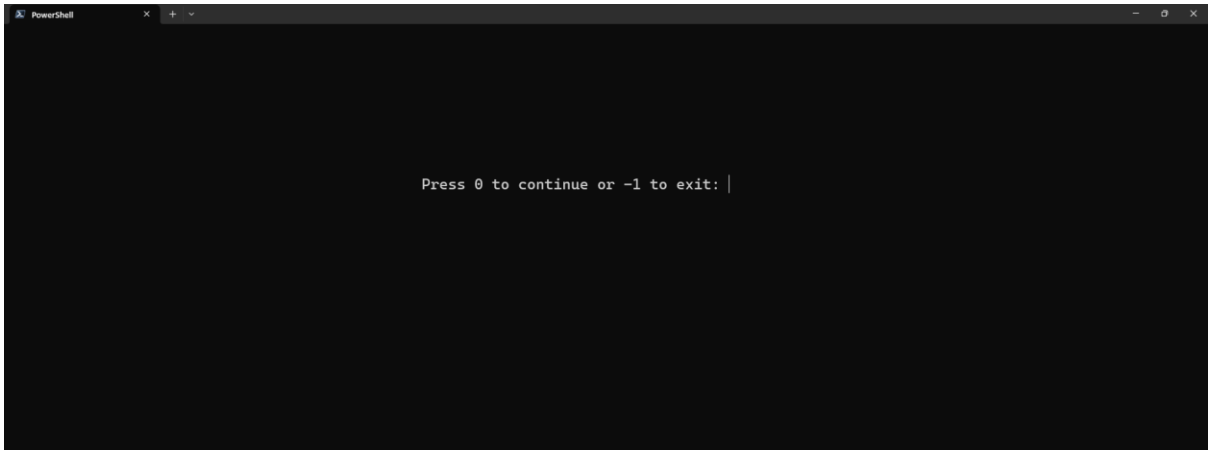


Input:



Result:


PowerShell
TEXT: For example, a teacher or employer could use a string matching algorithm to compare a student's essay to a database of pu
blished articles and books. The algorithm would look for similarities between the text of the essay and the texts in the database, suc
h as identical phrases, sentences, or paragraphs. If the algorithm finds a high degree of similarity, it could flag the essay as poten
tially plagiarized, and the teacher or employer could investigate further to determine if plagiarism has occurred.First, you would nee
d to define a function that takes two strings as input and returns a measure of their similarity. One commonly used measure is the Lev
enshtein distance, which is the minimum number of insertions, deletions, and substitutions required to transform one string into the o
ther.If the algorithm finds a high degree of similarity, it could flag the essay as potentially plagiarized, and the teacher or employ
er could investigate further to determine if plagiarism has occurred.
PATTERN: If the algorithm finds a high degree of similarity, it could flag the essay as potentially plagiarized, and the teache
r or employer could investigate further to determine if plagiarism has occurred.
Number of matches of "If the algorithm finds a high degree of similarity, it could flag the essay as potentially plagiarized, a
nd the teacher or employer could investigate further to determine if plagiarism has occurred." is 2
Percentage similar : 39.839%
Press any key to continue . . .


PowerShell
Press 0 to continue or -1 to exit:

Comparative analysis of all approaches/algorithms used:

Algorithm	Worst-case Time Complexity	Average-case Time Complexity	Space Complexity
Brute Force Approach	**O(mn)**	**-**	**O(1)**
Knuth-Morris-Pratt (KMP) Algo	**O(m+n)**	**O(m)**	**O(m)**
Suffix Trie Approach	**O(mn)**	**O(m+n)**	**O(mn)**
Rabin Karp Algo	**O(mn)**	**O(m+n)**	**O(1)**
Automaton Matcher Algo	**O(m+n)**	**-**	**O(m)**

Note: m is the length of the pattern, n is the length of the text.

Time taken by each algorithm:

S.no	Native	KMP	Tries	Automata	Rabin Karp
1.	0.878	0.878	1.130	0.871	0.854
2.	0.864	0.876	0.870	0.847	0.960
3.	0.860	1.106	0.890	0.868	0.860
4.	0.857	0.864	0.876	0.870	1.121
5.	0.867	0.860	0.871	0.868	0.863
6.	1.136	1.119	1.132	0.874	1.659
7.	0.859	0.866	0.876	0.871	0.882
8.	0.840	0.862	1.159	0.855	0.852
9.	0.894	0.877	1.119	0.871	0.856
10.	1.129	0.875	0.880	0.865	0.847
Mean	0.9184	0.9183	0.9803	0.8660	0.9754

Choosing the best amongst the given:

The best approach for string matching among the above-given approaches depends on the specific requirements and constraints of the problem at hand.

- If the pattern is small, then the Brute Force approach can be used as it has a simple implementation and does not require pre-processing.
- If the pattern is large, then the Rabin-Karp algorithm may be a good choice as it has good average-case time complexity and does not require much pre-processing.
- If the pattern changes frequently, then the Suffix Trie Approach may be useful as it allows for fast updates to the trie.
- If multiple patterns need to be matched, then the Automaton Matcher Algorithm may be a good choice as it can be used to construct an automaton that recognizes all the patterns simultaneously.
- If the focus is on the average-case time complexity, then the KMP algorithm may be the best choice.

Therefore, it is important to analyse the requirements of the specific problem before choosing the best approach for string matching.

6.Conclusion:

In this report, we have discussed the Brute Force approach, KMP Algorithm, Rabin-Karp Algorithm, Automaton Matcher Algorithm, and Suffix Trie Approach for string matching. Each algorithm has its advantages and disadvantages, and the choice of algorithm depends on the problem requirements. KMP algorithm is more efficient for small patterns, Rabin-Karp algorithm is more suitable for large patterns, Automaton Matcher Algorithm is useful for pattern matching with multiple patterns, while the Suffix Trie Approach is useful for problems where the pattern changes frequently.

7.References

[1]. https://www.javatpoint.com/daa-string-matching-introduction

[2]. https://www.geeksforgeeks.org/applications-of-string-matching-algorithms/

[3].https://www.cs.auckland.ac.nz/courses/compsci369s1c/lectures/GG-notes/CS369-StringAlgs.pdf