جامعة أم القرى
UMM AL-QURA UNIVERSITY

String Matching Algorithms

Algorithms Project

14012402-4

Ruba Balubaid 442017269
Yara Al similan 442011215
Noor Alhashmi 442007783
Shaden Anagreh 442017295
Hadel Alnasiri 442004645

Under Supervision:

Dr. Manal Alharbi

1444 - 2023

- **Project objectives:**

This project aims to present 3 different data structures to implement 3 different string matching algorithms (KMP, Naïve, and Longest common subsequence) on the same data, for analyzing and comparing the overall performances of each algorithm.

- **Programming Language :**

Python.

- **Project specifications:**

For this project, we implement the following 3 String Matching Algorithms:

- Naive string matching algorithm:

algorithm pseudocode:

• Input: two strings txt, pat
• Output: the index in which a pattern (pat) match has been found in the text (txt).

```
Naïve_String_Matching(txt, pat)
1) n = length(txt)
2) m = length(pat)
3) for i = 0 to (n-m)
4)     if pat[1…m] = txt[i+1….i+m];
5)         print "Match found at " i
```

- Longest Common Subsequence string matching algorithm:

  algorithm pseudocode:

LCS-LENGTH (X,Y)
1) m = X.length
2) n = Y.length
3) let b[1…m,1…n] and c [0..m,0..n] be new tables
4) for I = 1 to m
5) c[I,0] = 0
6) for j = 0 to n
7) c[I,0] = 0
8) for I = 1 to me
9) for j = 1 to n
10) if X == Y
11) c[I,j] = c[i-1, j-1] +1
12) b[I,j]= c[i-1, j-1] +1
13 elseif c[i-1, j] > c[I,j-1]
14) c[I,j] = c[i-1,j]
15) b[I,j] = c[i-1,j]
16) else c[I,j] = c[I, j -1]
17) b[I,j] = c[I, j -1]
18) return c and b

- KMP string matching algorithm:

algorithm pseudocode:

COMPUTE- PREFIX- FUNCTION (P)

1. m ←length [P] //'p' pattern to be matched
2. Π [1] ← 0
3. k ← 0
4. for q ← 2 to m
5. do while k > 0 and P [k + 1] ≠ P [q]
6. do k ← Π [k]
7. If P [k + 1] = P [q]
8. then k← k + 1
9. Π [q] ← k
10. Return Π

KMP-MATCHER (T, P)

1. n ← length [T]
2. m ← length [P]
3. Π← COMPUTE-PREFIX-FUNCTION (P)
4. q ← 0 // numbers of characters matched
5. for i ← 1 to n // scan S from left to right
6. do while q > 0 and P [q + 1] ≠ T [i]
7. do q ← Π [q] // next character does not match
8. If P [q + 1] = T [i]
9. then q ← q + 1 // next character matches
10. If q = m // is all of p matched?
11. then print "Pattern occurs with shift" i - m
12. q ← Π [q]

- **Project specifications(cont.):**

We present the following 3 different data structures to implement each of the algorithms:

3. Hash table: We used this data structure because they allow for fast lookups and can store large amounts of data. They also provide constant time access to elements, which is important for string matching algorithms that need to quickly find matches.
2. List: Lists are a good choice for string matching algorithms because they are easy to traverse and can store large amounts of data. Additionally, lists can be sorted, which is useful for finding matches quickly.
3. Linked List: Linked lists are a good choice for string matching algorithms because they allow for fast insertion and deletion of elements, which is important when dealing with large datasets. Additionally, linked lists can be traversed quickly, which is important when searching for matches.

- **README :**

- *Required Libraries:*

Time, pandas

- *Data's file:*

We have three file contains data. We use "both_covid_data" file.

- ## README(cont.) :

- ***Project Files:***

The main file (StringMatchingAlgo), which contains as well as the data file, and the Project main report, the following three files**:**

1. **Naïve_Algo:**

This file contains 3 files, each present a different data structure to implement the Naive String Matching algorithm (Same data used):

1. Naïve Algo – List: *(.py)*

In this file, data will be read from the given file and stored in a data frame. A method named naive_string_match() will search for a pattern match in the data stored in the data frame, and return the indices for each pattern match has been found in a List named matches.

2. Naïve Algo – Linked List: *(.py)*

This file contains 2 classes (Node, Linked List), to create a linked list that contains nodes that hold data elements from the data frame, and a pointer to the next element. Linked List class contains 3 methods: append(), to add new nodes, printList(), to print the linked list content, and method extract(), which aims to extract the data from each node to be send to the method named naive_string_match(), which aims to search -in each data element - for a pattern matching and return the index in which it has been found.

3. Naïve Algo – Hash Table: *(.py)*

In this file the data will be stored from the data frame in a Hash Table, and it contains 2 methods: createHashTable() which create a hash table of all the substrings of the given pattern, and searchPattern() method which searches for all occurrences of the pattern (Naïve algorithm) in the given data. Method accepts each data element stored in the hash table and returns the index in which a pattern match was found.

- **README(cont.) :**

2. KMP algo:

This file contains 3 files, each present a different data structure to implement the KMP String Matching algorithm (Same data used) :

1. KMPList: *(.py)*

In this file there are some methods:

1) readlist: for read a data but we do not use

2,3) KMP and computeLPSArray: algorithm's method

4) read_csv: we use it for a read data

2. KMPLinkedList: *(.py)*

In this file there are a Node class and KMPSearch method for implement KMP algorithm. For read data we use a read_csv method which is convert a csv file to data frame and we use a pandas library here.

3. KMPHashTable: *(.py)*

This file contains same methods in KMPList file, but here we use a hash table data structure.

3. Longest algo:

This file contains 3 files, each present a different data structure to implement the longest String Matching algorithm (Same data used):

1. Longest_List: *(.py)*

in the code of the longest list, first, I created a data frame to read the data from a file, then I created a code to join all the elements to the row, then I defined two variables to read the lengths and defined an array to store the dp values, then I applied the following steps build L[m+1][n+1] in bottom up fashion.Thus, the letters will be stored one by one, in the event that the next letter in x and y is the same, meaning the current letter is part of the longest, but if it is not, it will search for the largest number of the two, then it will move to the direction of the largest value.

- **README(cont.) :**

3. Longest algo (cont.):

2. Longest_Linkedlist: (.py)

In the longest LinkedList code, also, I created a data frame to read data from a file, then I created code to join all the elements to the row
Next, create a node class, define the node, and enter the data
In linked list class contents a Nood object
I also added a function to initialize the header, so that it prints the contents of the list starting from the header, after that I created a method to return the length of the list, and I also created a method to return the longest string between the two strings using LinkedList

3. Longest_Hashtable: (.py)

In the longest HashTable code, also, I created a data frame to read data from a file, then I created code to join all the elements to the row.
First, I created a table to store the results of the sub-problems, then it would store the lcs string from bottom to top, and it would find the length of the LCs. Then, create a character array to store the lcs string
If the current letter in X and Y is the same, then the current letter is part of the LCS, and if it is not, search for the larger number of the two and go to the direction of the larger value.

Python Codes

- Screenshots was taken from each python code has been written shown below.

Longest List code:

```
# -*- coding: utf-8 -*-
"""
Created on Sat Feb  4 10:13:00 2023

@author: shade
"""
#importing pandas library
import pandas as pd
import time


df = pd.read_csv("both_covid_data.csv")
df = df.astype(str)


df["joined_row"] = df.apply(lambda row: ''.join(row), axis=1)



start_time = time.perf_counter()

def lcs(X, Y):

    m = len(X)
    n = len(Y)


    L = [[None]*(n+1) for i in range(m+1)]


    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0 :
                L[i][j] = 0
            elif X[i-1] == Y[j-1]:
                L[i][j] = L[i-1][j-1]+1
            else:
                L[i][j] = max(L[i-1][j], L[i][j-1])



    index = L[m][n]

```

```

    index = L[m][n]


    lcs=[""] * (index+1)

    lcs[index] = ""


    i=m; j=n;

    while i > 0 and j > 0:


        if X[i - 1] == Y[j - 1]:

            lcs[index - 1]=X[i - 1];

            i -= 1; j -= 1; index -= 1;



        elif L [i - 1][j] > L [i][j - 1]:

            i -= 1;

        else:                j -= 1;

    print ("LCS of " + X + " and " + Y + " is " + "".join(lcs))

end_time = time.perf_counter()


print("Time taken for implementing (Longest String Matching algorithm) using Array:",end_time-start_time)


```

Longest HashTable code:

```python
# -*- coding: utf-8 -*-
"""
Created on Sat Feb  4 20:33:26 2023

@author: shade
"""

import pandas as pd
import time

df = pd.read_csv("both_covid_data.csv")
df = df.astype(str)


df["joined_row"] = df.apply(lambda row: ''.join(row), axis=1)


start_time = time.perf_counter()


def longestCommonSubsequence(str1, str2):
    m, n = len(str1), len(str2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if str1[i - 1] == str2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
```

```python
    index = dp[m][n]
    lcs = [""] * index

    i, j = m, n
    while i > 0 and j > 0:
        if str1[i - 1] == str2[j - 1]:
            lcs[index - 1] = str1[i - 1]
            i -= 1
            j -= 1
            index -= 1
        elif dp[i - 1][j] > dp[i][j - 1]:
            i -= 1
        else:
            j -= 1

    return "".join(lcs)

end_time = time.perf_counter()

print("Time taken for implementing (Longest String Matching algorithm) using Hash Table:",end_time-start_time)
```

Longest linked list code:

```
# -*- coding: utf-8 -*-
"""
Created on Sun Feb  5 12:34:29 2023

@author: shade
"""

# -*- coding: utf-8 -*-
"""
Created on Wed Feb  1 06:56:04 2023

@author: shade
"""
import pandas as pd
import time


df = pd.read_csv("both_covid_data.csv")
df = df.astype(str)


df["joined_row"] = df.apply(lambda row: ''.join(row), axis=1)



start_time = time.perf_counter()

class Node:


    def __init__(self, data):
        self.data = data
        self.next = None


class LinkedList:

    def __init__(self):
        self.head = None

    def printList(self):

        temp = self.head
```

```

        while (temp):

            print (temp.data)

            temp = temp.next

    def getLength(self):

        temp = self.head

        count = 0;

        while (temp):

            count += 1;

            temp = temp.next;

        return count;

    def lcs(self, X, Y, m, n):

        if m == 0 or n == 0:

            return 0;


        elif X[m-1] == Y[n-1]:

            node = Node(X[m-1])

            if self.head is None:

                self.head = node

            else:

                current_node = self.head

                while current_node.next is not None:
```


```
                current_node = self.head

                while current_node.next is not None:
```


```
                    current_node=current_node.next
                    current_node=current_node.next
                    current_node.next=node
                    return 1 + self.lcs(X, Y, m-1, n-1);
                else:

                    return max(self.lcs(X, Y, m, n-1), self.lcs(X, Y, m-1, n));





end_time = time.perf_counter()

print("Time taken for implementing (Longest String Matching algorithm) using Linked List:",end_time-start_time)
```

KMP linked list code:

```python
# -*- coding: utf-8 -*-
"""
Created on Wed Jan 11 07:24:49 2023

@author: ruba balubaid
"""

"""
Definition for variables and methods in code:

- myFile: is a file to read
- text: array to store information from a file
- readlist(): to read a txt file
- KMP(text, pattern): to take array and pattren
- n,m: to calculate length
- lps: will hold the longest prefix suffix values for pattern
"""

# linked list data strcture code
import time
import pandas as pd;

start_time = time.perf_counter()

"""
def readlist():
    #opening the file in read mode
    myFile = open("both_covid_data.txt","r")
    text = []
    for i in myFile:
        text.append(str(i))
    myFile.close()
    return text

text = readlist()
"""
# Python program for KMP Algorithm using Linked List

# A linked list node
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

def KMPSearch(pat, txt):
    M = len(pat)
    N = len(txt)

    # create linked list of pattern characters
    head1 = Node(None) # dummy node to mark beginning
    prev = head1

    for i in range(M):

        # create a new node for every character of pattern
        curr = Node(pat[i])

        # append the new node at the end of linked list
        prev.next = curr

        # update previous pointer to current node
        prev = curr

    # create linked list of text characters
    head2 = Node(None) # dummy node to mark beginning
    prev = head2

    for i in range(N):

        # create a new node for every character of text
        curr = Node(txt[i])

        # append the new node at the end of linked list
        prev.next = curr

        # update previous pointer to current node
        prev = curr

    i, j, count= 0, 0, 0

    while i < N:
        # if characters match then move ahead in both lists
        if (head2.data == head1.data):
            count += 1
            i += 1
            j += 1
            head2=head2.next
            head1=head1.next
        else:
            if (j != 0):
                j=0
            else:
                i+=1
                head2=head2.next
        if (count == M):
            return True
        return False
    count += 1
```

```python
def read_csv():
    # Read the CSV file into a DataFrame
    df = pd.read_csv("both_covid_data.csv")
    df = df.astype(str)
    #print(len(df))
    # Join all the elements in each row with no separator
    df["joined_row"] = df.apply(lambda row: ''.join(row), axis=1)

    # Define the target string
    target = "0010"

    # Initialize a variable to store the result
    result = -1

    # Iterate through the joined_row column of the DataFrame
    for i, s in enumerate(df["joined_row"]):

        col=KMPSearch(s,target)
        if col!=-1:
        # If a match is found, store the index in the result variable
            result = i
            break

    # Print the result
    if result != -1:
        print("Match found at index", result," at col",col)
    else:
        print("No match found.")

read_csv()


end_time = time.perf_counter()

print("Time taken in KMP Linked List:", end_time - start_time)
print()
```

KMP list code:

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Jan 10 21:30:34 2023

@author: ruba balubaid
"""


"""
Definition for variables and methods in code:

- myFile: is a file to read
- text: array to store information from a file
- readlist(): to read a txt file
- KMP(text, pattern): to take array and pattren
- n,m: to calculate length
- lps: longest proper prefix will hold the longest prefix suffix values for pattern
- read_csv(): to convert csv file to data frame and read it

"""

# we use array data strcture in KMP algorithm

import time;
import pandas as pd;


start_time = time.perf_counter()

# Code to be timed


"""
def readlist():
    #opening the file in read mode
    myFile = open("both_covid_data.txt","r")
    text = []
    for i in myFile:
        text.append(str(i))
    myFile.close()
    return text

text = readlist()

"""
```

```python
def KMP(pat, txt):
    M = len(pat)
    N = len(txt)

    # create lps[] that will hold the longest prefix suffix
    # values for pattern
    lps = [0]*M
    j = 0  # index for pat[]

    # Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps)

    i = 0  # index for txt[]
    while (N - i) >= (M - j):
        if pat[j] == txt[i]:
            i += 1
            j += 1

        if j == M:
            print("Found pattern at index " + str(i-j))
            j = lps[j-1]

        # mismatch after j matches
        elif i < N and pat[j] != txt[i]:
            # Do not match lps[0..lps[j-1]] characters,
            # they will match anyway
            if j != 0:
                j = lps[j-1]
            else:
                i += 1


def computeLPSArray(pat, M, lps):
    len = 0  # length of the previous longest prefix suffix

    lps[0] = 0 # lps[0] is always 0
    i = 1

    # the loop calculates lps[i] for i = 1 to M-1
    while i < M:
        if pat[i] == pat[len]:
            len += 1
            lps[i] = len
```

```python
            i += 1
        else:
            # This is tricky. Consider the example.
            # AAACAAAA and i = 7. The idea is similar
            # to search step.
            if len != 0:
                len = lps[len-1]

                # Also, note that we do not increment i here
            else:
                lps[i] = 0
                i += 1


def read_csv():
    # Read the CSV file into a DataFrame
    df = pd.read_csv("both_covid_data.csv")
    df = df.astype(str)
    #print(len(df))
    # Join all the elements in each row with no separator
    df["joined_row"] = df.apply(lambda row: ''.join(row), axis=1)

    # Define the target string
    target = "0010"

    # Initialize a variable to store the result
    result = -1

    # Iterate through the joined_row column of the DataFrame
    for i, s in enumerate(df["joined_row"]):

        col=KMP(target, s)
        if col!=-1:
        # If a match is found, store the index in the result variable
            result = i
            break

    # Print the result
    if result != -1:
        print("Match found at index", result," at col",col)
    else:
        print("No match found.")

read_csv()


end_time = time.perf_counter()

print("Time taken in KMP list:", end_time - start_time)
```

KMP Hash table code:

```python
# -*- coding: utf-8 -*-
"""
Created on Sat Jan 28 18:24:23 2023

@author: ruba balubaid
"""


import time
import pandas as pd;

#start = time.process_time_ns()
start_time = time.perf_counter()


"""
def readlist():
    #opening the file in read mode
    myFile = open("both_covid_data.txt","r")
    text = []
    for i in myFile:
        text.append(str(i))
    myFile.close()
    return text

text = readlist()
  """


def KMP(text, pattern):
    n = len(text)
    m = len(pattern)

    # create a hash table to store the index of pattern
    hash_table = [0] * 256

    # fill the hash table
    for i in range(m):
        hash_table[ord(pattern[i])] = i + 1

    # initialize lps array and variables
    lps = [0] * m
    j, i = 0, 0

    while i < n:

    # if characters match, increment both pointers
      if text[i] == pattern[j]:
          i += 1
          j += 1

        # if j is equal to length of pattern, we have found a match.
      if j == m:
          print("Pattern found at index " + str(i-j))

          # reset j to lps value
          j = lps[j-1]

        # mismatch after j matches
      elif i < n and text[i] != pattern[j]:

          # Do not match lps[0..lps[j-1]] characters, they will match anyway
          if j != 0:
              j = lps[j-1]

          else:
              i += 1

      return -1


def read_csv():
    # Read the CSV file into a DataFrame
    df = pd.read_csv("both_covid_data.csv")
    df = df.astype(str)
    #print(len(df))
    # Join all the elements in each row with no separator
    df["joined_row"] = df.apply(lambda row: ''.join(row), axis=1)

    # Define the target string
    target = "0010"

    # Initialize a variable to store the result
    result = -1

    # Iterate through the joined_row column of the DataFrame
    for i, s in enumerate(df["joined_row"]):

        col=KMP(s,target)
        if col!=-1:
        # If a match is found, store the index in the result variable
            result = i
            break

    # Print the result
    if result != -1:
        print("Match found at index", result," at col",col)
    else:
        print("No match found.")

read_csv()



end_time = time.perf_counter()

print("Time taken in KMP hash table:", end_time - start_time)
```

Naive list code:

```python
# -*- coding: utf-8 -*-
"""
Created on Mon Jan 30 07:32:07 2023

@author: itzha
"""

#importing the necessary libraries
import time
import pandas as pd

#creating a dataframe that read data from the given file.
df = pd.read_csv("both_covid_data.csv")
df = df.astype(str)
# Join all the elements in each row with no separator.
df["joined_row"] = df.apply(lambda row: ''.join(row), axis=1)



 #Naive string matching algorithm to find patterns in data stored in a frame.
start_time = time.perf_counter()
def naive_string_match(dataframe, pattern):

    # Initialize the matches list
    matches = []
```

```python
    # Iterate over each row of the dataframe
    for index, row in dataframe.iterrows():

        # Iterate over each column of the row
        for col in row:

            # If pattern is found in the column value add it to matches list.
            if pattern in str(col):

                matches.append(index)

    return matches

#Pattern:
pattern = '0010'

matches = naive_string_match(df, pattern)
print("Matches found at positions:", matches)    #Print the matches indices.

end_time = time.perf_counter()

#Calculating and printing the run time needed to search for a pattern match.
print("Time taken for implementing (Naive String Matching algorithm) using Array: ", end_time - start_time)
```

Naive linked list code:

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Jan 31 22:27:54 2023

@author: itzha
"""

import pandas as pd
import time

#Pattern :
pattern = '0010'

#Class to create nodes that hold the data element, and a pointer to the next node.
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

# Linked List class contains a node object, to create a linked list.
class LinkedList:
    def __init__(self):
        self.head = None

    # method to add new node to the linked list
    def append(self, new_data):

        # create a new node using the given data and assign it to the head of the linked list if it is empty.
        if self.head is None:
            self.head = Node(new_data)

        # else, traverse till the last node and insert the new_node there.
        else:

            last = self.head

            while (last.next):

                last = last.next

            last.next = Node(new_data)
```

```python
    # method to print all nodes of the linked list
    def printList(self):

        temp = self.head

        while (temp):

            print (temp.data)

            temp = temp.next
    #method to traverse data element in each node.
    def extract(self):

        temp = self.head

        while (temp):
  #each data element in the linked list will be send as an argument to the naive string match method, to search for a pattern match.
           naive_string_match(pattern, temp.data)

           #next element..
           temp = temp.next

start_time = time.perf_counter()

   # method to apply Naive String Matching algorithm on the data stored in Linked List
def naive_string_match(pat, txt):
       M = len(pat)
       N = len(txt)

    #A loop to slide pat[] one by one.
       for i in range(N - M + 1):
         j = 0

        # For current index i, check for pattern match.
         while(j < M):
            if (txt[i + j] != pat[j]):
                break
            j += 1

            if (j == M):
             print("Pattern found at index ", i)  #Print the 1st index in which a pattern match has been found.
```

```python
mylist=LinkedList()

#creating a dataframe that read data from the given file.
df = pd.read_csv("both_covid_data.csv")
df = df.astype(str)
#Join all the elements in each row with no separator
df["joined_row"] = df.apply(lambda row: ''.join(row), axis=1)
for i,s in enumerate(df["joined_row"]):
   #Store the data in the linked list.
   mylist.append(s)


#search for a pattern match for each linked list element.
mylist.extract()
end_time = time.perf_counter()

#Calculating and printing the run time needed to search for a pattern match.
print("Time taken for implementing (Naive String Matching algorithm) using Linked List:", end_time - start_time)
```

Naive hash table code:

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Jan 31 22:45:24 2023

@author: itzha
"""

#importing pandas library
import pandas as pd
import time

#creating a dataframe that read data from the given file.
df = pd.read_csv("both_covid_data.csv")
df = df.astype(str)
#Join all the elements in each row with no separator
df["joined_row"] = df.apply(lambda row: ''.join(row), axis=1)



#creating a hash table, and storing the data from the dataframe in.
hash_table = {df.index[i] : df.joined_row[i] for i in range(len(df))}



#creatin a hash table (for the pattern.)
table = {}

#method to create a hash table of all the substrings of the pattern.
def createHashTable(pattern):
    m = len(pattern)
    for i in range(m):
        table[pattern[i:i+1]] = i


start_time = time.perf_counter()

#method to search for all occurrences of the pattern in the given text (data in the hash table) using Naive String Matching Algorithm.
def searchPattern(text, pattern):
    n = len(text)
    m = len(pattern)
```

```python

    #Create a hash table for pattern
    createHashTable(pattern)

    # Traverse through the text (data from the hash table)
    for i in range(n-m+1):

        # For current index i, check for pattern match
        j = 0;

        while j < m:

            # If characters don't match, break the loop and move to next index in text.
            if text[i + j] != pattern[j]:

                break;

            j += 1;

            # If *all* characters are matched, print the index at which it is found.
            if j == m:

                print("Pattern found at index", i);



#Pattern:
pattern = '0010'

#A loop to traverse the data elements in the hash table, and search in each for a pattern matching.
for key, value in hash_table.items():
    searchPattern(value, pattern)



end_time = time.perf_counter()

#Calculating and printing the run time needed to search for a pattern match.
print("Time taken for implementing (Naive String Matching algorithm) using Hash Table:", end_time - start_time)
```

• *Sample Output:*

```
Pattern found at index 1
Pattern found at index 5
Pattern found at index 6
Pattern found at index 2
Pattern found at index 5
Pattern found at index 3
Pattern found at index 1
Pattern found at index 2
Pattern found at index 1
Pattern found at index 5
Pattern found at index 5
Pattern found at index 2
Pattern found at index 5
Pattern found at index 6
Pattern found at index 2
Pattern found at index 3
Pattern found at index 2
Pattern found at index 5
Pattern found at index 2
Pattern found at index 5
Pattern found at index 1
Pattern found at index 1
Time taken for implementing (Naive String Matching
algorithm) using Hash Table: 0.008225399999901128
```

This screenshot is a part of the output from running (Naive Algo – Hash Table) python file.

as shown, the output will be the index in which a pattern match has been found, and that's for every data element (column by column) in the hash table.

Results & Analysis

- Comparison and and analysis for the overall performances for each algorithm as well as the data structures, shown below.

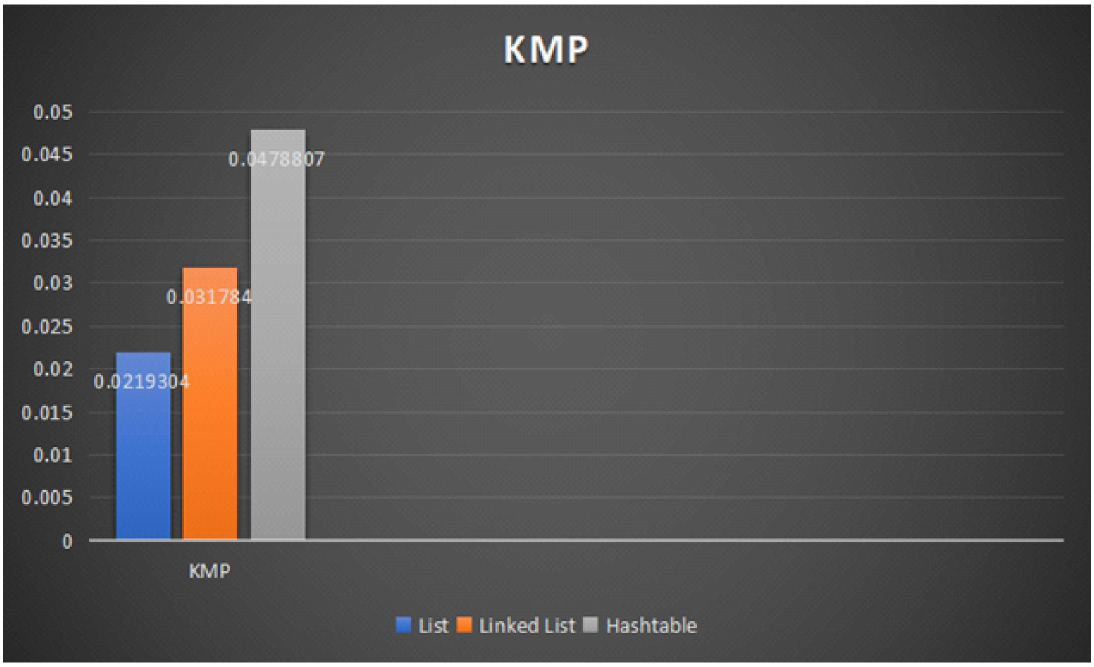
- **Device specifications:**

  MacBook Pro 2020.
  - M1 chip memory
  - 16 gigabyte storage
  - SSD disk 256 gigabyte
  - Mac OS operating system

- **Data size:**

Data from both_covid_data.cvs file, its size equals *2647*.

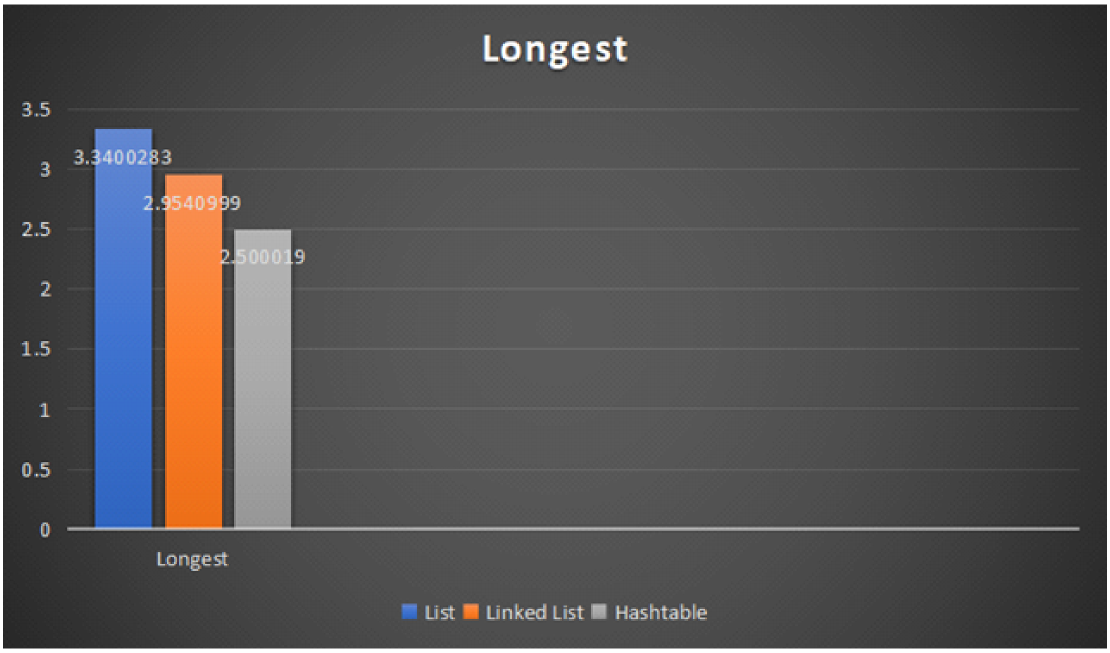
- ## **Reuslts and analysis:**



List is the fastest in KMP because it took less time in the implementation process 0.0219304. and the list faster than the linked list of data structures because they are more efficient at searching for patterns, List data structures use an array of elements that can be searched in a linear fashion, while linked list data structures require traversing the entire list to find a pattern,The linear search of the list data structure is much faster than the traversal of the linked list structure.

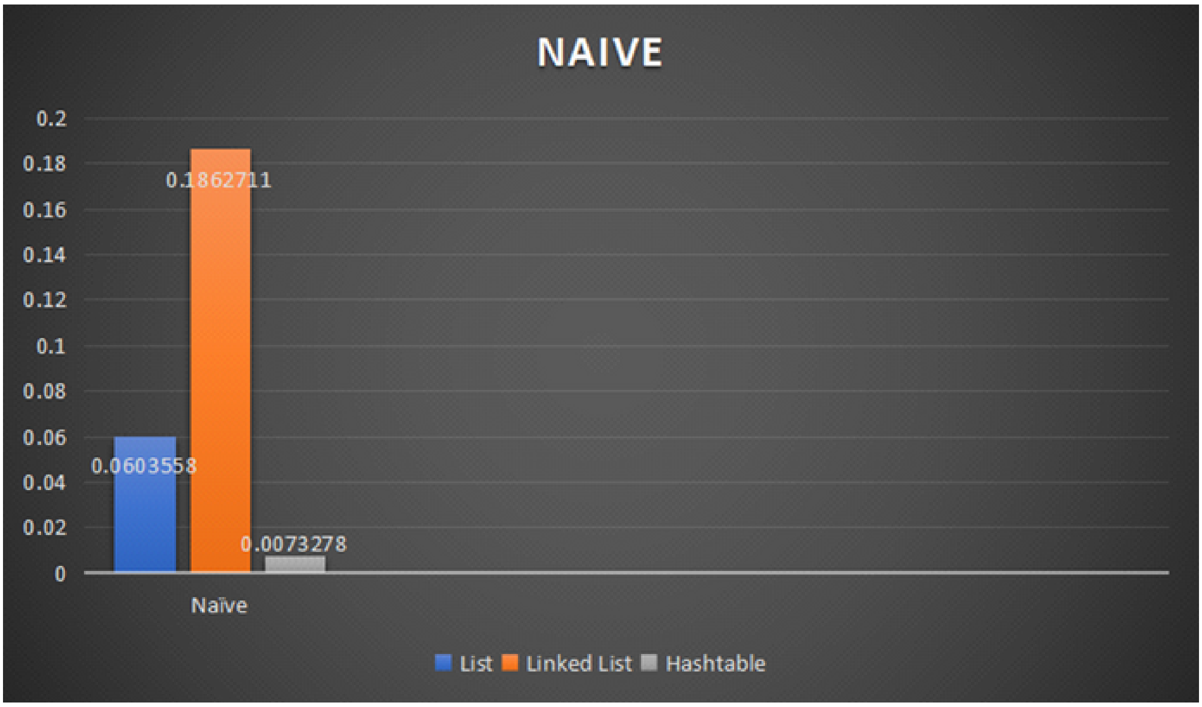
and faster than hash table because they can quickly traverse through the list of characters in the pattern to find a match,Hash tables require more time to search for a particular character in the pattern, as they must hash each character and then compare it to the stored value. List data structures can quickly traverse through the list of characters and find a match, making them faster than hash tables for KMP string matching.

- **Reuslts and analysis (cont.):**



hash table fastest in longest because it took less time in the implementation process 2.500019 .Hash tables are faster than lists because they use a hashing algorithm to quickly locate the desired string. This means that instead of having to search through an entire list, the hash table can quickly locate the desired string by using its hash value. This makes it much faster than searching through a list, which requires linear time complexity, and faster than linked list because they allow for constant time lookups. This means that when searching for a string, the algorithm can quickly determine if the string is present in the hash table without having to traverse through a linked list. Additionally, hash tables can store more data than linked lists, which allows for more efficient storage and retrieval of data.

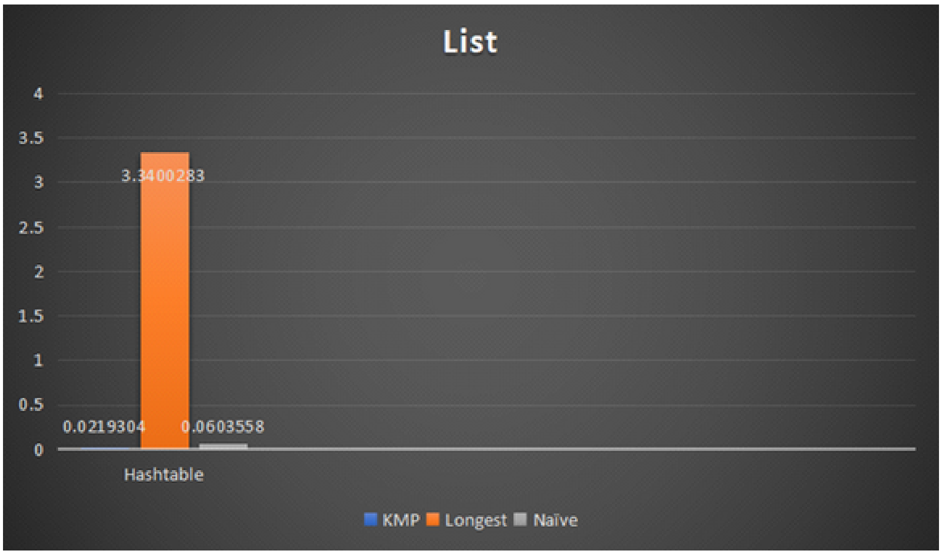
- **Reuslts and analysis (cont.):**



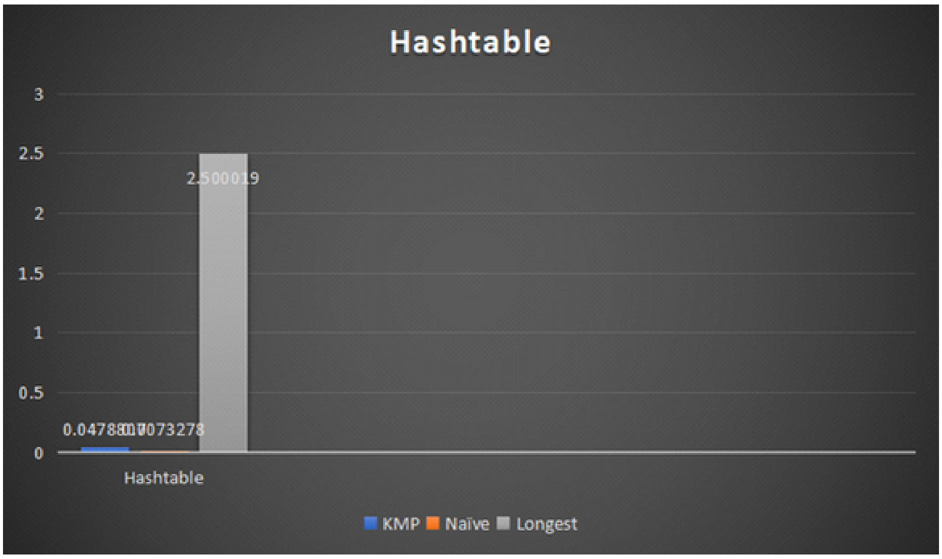
hash table fastest in naive because it took less time in the implementation process 0.0073278.Hash tables are faster than linked lists in naive string matching algorithms because they allow for faster lookups. Hash tables use a hashing function to map keys to values, which makes it easier and faster to find the desired value. Linked lists, on the other hand, require linear search time, which can be slow if the list is long , and faster than lists in naive string matching algorithms because they allow for faster lookups. Hash tables use a hashing function to map a key to an index in the table, which allows for constant time lookup of the value associated with the key. Lists, on the other hand, require linear time lookup since each element must be searched through sequentially. This makes hash tables much more efficient when searching for a specific value.

- ## Reuslts and analysis (cont.):

(comparing between the same data structure used to implement each algorithm)

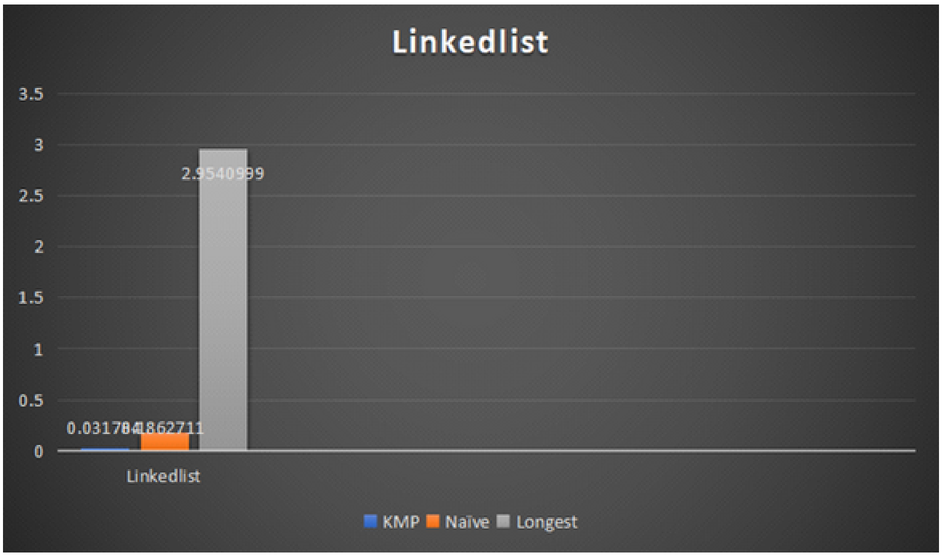


KMP is faster in the list data structure because it takes 0.0219304 for execute and The time complexity of the KMP string matching algorithm using a List is O(n). This is because the algorithm only needs to iterate through the list once to find a match, and it does not need to perform any additional operations.



naive is faster in the hash table data structure because it takes 0.0073278 for execute and The time complexity of a naive string matching algorithm using a hash table is O(n). This is because the algorithm must iterate through each character in the string to generate a hash value, and then compare the generated hash value with the stored hash values in the table.

- **Reuslts and analysis (cont.):**
  (comparing between the same data structure used to implement each algorithm)



KMP is faster in the linked list data structure because it takes0.031784 for execute and the time complexity of the KMP string matching algorithm using a linked list is O(n+m), where n is the length of the string and m is the length of the pattern.

• Group Work Report :

	Ruba Balubaid	Hadeel Alnasiri	Yara Al similan	Noor Alhashmi	Shaden Anagreh
Naive codes (3 data structures)		✔			
KMP codes (3 data structures)	✔				
LCS codes (3 data structures)					✔
Comparisons			✔		
Project Report	✔	✔		✔	
Project presentation	✔				