**DAA ASSIGNMENT**

**Name: Palak Agarwal**

**Section: F**

**University No: 2014755**

Q1: Explain Naive Method of String Matching.

Ans:

**Algorithm**

NAIVE-STRING-MATCHER (T, P)  
1. n ← length [T]  
2. m ← length [P]  
3. for s ← 0 to n -m  
4. do if P [1.....m] = T [s + 1....s + m]  
5. then print "Pattern occurs with shift" s

**Analysis:**

This for loop from 3 to 5 executes for n-m + 1(we need at least m characters at the end) times and in iteration we are doing m comparisons. So the total complexity is O (n-m+1).

**Code**

#include <stdio.h>

#include <string.h>

void search(char\* pat, char\* txt)

{

    int M = strlen(pat);

    int N = strlen(txt);

    /\* A loop to slide pat[] one by one \*/

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

        int j;

        /\* For current index i, check for pattern match \*/

        for (j = 0; j < M; j++)

            if (txt[i + j] != pat[j])

                break;

        if (j == M) // if pat[0...M-1] = txt[i, i+1, ...i+M-1]

            printf("Pattern found at index %d \n", i);   } }

void main() {

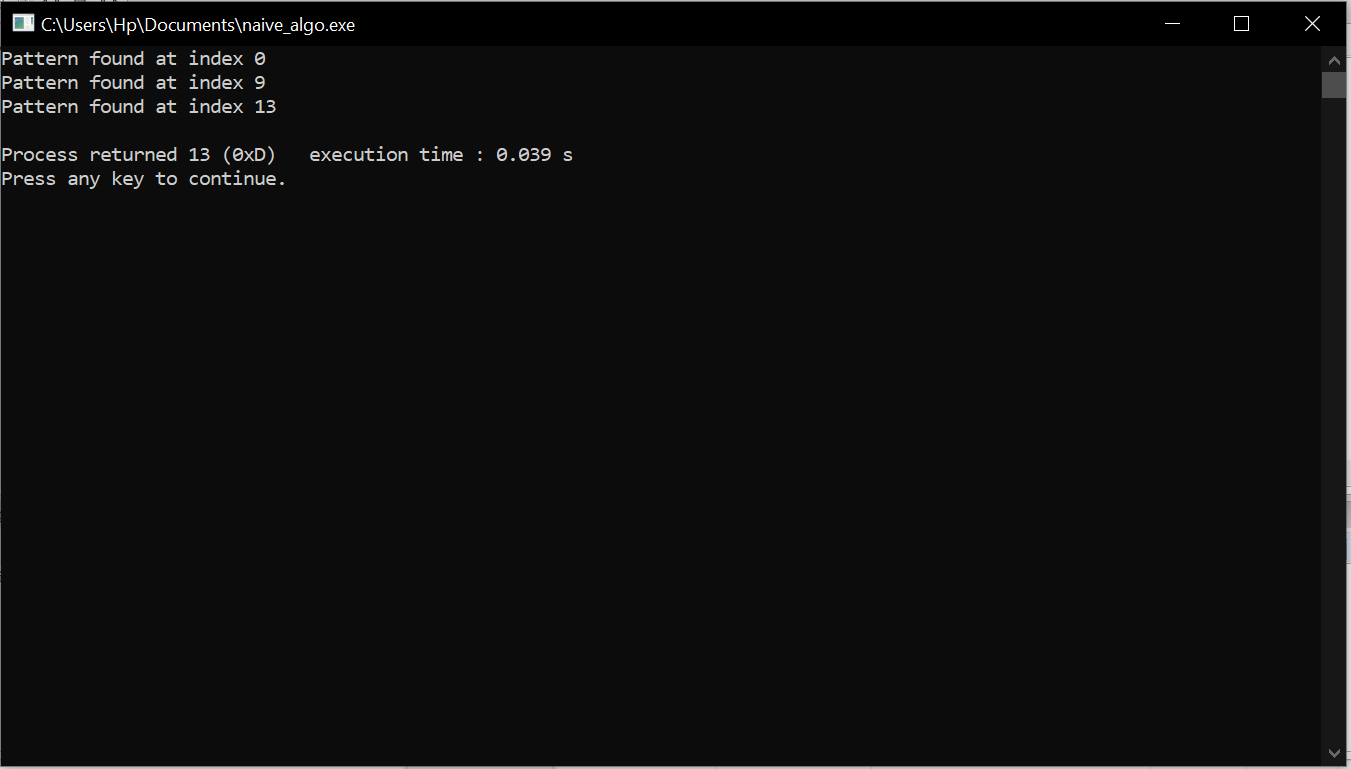
    char txt[] = "AABAACAADAABAAABAA";

    char pat[] = "AABAA";

    search(pat, txt);

}

**Output:**



**Q2.: Explain Robin Karp Algorithm.**

**Ans:** The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character sub sequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for next M-character sequence.

Algorithm:

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

Time Complexity:

The average and best-case running time of the Rabin-Karp algorithm is O(n+m), but its worst-case time is O(nm).

**Code:**

#include<stdio.h>

#include<string.h>

#define d 256

void search(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)

printf("Pattern found at index %d \n", i);

}

if ( i < N-M )

{

t = (d\*(t - txt[i]\*h) + txt[i+M])%q;

if (t < 0)

t = (t + q);

}

}

}

int main()

{

char txt[] = "GRAPHIC ERA";

char pat[] = "ERA";

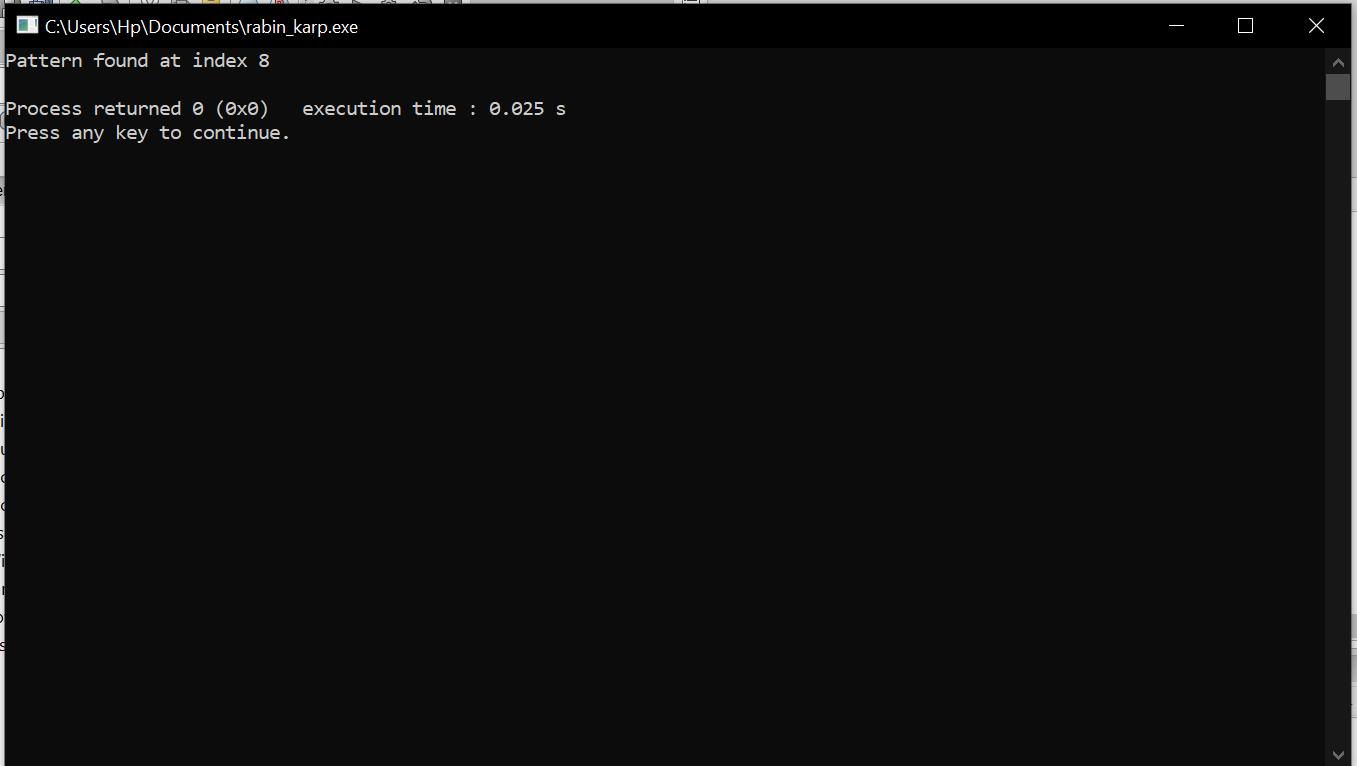
int q = 101;

search(pat, txt, q);

return 0;

}

**Output:**



**Q3: Explain Knuth Morris Pratt Algorihtm.**

**Ans:** **Algorithm**

Start

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

Stop

**Code:**

#include<stdioi.h>

#include<string.h>

void prefixSuffixArray(char\* pat, int M, int\* pps) {

int length = 0;

pps[0] = 0;

int i = 1;

while (i < M) {

if (pat[i] == pat[length]) {

length++;

pps[i] = length;

i++;

} else {

if (length != 0)

length = pps[length - 1];

else {

pps[i] = 0;

i++;

}

}

}

}

void KMPAlgorithm(char\* text, char\* pattern) {

int M = strlen(pattern);

int N = strlen(text);

int pps[M];

prefixSuffixArray(pattern, M, pps);

int i = 0;

int j = 0;

while (i < N) {

if (pattern[j] == text[i]) {

j++;

i++;

}

if (j == M) {

printf("Found pattern at index %d\n", i - j);

j = pps[j - 1];

}

else if (i < N && pattern[j] != text[i]) {

if (j != 0)

j = pps[j - 1];

else

i = i + 1;

}

}

}

int main() {

char text[] = "xyztrwqxyzfg";

char pattern[] = "xyz";

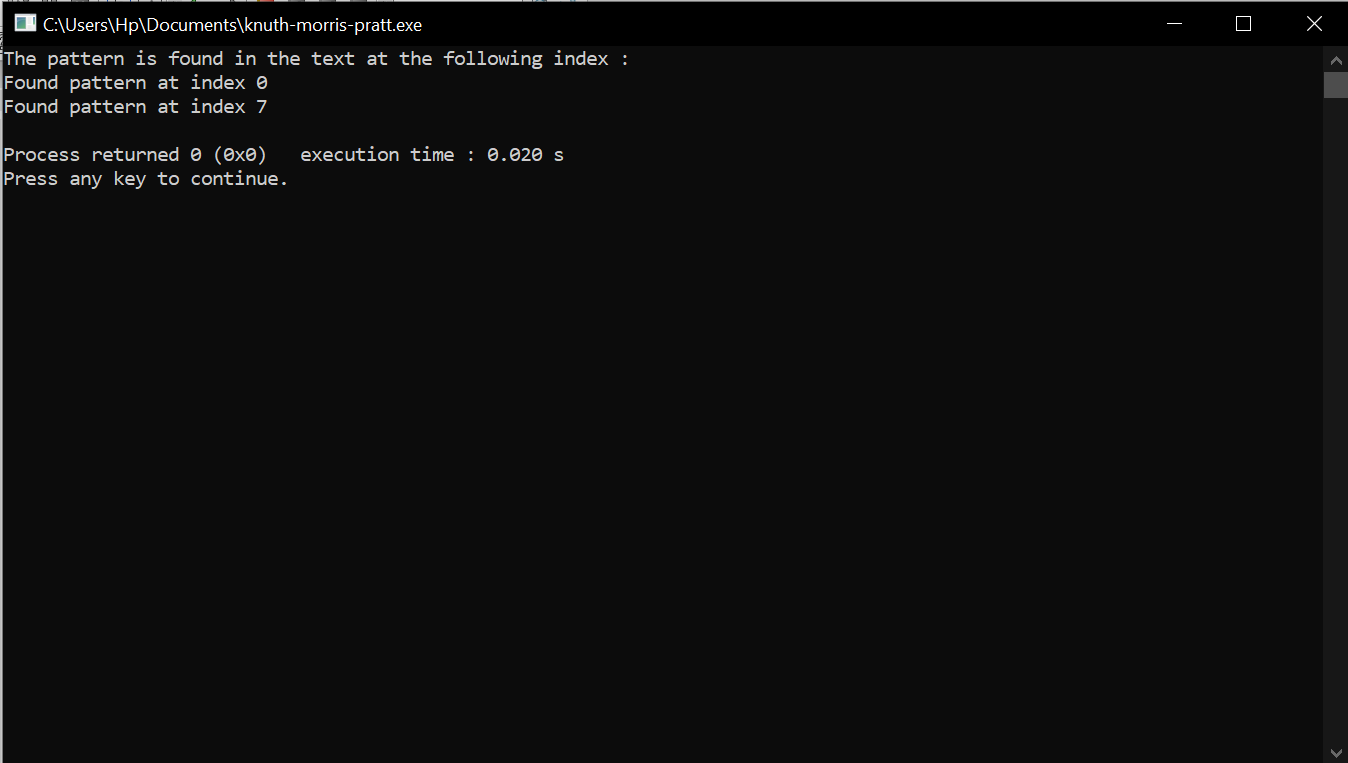
printf("The pattern is found in the text at the following index : \n");

KMPAlgorithm(text, pattern);

return 0;

}

**Output:**

****