**CHAROTAR UNIVERSITY OF SCIENCE &**

**TECHNOLOGY**

**DEVANG PATEL INSTITUTE OF ADVANCE TECHNOLOGY & RESEARCH**

**Computer Science & Engineering**

**NAME: PARTH NITESHKUMAR PATEL**

**ID: 19DCS098**

**SUBJECT: DESIGN AND ANALYSIS OF**

**ALGORITHM**

**CODE: CS 351**

**STRING MATCHING ALGORITHM**

**PRACTICAL-8.1**

**AIM:**

Suppose you are given a source string S[0 ..n − 1] of length n, consisting of symbols a and b. Suppose that you are given a pattern string P[0 ..m − 1] of length m < n, consisting of symbols a, b, and \*, representing a pattern to be found in string S. The symbol \* is a “wild card” symbol, which matches a single symbol, either a or b. The other symbols must match exactly. The problem is to output a sorted list M of valid “match positions”, which are positions j in S such that pattern P matches the substring S [j..j + |P|− 1].

For example, if S = ababbab and P = ab\*, then the output M should be [0, 2]. Implement a

straightforward, naive algorithm to solve the problem.

**PROGRAM CODE:**

#include<iostream>

#include <string.h>

using namespace std;

int main()

{

    char t[100], p[100];

    int tn, pn, shift[20] = {0}, s = 0, i, j = 0, count = 0, m = 0;

    cout<<"ENTER THE TEXT : ";

    cin>>t;

    cout<<"ENTER THE PATTERN : ";

    cin>>p;

    tn = strlen(t);

    pn = strlen(p);

    while (s != (tn - pn + 1))

    {

        j = 0;

        for (i = s; i < pn + s; i++)

        {

            if (p[j] == t[i])

            {

                count++;

                if (count == pn)

                {

                    count = 0;

                    shift[m] = s;

                    m++;

                }

            }

            else

            {

                count = 0;

                break;

            }

            j++;

        }

        s++;

    }

    if (m > 0)

    {

        printf("\n\nVALID SHIFTS : ");

        for (i = 0; i < m; i++)

            printf("%d \n", shift[i]);

    }

    else

    {

        printf("\n\n NO VALID SHIFTS AVAILABLE");

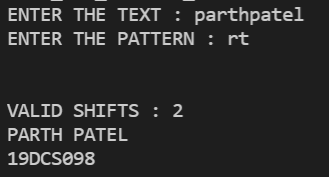
    }

    cout<<"PARTH PATEL\n19DCS098"<<endl;

    return 0;

}

**OUTPUT:**



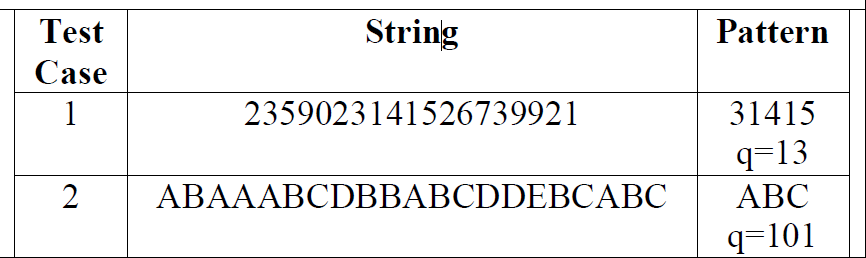
**Conclusion:**

* Naive pattern searching is the simplest method among other pattern searching algorithms.
* It checks for all character of the main string to the pattern.
* This algorithm is helpful for smaller texts.
* It does not need any pre-processing phases.
* The time complexity is **O(m\*n)**. The m is the size of pattern and n is the size of the main string

**PRACTICAL-8.2**

**AIM:**

Implement Rabin karp algorithm and test it on the following test cases:



**PROGRAM CODE:**

#include <iostream>

#include <string.h>

#define d 256

using namespace std;

void search(char pattern[], char text[], int q)

{

    int M = strlen(pattern);

    int N = strlen(text);

    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 + pattern[i]) % q;

        t = (d \* t + text[i]) % q;

    }

    for (i = 0; i <= N - M; i++)

    {

        if (p == t)

        {

            for (j = 0; j < M; j++)

            {

                if (text[i + j] != pattern[j])

                    break;

            }

            if (j == M)

                cout<<"PATTERN FOUND AT INDEX : "<<i<<endl;

        }

        if (i < N - M)

        {

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

            if (t < 0)

                t = (t + q);

        }

    }

}

int main()

{

    char text[100];

    char patterntern[100];

    int q;

    cout<<"ENTER THE TEXT : ";

    cin>>text;

    cout<<"ENTER THE PATTERN : ";

    cin>>patterntern;

    cout<<"ENTER THE VALUE OF q : ";

    cin>>q;

    search(patterntern, text, q);

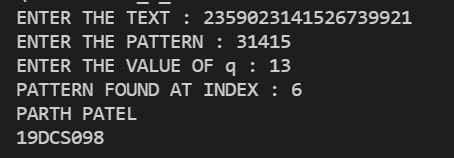
    cout<<"PARTH PATEL\n19DCS098"<<endl;

    return 0;

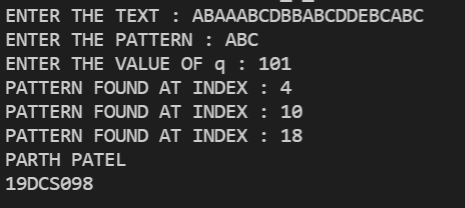
}

**OUTPUT:**

**Test Case-1:**



**Test Case-2:**



**CONCLUSION:**

* For text of length n and p patterns of combined length m, its average and best case time complexity is **O(n+m)** , but its worst-case time complexity is **O(nm)**