Suraj Iyer
2021300045
SE Comps A, Batch C

DAA Experiment 10

Aim – To study String matching algorithm

Details – Text-editing programs frequently need to find all occurrences of a pattern in the text. Typically, the text is a document being edited, and the pattern searched for is a particular word supplied by the user. Efficient algorithms for this problem—called "string matching"—can greatly aid the responsiveness of the text-editing program. Among their many other applications, string-matching algorithms search for particular patterns in DNA sequences. Internet search engines also use them to find Web pages relevant to queries. We formalize the string-matching problem as follows.
We assume that the text is an array T [1 : n] of length n and that the pattern is an array T [1 : m] of length m <=n. We further assume that the elements of P and T are characters drawn from a finite alphabet∑. For example, we may have ∑ = {aa, bb, cc, … , zz} or ∑ = {0,1}. The character arrays P and T are often called strings of characters. Given a text array, T [1.....n], of n character and a pattern array, P [1......m], of m characters.

The problems are to find an integer s, called a valid shift where 0 ≤ s < n-m and T [s+1......s+m] = P [1......m]. In other words, to find even if P in T, i.e., where P is a substring of T. The items of P and T are characters drawn from some finite alphabet such as {0, 1} or {A, B .....Z, a, b..... z}. Given a string T [1......n], the substrings are represented as T [i......j] for some 0≤i ≤ j≤n-1, the string formed by the characters in T from index i to index j, inclusive. This process that a string is a substring of itself (take i = 0 and j =m). The proper substring of string T [1......n] is T [1......j] for some 00 or j < m-1.

There are different strings matching algorithms. Each string-matching algorithm performs some preprocessing based on the pattern and then finds all valid shifts; we call this latter phase "matching." Following figure shows the preprocessing and matching times for each of the algorithms.

Code –

```
#include <stdio.h>
#include <string.h>
#include <math.h>

#define d 256

void rabinkarp(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)
```

```c
                t = (t + q);
            }
    }
}

int search(char p[30],char t[30], int i)
{
    int f=0;
    for(int k=0;k<strlen(p);k++)
    {
        if(p[k]!=t[k+i])
        {
            f=1;
            break;
        }
    }
    return f;
}

int anum(char alpha)
{
    for(int k=1;k<=26;k++)
    {
        if(alpha=='a'+k-1)
        return k;
    }
}

int main(void)
{

    char t[30],p[30];
    printf("\nEnter a sentence : ");
    gets(t);
    printf("\nEnter the word to be searched: ");
    gets(p);
    printf("\nT = %s",t);
    printf("\nP = %s",p);
    printf("\n\n");
```

```
    //naive
    printf("\n\nNaive approach : ");
    int x=0;
    for(int k=0;k<strlen(t);k++)
    {
        if(search(p,t,k)==0)
        {
        printf("\nString found from (%d , %ld)",k,k+strlen(p));
        x=1;
        break;
        }
    }
    if(x==0)
    printf("\nString not found!");

    printf("\n\nRabin Karp Algorithm: \n");
    int q=101;

    rabinkarp(p, t, q);

    return 0;

}
```

Output –

```
T = My name is Suraj Iyer
P = Suraj Iyer



Naive approach :
String found from (11 , 21)

Rabin Karp Algorithm:
Pattern found at index 11
PS C:\Users\Suraj>
```

Conclusion – Hence I have implemented various kinds of string algorithm techniques, and have come to conclude that Rabin Karp provides us with a much more efficient way to solve the problem of string matching.