K. K. Wagh Institute of Engineering Education and Research, Nashik.
Department of Computer Engineering Academic Year 2022-23

Course Name: Laboratory Practice – III **Course Code:** 410246

Class: BE **Div:** A

Name of Students: Atharva Anil More (61)
Shreyash Suresh Lanjewar (62)
Siddesh Mukesh Patankar(63)
Ketan Pralhad Patil (64)

Name of Faculty: Prof. C. R. Patil

Mini Project Report

Title of Mini-Project: Implement the Naive string matching algorithm and Rabin-Karp algorithm for string matching. Observe difference in working of both the algorithms for the same input.

Objective:

1. Understand and explore the working both algorithms and its applications.
2. Understand the time and space complexity of both the algorithms.

Theory:

Navies String Matching:

The naïve approach tests all the possible placement of Pattern P [1.......m] relative to text T [1......n]. We try shift s = 0, 1.......n-m, successively and for each shift s. Compare T [s+1.......s+m] to P [1......m].

The naïve algorithm finds all valid shifts using a loop that checks the condition P [1.......m] = T [s+1.......s+m] for each of the n - m +1 possible value of s.

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.
```

The number of comparisons in the worst case is O(M * (N – M + 1)). Naive Complexity.

Analysis: The for loop 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).

Rabin Karp Algorithm:

The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for next M-character sequence. If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match.

Algorithm:

RABIN-KARP-MATCHER (T, P, d, q)

```
1. n ← length [T]
2. m  ← length [P]
3. h  ←  dm-1 mod q
4. p ←  0
5. t0 ←  0
6. for i ← 1 to m
7. do p ←  (dp + P[i]) mod q
8. t0 ← (dt0+T [i]) mod q
9. for s  ←  0 to n-m
10. do if p = ts
```

```
11. then if P [1.....m] = T [s+1.....s + m]
12. then "Pattern occurs with shift" s
13. If s < n-m
14. then ts+1 ← (d (ts-T [s+1]h)+T [s+m+1])mod q
```

Complexity:

The running time of RABIN-KARP-MATCHER in the worst case scenario O ((n-m+1) m but it has a good average case running time. If the expected number of strong shifts is small O (1) and prime q is chosen to be quite large, then the Rabin-Karp algorithm can be expected to run in time O (n+m) plus the time to require to process spurious hits.

Process of String Matching-

1. Coding
2. Compilation – Used EVM

Tool: c++ compiler

Code:

```
// C++ program for Naive Pattern Searching algorithm and Rabin Karp Pattern Searching algorithm
#include <bits/stdc++.h>
using namespace std;

#define d 256

/* pat -> pattern
   txt -> text
   q -> A prime number
*/

void search_RK(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) cout << "Pattern found at index " << i << endl;
        }

        if (i < N - M)
        {
            t = (d * (t - txt[i] * h) + txt[i + M]) % q;

            if (t < 0) t = (t + q);
        }
    }
}

void search_NS(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) cout << "Pattern found at index " << i << endl;
}
}



int main()
{
char txt[] = "AABAACAADAABAAABAA";
char pat[] = "AABA";
```

```
    cout<<"Naive approach: "<<endl;
search_NS(pat, txt);
cout<<endl;
cout<<"Rabin Karp approach: "<<endl;
search_RK(pat, txt, INT_MAX);
return 0;
}
```

Output:

```
Naive approach:
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13

Rabin Karp approach:
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

Conclusion:

Our mini-project is on “Implemented the Naive string matching algorithm and Rabin-Karp algorithm for string matching.” From this mini-project, we understood and observed difference in working of both the algorithms for the same input. We also computed the time complexities of both the algorithms and found that the Rabin Karp algorithm is most efficient algorithm for the string matching.