PUNE INSTITUTE OF COMPUTER TECHNOLOGY,
DHANKAWADI PUNE-43

MINI-PROJECT ON

NAÏVE STRING MATCHING AND RABIN-KARP STRING MATCHING ALGORITHMS

Submitted By

Shreyash Shinkar	**41373**
Sourav Kotkar	**41378**
Shashank Udgirkar	**41384**

Under the guidance of
Prof. Satya Prakash Patel

DEPARTMENT OF COMPUTER ENGINEERING
Academic Year 2022-23

PUNE INSTITUTE OF COMPUTER TECHNOLOGY,
DHANKAWADI PUNE-43

CERTIFICATE

This is to certify that **Shreyash Shinkar (41373), Sourav Kotkar (41378), Shashank Udgirkar (41384)** student of B.E. (Computer Engineering Department) Batch 2022-2023, have satisfactorily completed a report on “**Naïve String Matching and Rabin-Karp String Matching Algorithms**” towards the partial fulfillment of the fourth year Computer Engineering Semester VII of SPPU.

Dr. G. V. Kale
Head of Department,
Computer Engineering

Date:
Place: Pune

CONTENTS

Introduction:

String Matching Algorithm is also called "String Searching Algorithm." This is a vital class of string algorithm that is declared as "this is the method to find a place where one of several strings are found within the larger string."

Algorithms used for String Matching:

There are different types of method is used to finding the string

1. The Naive String-Matching Algorithm
2. The Rabin-Karp-Algorithm
3. Finite Automata
4. The Knuth-Morris-Pratt Algorithm
5. The Boyer-Moore Algorithm

Problem Statement:

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:

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.

Theory:

The “Naive” Method

Its idea is straightforward — for every position in the text, consider it a starting position of the pattern and see if you get a match. The “naive” approach is easy to understand and implement but it can be too slow in some cases. If the length of the text is n and the length of the pattern m, in the worst case it may take as much as (n * m) iterations to complete the task.

Algorithm:

```
function brute_force(text[], pattern[])
{
        // let n be the size of the text and m the size of the
        // pattern
        for (i = 0; i < n; i++)
        {
                for (j = 0; j < m && i + j < n; j++)
                        if (text[i + j] != pattern[j]) break;
                // mismatch found, break the inner loop
                if (j == m) // match found
        }
}
```

Rabin Karp Algorithm

This is the “naive” approach augmented with a powerful programming technique – the hash function. Every string s[] of length m can be seen as a number H written in a positional numeral system in base B (B >= size of the alphabet used in the string):

H = s[0] * B(m – 1) + s[1] * B(m – 2) + … + s[m - 2] * B1 + s[m - 1] * B0 If we calculate the number H (the hash value) for the pattern and the same number for every substring of length m of the text than the inner loop of the “naive” method will disappear – instead of comparing two strings character by character we will have just to compare two integers.

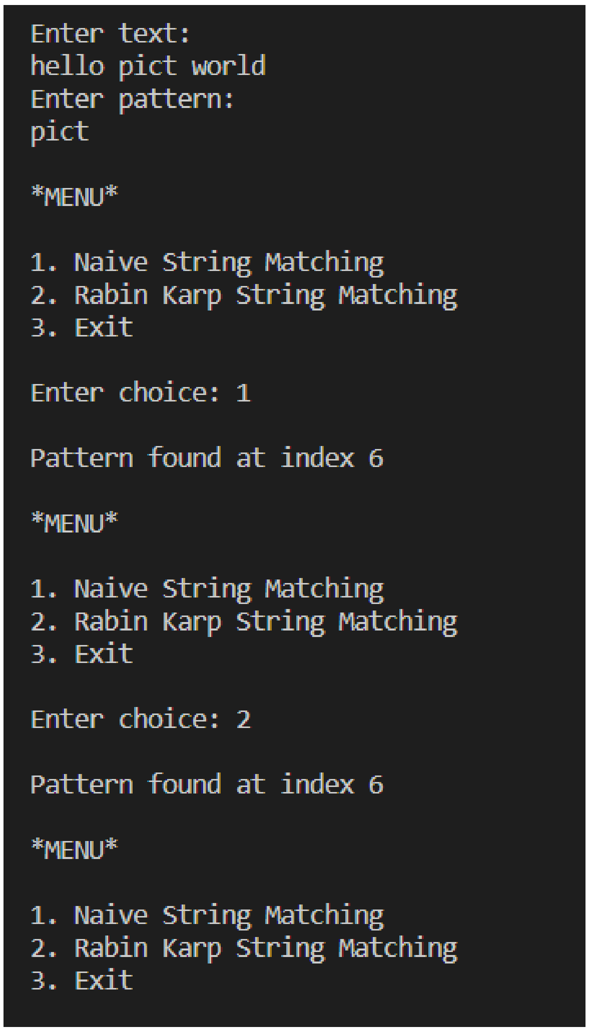
Algorithm:

```
// correctly calculates a mod b even if a < 0
function int_mod(int a, int b)
{
        return (a % b + b) % b;
}

function Rabin_Karp(text[], pattern[])
{
        // let n be the size of the text, m the size of the
        // pattern, B - the base of the numeral system,
        // and M - a big enough prime number
        if (n < m) return; // no match is possible
        // calculate the hash value of the pattern
        hp = 0;
        for (i = 0; i < m; i++)
                hp = int_mod(hp * B + pattern[i], M);
        // calculate the hash value of the first segment of the text of length m
        ht = 0;
        for (i = 0; i < m; i++)
                ht = int_mod(ht * B + text[i], M);
        if (ht == hp) //check character by character if the first segment of the text matches the pattern;
        // start the "rolling hash" - for every next character in
        // the text calculate the hash value of the new segment
        // of length m; E = (Bm-1) modulo M
        for (i = m; i < n; i++)
        {
                ht = int_mod(ht - int_mod(text[i - m] * E, M), M);
                ht = int_mod(ht * B, M);
                ht = int_mod(ht + text[i], M);
                if (ht == hp) check character by character if the current segment of the text matches the pattern;
        }
}
```

Code:

```
#include <bits/stdc++.h>
using namespace std;

#define d 256

void RabinKarpSearch(string pat, string txt, int q)
{
    int M = pat.size();
    int N = txt.size();
    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<<"\nPattern found at index "<<i<<endl;
                return;
            }
        }

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

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

    cout<<"\nPattern not found"<<endl;
}

void NaiveSearch(string pat, string txt)
{
    int M = pat.size();
    int N = txt.size();

    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<<"\nPattern found at index "<<i<<endl;
            return;
        }
    }

    cout<<"\nPattern not found"<<endl;
}

int main()
{
    string txt;
    string pat;
    cout<<"\nEnter text: \n";
    getline(cin, txt);
    cout<<"Enter pattern: \n";
    getline(cin, pat);

    while(true){
        cout<<"\n*MENU*\n"<<endl;
        cout<<"1. Naive String Matching"<<endl;
        cout<<"2. Rabin Karp String Matching"<<endl;
        cout<<"3. Exit"<<endl;

        int c;
        cout<<"\nEnter choice: ";
        cin>>c;
        if(c == 1){
            NaiveSearch(pat, txt);
        }
        else if(c == 2){
            int q = INT_MAX;
            RabinKarpSearch(pat, txt, q);
        }
        else if(c == 3){
            break;
        }
        else{
            cout<<"\nEnter valid choice."<<endl;
        }
    }

    return 0;
}
```

Output:



```
Enter text:
hello pict world
Enter pattern:
pict

*MENU*

1. Naive String Matching
2. Rabin Karp String Matching
3. Exit

Enter choice: 1

Pattern found at index 6

*MENU*

1. Naive String Matching
2. Rabin Karp String Matching
3. Exit

Enter choice: 2

Pattern found at index 6

*MENU*

1. Naive String Matching
2. Rabin Karp String Matching
3. Exit
```

Time Complexity and Performance:

Naive Algorithm

Time Complexity - O(n^2)

Rabin Karp Algorithm

Time Complexity -

Best Case - O(n + m)

Worst Case - O(nm)

Conclusion:

Thus, in this assignment we have studied different algorithms for string matching like Naive method and Rabin Karp algorithm. We also compared the time complexity of these two algorithms and found that Rabin Karp has the best-case time complexity among the two.

References:

https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/

https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/