A Project Report On

Design and Analysis of Algorithms(Mini Project I)

CLASS: BE-4

GUIDED BY

Prof. Deepika Bhaiya

DEPARTMENT OF COMPUTER ENGINEERING PUNE

INSTITUTE OF COMPUTER TECHNOLOGY DHANKAWADI, PUNE-43

SAVITRIBAI PHULE PUNE UNIVERSITY 2022-23

SUBMITTED BY

Burhanuddin Merchant **Roll No: 41439**

Mehul Oswal **Roll No: 41444**

Title:

String Matching Algorithm.

Problem Definition:

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

Theory:

Pattern matching is the process of checking a perceived sequence of strings for the presence of the constituents of some pattern. In contrast to pattern recognition, the match usually has to be exact. The patterns generally have the form sequences of pattern matching including outputting the locations of a pattern within a string sequence, to output some component of the matched pattern and substituting the matching pattern with some other string sequence (i.e., search and replace). Pattern matching concept is used in many applications Following figure shows the different applications.

String matching algorithms are an important class of string algorithms that tries to find one or many indices where one or several strings(or patterns) are found in the larger string(or text).

String matching is used in various applications like **spell checkers**, **spam filters**, **search engines**, **plagiarism detectors**, **bioinformatics, DNA sequencing**, etc.

Algorithms used for pattern searching.

1. Naive Pattern Matching.
2. Rabin Karp String Matching Algorithm.
3. Knuth–Morris–Pratt algorithm.
4. Boyer–Moore string Matching algorithm.

Naive String Matching.

A naive string-matching algorithm compares the given pattern against all positions in the given text. Each comparison takes time proportional to the length of the pattern, and the number of positions is proportional to the length of the text. Therefore, the worst-case time for such a method is proportional to the product of the two lengths. In many practical cases, this time can be significantly reduced by cutting short the comparison at each position as soon as a mismatch is found, but this idea cannot guarantee any speedup.

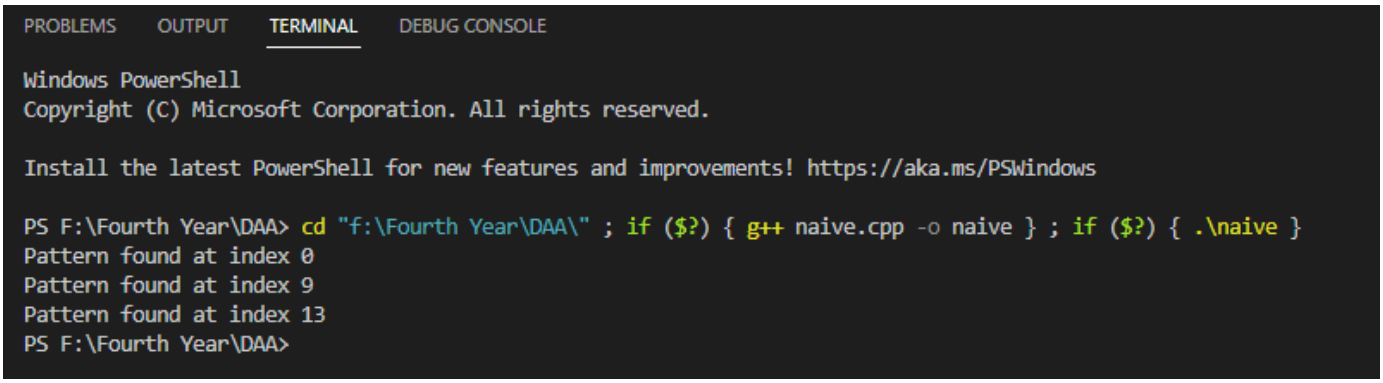
Technique: Each character of the pattern is compared to a substring of the text which is the length of the pattern until there is a mismatch or a match.

Code:

```
#include <bits/stdc++.h>

using namespace std;

int c=0;

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 < n; j++)

            if (txt[i + j] != pat[j])

                break;

        if (j== m) {

            c=1;

            cout << "Pattern found at index " << i << endl;
```

```
        }
    }
}


int main()
{
    string text = "AABAACAADAABAAABAA";
    string pat = "AABA";
   NaiveSearch(pat, text);
   if(!c){
    cout<<"Pattern Not Found"<<endl;
   }
    return 0;
}
```

Output ScreenShot



Time Complexity:

The **best case** occurs when the first character of the pattern is not present in the text at all.

txt[] = "EABCCAADDEE";

pat[] = “ZC”;

The number of comparisons in the best case is O(n).

The **worst case** of Naive Pattern Searching occurs in when all characters of the text and pattern are the same or when only the last character is different.

txt[] = "AAAAAAAAAAAAAAAAA";

pat[] = “AAAAA";

txt[] = "AAAAAAAAAAAAAAAAAB";

pat[] = "AAAAB";

The number of comparisons in the worst case is **O(m*(n-m+1))**. Although strings which have repeated characters are not likely to appear in English text, they may well occur in other applications (for example, in binary texts).

Space Complexity

This is an in-place algorithm. So O(1) auxiliary space is needed.

Rabin Karp String Matching Algorithm.

Rabin-Karp algorithm slides the pattern one by one. But unlike the Naive algorithm, Rabin Karp algorithm matches the hash value of the pattern with the hash value of the current substring of text. If the hash values match then only it starts matching individual characters.

Technique: Hashing! So Rabin Karp algorithm needs to calculate hash values for the following strings.

1) Pattern itself.
2) All the substrings of the text of length m.

Since we need to efficiently calculate hash values for all the substrings of size m of text, we must have a hash function that has the following property.

Hash at the next shift must be efficiently computable from the current hash value and next character in the text or we can say hash(txt[s+1 .. s+m]) must be efficiently computable from a hash(txt[s .. s+m-1]) and txt[s+m] i.e., hash(txt[s+1 .. s+m])= rehash(txt[s+m], hash(txt[s .. s+m-1])) and rehash must be O(1) operation.

To do rehashing, we need to take off the most significant digit and add the new least significant digit in the hash value.

Rehashing is done using the following formula.

hash(txt[s+1 .. s+m]) = (d (hash(txt[s .. s+m-1]) – txt[s]*h) + txt[s + m]) mod q

hash(txt[s .. s+m-1]): Hash value at shift s.

hash(txt[s+1 .. s+m]): Hash value at next shift (or shift s+1)

d: Number of characters in the alphabet

q: A prime number

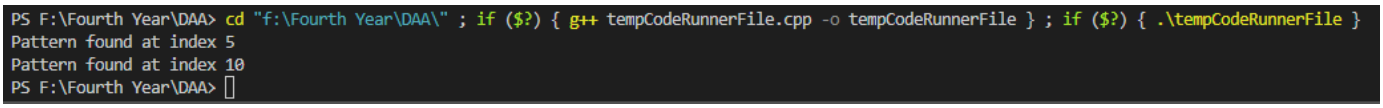
h: d^(m-1)

Code:

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

#define d 256


void search(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 << "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);
        }
    }
}

int main()
{
     string text = "ABAACAABAAAABAA";
    string pat = "AABA";


    int q = INT_MAX;


    search(pat, text, q);
    return 0;
}
```

Output Screenshot:



Time Complexity:

The **average** case running time of the Rabin-Karp algorithm is O(n+m),

but its **worst-case** time is O((n-m+1) m).

The best case is O(m).

One of the Worst cases of Rabin-Karp algorithm O(mn) occurs when all characters of pattern and text are the same as the hash values of all the substrings of txt[] match with the hash value of pat[].

For example: pat[] = “AAA” and txt[] = “AAAAAAA”.

Space Complexity

The space Complexity of the Rabin-Karp algorithm is O(m)

Comparison Tables for Algorithms

Different Techniques used by Different Algorithms

Algorithm	Techniques
Naive string search algorithm	Each character of the pattern is compared to a substring of the text which is the length of the pattern, until there is a mismatch or a match.
Rabin–Karp string search algorithm	Hashing
Knuth–Morris–Pratt algorithm	Two indices l and r into text string t
Boyer–Moore string search algorithm	Use both good suffix shift and bad character shift

Comparisons of Worst case Complexity of different algorithms

Algorithm	Preprocessing time		Matching time/Searching Phase/Running Time
	Time complexity	Space complexity	
Naive string search algorithm	O(n)	O(1)	O((n-m+1) m)
Rabin–Karp string search algorithm	O(m)	O(m)	Average O(n+m), worst O((n-m+1) m)
Knuth–Morris–Pratt algorithm	O(m)	O(m)	O(m+n)
Boyer–Moore string search algorithm	O(m + |Σ|)	O(m + |Σ|)	Ω(n/m), O(n)

Conclusion:

Thus, we have implemented and compared time complexity and analyzed the performance of the Naive string-matching algorithm and the Rabin-Karp algorithm.