

# Pune Institute of Computer Technology



**Department of Computer Engineering**

**(2022- 2023)**

**“Implement the Naive string matching algorithm and Rabin-Karp algorithm for string matching.”**

**Submitted to the**

**Savitribai Phule Pune University**

**In partial fulfillment for the award of the Degree of**

**Bachelor of Engineering**

**in**

**Computer Engineering**

**By**

- |           |                        |              |
|-----------|------------------------|--------------|
| <b>1)</b> | <b>Abhijeet Jagtap</b> | <b>41427</b> |
| <b>2)</b> | <b>Jay Sonawane</b>    | <b>41430</b> |

**Under the guidance of**

**Prof. Vaishali Kandeekar**

## 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  $B$  ( $B \geq$  size of the alphabet used in the string):

$H = s[0] * B^{(m-1)} + s[1] * B^{(m-2)} + \dots + s[m-2] * B^1 + s[m-1] * B^0$  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;  $E = (B^{m-1}) \text{ modulo } M$ 
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:

## Rabin-Karp Algorithm:

```
DAA_miniproject.py > search
1
2 d = 256
3 # pat -> pattern
4 # txt -> text
5 # q -> A prime number
6
7 def search(pat, txt, q):
8     M = len(pat)
9     N = len(txt)
10    i = 0
11    j = 0
12    p = 0 # hash value for pattern
13    t = 0 # hash value for txt
14    h = 1
15
16    for i in range(M-1):
17        h = (h*d) % q
18
19
20    for i in range(M):
21        p = (d*p + ord(pat[i])) % q
22        t = (d*t + ord(txt[i])) % q
23
24    # Slide the pattern over text one by one
25    for i in range(N-M+1):
26        # Check the hash values of current window of text and
27        # pattern if the hash values match then only check
28        # for characters one by one
29        if p == t:
30            # Check for characters one by one
31            for j in range(M):
32                if txt[i+j] != pat[j]:
33                    break
34            else:
35                j += 1
36
37        # if p == t and pat[0..M-1] = txt[i, i+1, ...i+M-1]
38        if j == M:
39            print("Pattern found at index " + str(i))
40
41    # Calculate hash value for next window of text
42
43    if i < N-M:
44        t = (d*(t-ord(txt[i])*h) + ord(txt[i+M])) % q
45
46        if t < 0:
47            t = t+q
48
49
50 # Driver Code
51 if __name__ == '__main__':
52     txt = "There are two types of string matching algorithm Naive String Matching and Rabin-Karp Algorithm for string Searching"
53     pat = "string"
54     q = 101
55
56     search(pat, txt, q)
```

## Naive algorithm for string matching

```
def search(pat, txt):
    M = len(pat)
    N = len(txt)

    for i in range(N - M + 1):
        j = 0

        while(j < M):
            if (txt[i + j] != pat[j]):
                break
            j += 1

        if (j == M):
            print("Pattern found at index ", i)

if __name__ == '__main__':
    txt = "There are two types of string matching algorithm Naive String Matching and Rabin-Karp Algorithm for string Searching"
    pat = "tring"

    search(pat, txt)
```

## Output-

The image contains two screenshots of a Python IDE. The top screenshot shows a file named `DAA_miniproject.py` with a function `search(pat, txt)` implementing a naive string matching algorithm. The code iterates over all possible starting positions in the text and compares the pattern character by character. The output in the terminal shows the pattern found at indices 24, 56, and 101. The bottom screenshot shows the same file with a more complex implementation of the Rabin-Karp algorithm using rolling hashes. The output shows the pattern found at indices 23 and 100.

```
58
59 def search(pat, txt):
60     M = len(pat)
61     N = len(txt)
62
63     for i in range(N - M + 1):
64         j = 0
65
66         while(j < M):
67             if (txt[i + j] != pat[j]):
68                 break
69             j += 1
70
71         if (j == M):
72             print("Pattern found at index ", i)
73
74
75 if __name__ == '__main__':
76     txt = "There are two types of string matching algorithm Naive String /
77     pat = "tring"
78
79     search(pat, txt)
80
```

```
PS E:\BE\41427_LP-III_codes\DAAG> & C:/Users/abhi/AppData/Local/Programs/Python/Pyth
on38/python.exe e:/BE/41427_LP-III_codes/DAAG/DAA_miniproject.py
Pattern found at index 24
Pattern found at index 56
Pattern found at index 101
PS E:\BE\41427_LP-III_codes\DAAG>
```

```
21
22 p = (d*p + ord(pat[i])) % q
23 t = (d*t + ord(txt[i])) % q
24
25 # Slide the pattern over text one by one
26 for i in range(N-M+1):
27     # Check the hash values of current window of text and
28     # pattern if the hash values match then only check
29     # for characters one by one
30     if p == t:
31         # Check for characters one by one
32         for j in range(M):
33             if txt[i+j] != pat[j]:
34                 break
35             else:
36                 j += 1
37
38     # if p == t and pat[0..M-1] = txt[i, i+1, ...i+M-1]
39     if j == M:
40         print("Pattern found at index " + str(i))
41
42     # Calculate hash value for next window of text
43     if i < N-M:
44         t = (d*(t-ord(txt[i])*h) + ord(txt[i+M])) % q
45
46         if t < 0:
47             t = t+q
48
49
50 if __name__ == '__main__':
51     txt = "There are two types of string matching algorithm Naive String /
52     pat = "string"
53     q = 101
54
```

```
PS E:\BE\41427_LP-III_codes\DAAG> & C:/Users/abhi/AppData/Local/Programs/Python/Pyth
on38/python.exe e:/BE/41427_LP-III_codes/DAAG/DAA_miniproject.py
Pattern found at index 23
Pattern found at index 100
PS E:\BE\41427_LP-III_codes\DAAG>
```

## 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.