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

1)	**Abhijeet Jagtap**	**41427**
2)	**Jay Sonawane**	**41430**

Under the guidance of

Prof. Vaishali Kandekar

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):

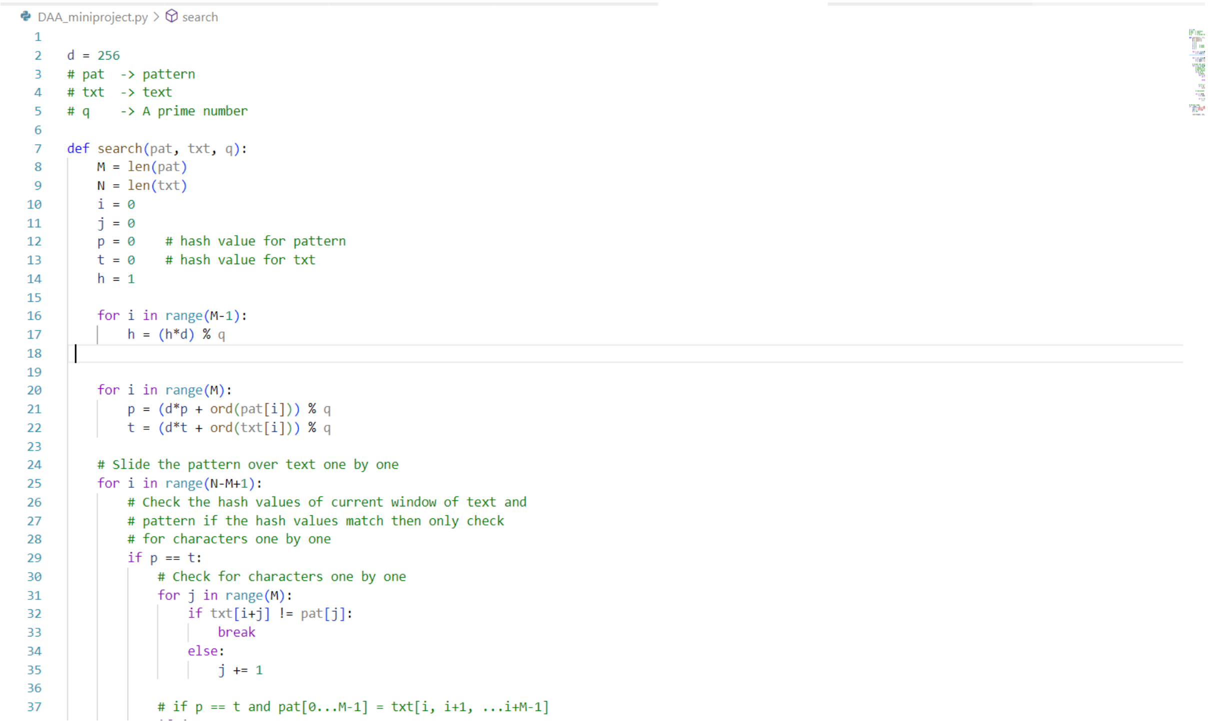
H = s[0] * B(m – 1) + s[1] * B(m – 2) + … + s[m - 2] * B1 + s[m - 1] * B0 If we calculate the number H (the hash value) for the pattern and the same number for every substring of length m of the text than the inner loop of the “naive” method will disappear – instead of comparing two strings character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

Algorithm :

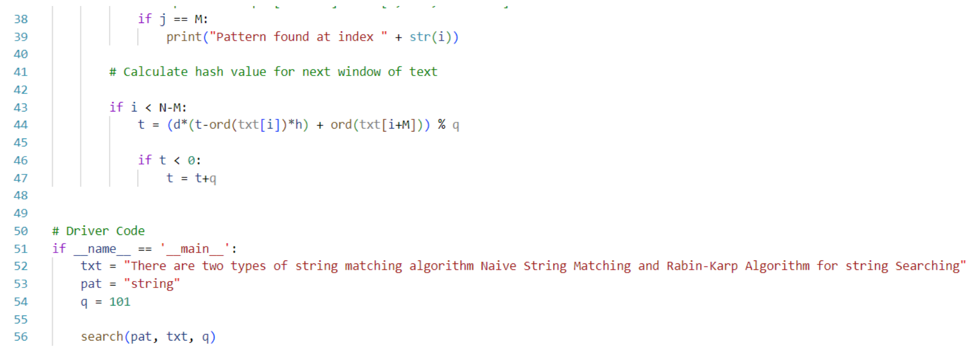
```
// correctly calculates a mod b even if a < 0
function int_mod(int a, int b)
{
    return (a % b + b) % b;
}
function Rabin_Karp(text[], pattern[])
{
    // let n be the size of the text, m the size of the
    // pattern, B - the base of the numeral system,
    // and M - a big enough prime number
    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:

Rabin-Karp Algorithm:

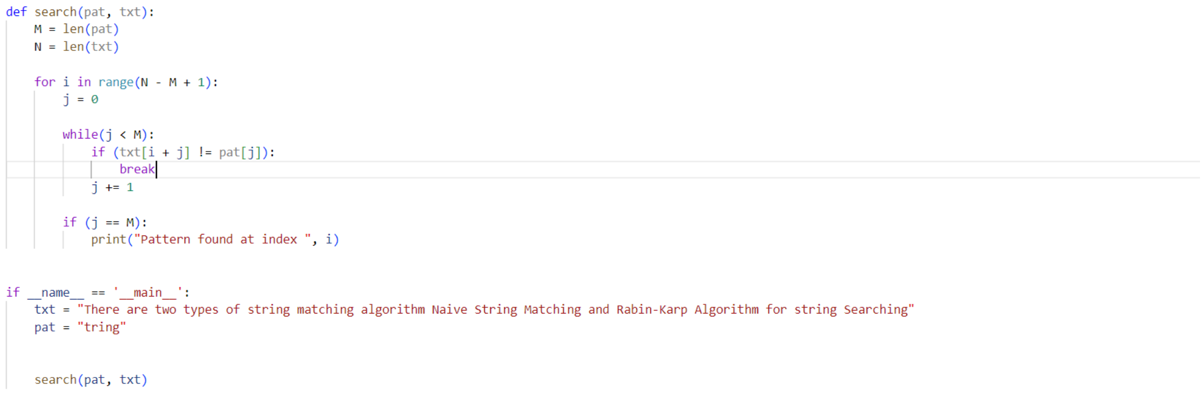


```
DAA_miniproject.py > search

d = 256
# pat  -> pattern
# txt  -> text
# q    -> A prime number

def search(pat, txt, q):
    M = len(pat)
    N = len(txt)
    i = 0
    j = 0
    p = 0    # hash value for pattern
    t = 0    # hash value for txt
    h = 1

    for i in range(M-1):
        h = (h*d) % q


    for i in range(M):
        p = (d*p + ord(pat[i])) % q
        t = (d*t + ord(txt[i])) % q

    # Slide the pattern over text one by one
    for i in range(N-M+1):
        # Check the hash values of current window of text and
        # pattern if the hash values match then only check
        # for characters one by one
        if p == t:
            # Check for characters one by one
            for j in range(M):
                if txt[i+j] != pat[j]:
                    break
                else:
                    j += 1

            # if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
```



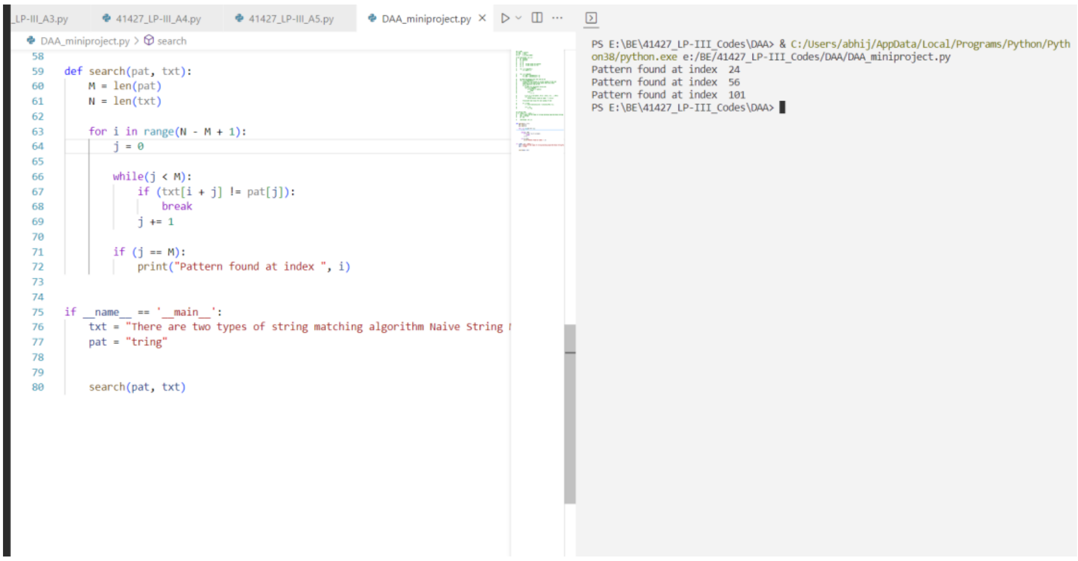
```
            if j == M:
                print("Pattern found at index " + str(i))

        # Calculate hash value for next window of text

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

            if t < 0:
                t = t+q


# Driver Code
if __name__ == '__main__':
    txt = "There are two types of string matching algorithm Naive String Matching and Rabin-Karp Algorithm for string Searching"
    pat = "string"
    q = 101

    search(pat, txt, q)
```

Naive algorithm for string matching



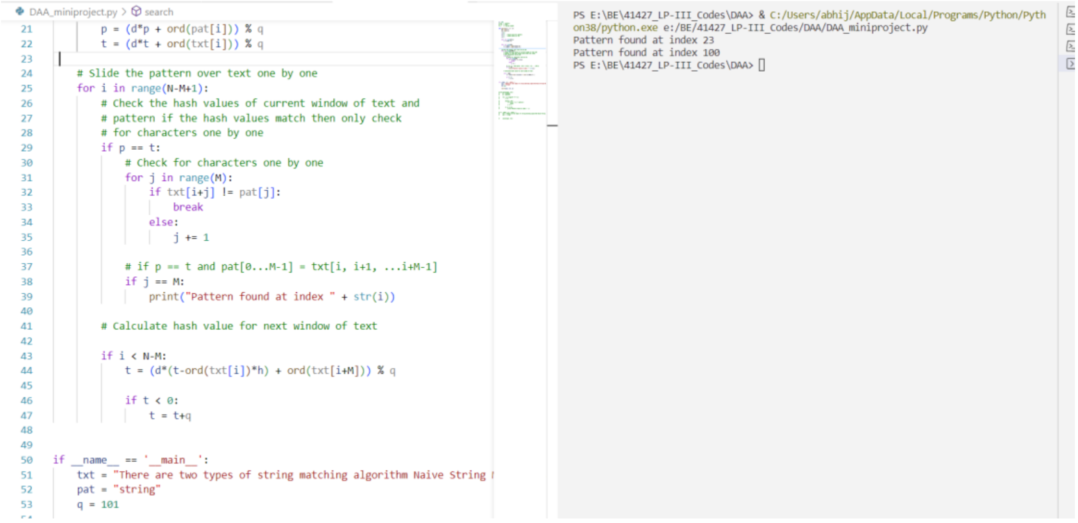
```
def search(pat, txt):
    M = len(pat)
    N = len(txt)

    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-



```python

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
    pat = "tring"


    search(pat, txt)
```

```
PS E:\BE\41427_LP-III_Codes\DAA> & C:/Users/abhij/AppData/Local/Programs/Python/Pyth
on38/python.exe e:/BE/41427_LP-III_Codes/DAA/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\DAA> 
```



```python
            p = (d*p + ord(pat[i])) % q
            t = (d*t + ord(txt[i])) % q

    # Slide the pattern over text one by one
    for i in range(N-M+1):
        # Check the hash values of current window of text and
        # pattern if the hash values match then only check
        # for characters one by one
        if p == t:
            # Check for characters one by one
            for j in range(M):
                if txt[i+j] != pat[j]:
                    break
                else:
                    j += 1

            # if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
            if j == M:
                print("Pattern found at index " + str(i))

        # Calculate hash value for next window of text

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

            if t < 0:
                t = t+q


if __name__ == '__main__':
    txt = "There are two types of string matching algorithm Naive String
    pat = "string"
    q = 101
```

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

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.