COUNTING PALINDROMIC SUBSTRING

GROUP MEMBERS:

KODI PRAVALLIKA - IIT2019234
VISHWAM SHRIRAM MUNDADA - IIT2019235
NOONSAVAATH SAMYUKTA - IIT2019236

ABSTRACT

The project aims at counting number of non-empty sub strings that are palindromes in a given string.

CONTENTS

INTRODUCTION

We are given a string, our goal is to find the number of palindromic substrings in this string.

For solving this we have analyzed two different approaches:-

- Brute force.
- Dynamic programming approach.

These algorithms are tested for different sample test cases for time complexity and space complexity. In our report, we tend to focus on establishing a rule or a relationship between the time and input data.

ALGORITHM DESIGN
Brute force(Naive method)

Each substring is denoted by a pair of variables pointing to the start and end indices of the sub-string.

- A single character substring is denoted by start and end indices being equal in value.

Checking for a palindrome is simple; we check if the ends of the substring are the same character, going outside-in:

  - If they aren't, this substring is not a palindrome.
  - Else, we continue checking inwards until we get to the middle.

PSEUDO CODE (Naive approach)

```
function MAIN()
        string s
         Input s
         print COUNTPALINDROMES(s)
function COUNTPALINDROMES(s)
         count ← 0
         for i ← 0 to length[s]
                  for j ← i to length[s]
                           if ISPALINDROME(s, i, j)
                                    count <- count+1
         return count
```

```
function ISPALINDROME(s, st, end)
         mid ← (end-st+1)/2
         for i ← 0 to mid
                  if(s[st+i] != s[end-i])
                           return false
         return true
```

ALGORITHM DESIGN
Dynamic programming approach

▶We create a dp table(dp[n][n]), where dp(i, j) tells us whether the substring composed of the ith to the jth characters of the input string, is a palindrome or not.

▶There are two base-cases:

$$\text{Case 1 : } dp(i, i) = \text{true} \text{ , Case 2 : } dp(i,i+1) = \begin{cases} \textit{True} & \text{if } s[i]=s[i+1] \\ \textit{False} & \text{Otherwise} \end{cases}$$

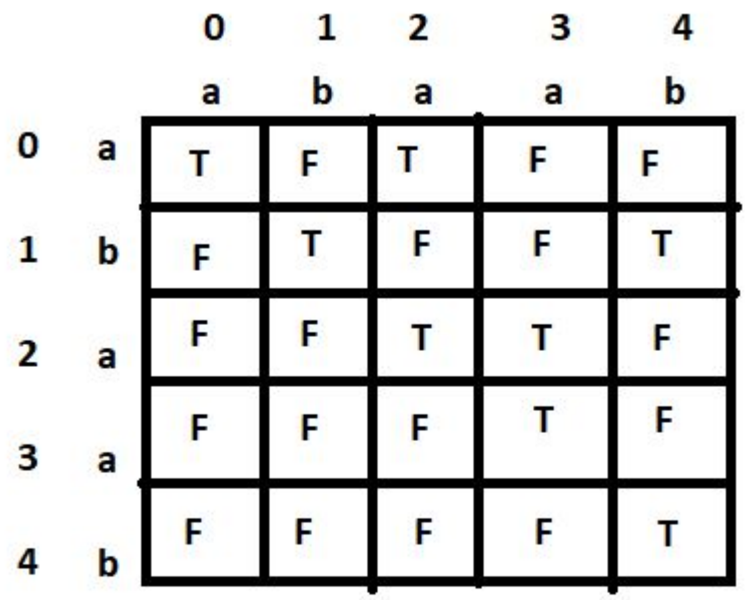
▶A string is considered a palindrome(for length >2) if:

- Its first and last characters are equal, and
- The rest of the string (excluding the boundary characters) is also a palindrome.

$$dp(i,j) = \begin{cases} \textit{True} & \text{if } dp(i+1,j-1) \text{ \&\& } s[i]=s[j] \\ \textit{False} & \text{Otherwise} \end{cases}$$

EXAMPLE

DP TABLE FOR STRING “abaab”



		0	1	2	3	4
		a	b	a	a	b
0	a	T	F	T	F	F
1	b	F	T	F	F	T
2	a	F	F	T	T	F
3	a	F	F	F	T	F
4	b	F	F	F	F	T

Count of palindromic substrings in this example is number of true values in the table, which is equal to 8

PSEUDO CODE(DP Approach)

```
function MAIN()
    string s
    Input s
    print COUNTPALINDROMES(s)

function COUNTPALINDROMES(s)
    n ← length[s]
    count ← 0
    bool dp[n][n]
    MEMSET(dp, false, SIZEOF(dp))

    for gap <- 0 to n
        for i ← 0 to n-gap
            j ← i+gap

            if gap = 0
                dp[i][j] = true

            else if gap = 1
                dp[i][j] = s[i] == s[j]

            else
                dp[i][j] = s[i] == s[j] && dp[i+1][j-1]

            if(dp[i][j])
                count ← count+1
    return count
```

ALGORITHM ANALYSIS
Brute force(Naive method)

In case of brute force algorithm we use nested loop to traverse through the string for all possible substrings. Now inside this nested loop we check if that substring is palindrome. To check whether if it is palindromic substring, we use a for loop.

So, totally we use nested loop which consists of 3 ’for’ loops, therefore the time complexity is $O(N^3)$.

- ►Time Complexity: $O(N^3)$

- ►Space Complexity: O(1)

O(1). We don't need to allocate any extra space since we are repeatedly iterating on the input string itself.

ALGORITHM ANALYSIS
Dynamic programming approach

In case of dynamic programming approach, we create a dp table (dp[n][n]), we traverse through the table and give the values as discussed in previous slides. So, for traversing we use a nested loop.So the time complexity O(N^2)

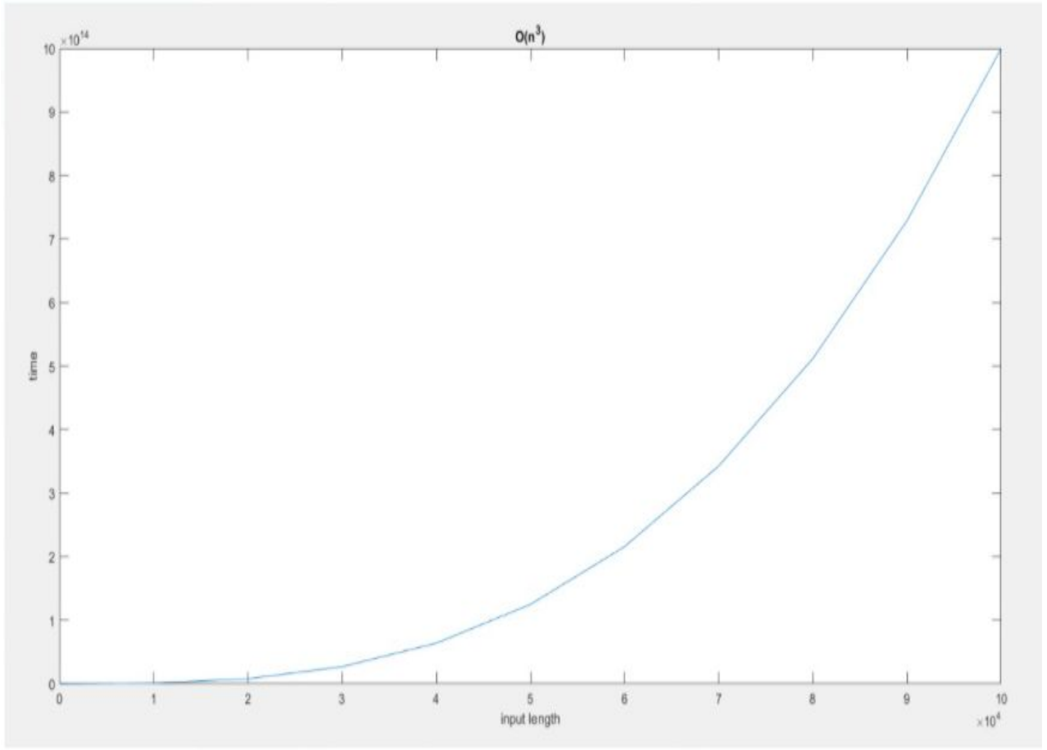
- ▶Time Complexity: $O(N^2)$

As we have created a table of size n x n, space complexity is also $O(N^2)$

- ▶Space Complexity: $O(N^2)$

EXPERIMENTAL STUDY

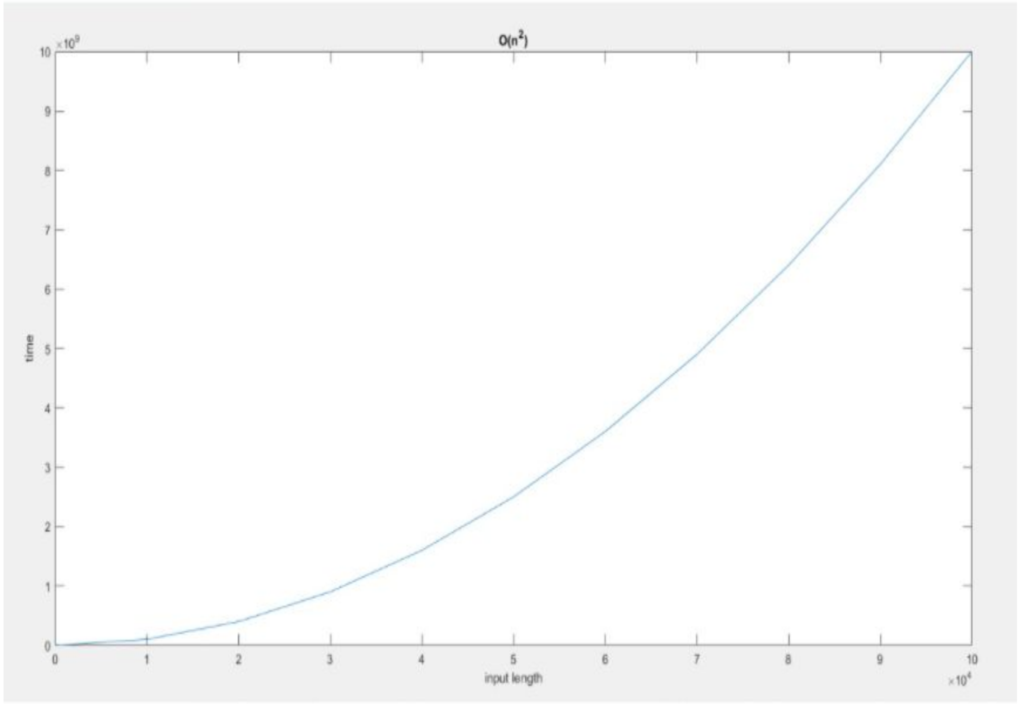
Brute force(Naive method)



In the brute force as n increases, the time increases by n^3.

EXPERIMENTAL STUDY

Using Dynamic programming algorithm



In the Dynamic programming as n increases, the time increases by n^2.

IMPORTANT LINKS

Reference links:-

https://www.geeksforgeeks.org/longest-palindrome-substring-set-1/

https://leetcode.com/problems/palindromic-substrings/

Code link:-

Dynamic programming approach: https://ideone.com/5j8HQW

Brute force: https://ideone.com/ZkKHwv

CONCLUSION

From the experimental study we concluded that the average running time of Dynamic programming algorithm is best, which can be observed from the mutual graph of Dynamic programming algorithm and Brute Force algorithm as shown.

THANK YOU