Design and Analysis of Algorithms

Group - 2

Group Members:

IIT2019189: Nidhi Kamewar
IIT2019190: Rishi Gupta
IIT2019191: Pechetti Venkata Karthik

Content Listings

- Problem Statement
- Introduction
- Algorithm Design
- Code Explaination
- Example
- Pseudo Code
- Time Complexity
- Space Complexity
- Conclusion
- References

Problem Statement

Given a string S, count the number of non-empty sub strings that are palindromes. A sub string is any continuous sequence of characters in the string. A string is said to be palindrome, if the reverse of the string is same as itself. Two sub strings are different if they occur at different positions in S. Solve using Dynamic programming.

Introduction

- In this problem , we have to find the count of the substrings of a given string which are palindromic.
- The given problem has :
  - **Overlapping sub-problems** of recalculating the palindromic substrings of substrings in the given string.
  - **Optimal substructure property** i.e., the given problem can be solved by the solving the sub-problems of the problem.
- Hence, we can solve the given problem using Dynamic programming.

Algorithm Design

- The algorithm we designed is based on **Top-Down** approach of memorization in Dynamic Programming .
- This is because at first we check if the answer to the sub-problem of given problem is already computed or not (this is done using the memorization table):
    - if present we return the value in the table.
    - else we now treat the sub-problem as the main problem and **recursively** divide it into sub-problems until a base case is reached.

Code Explanation

1) First , given string S is passed as an input to the function **countSubstrings().**
2) The function **initializes all the values of the memorization table to -1** which represents that result of the sub-problem is not yet computed.
3) We **initialize the value of count to 0** which indicates the number of palindromes of the string which will be incremented when a palindromic substring is found.
4) The function **iteratively checks the substrings** by considering the strings from i to j where i will range from 0 to n-1 and j will range from i to n-1 respectively by calling the function isPal().
5) isPal() checks if the substring of S between i to j is a substring or not by **checking the lookup table** and returning the respective value or computing the result by **recursively calling itself** and storing the values in the lookup table.

Code Explanation (Contd.)

7) If **the starting and the ending character of the string is the same** then it will recursively call itself to check whether the substring from i+1 to j-1 is palindrome or not i.e., the substring in the middle.
8) If the value of the **starting index is greater than the ending index,** this can be called only by a string of length 2 with same characters or a string of length 1. Both are palindromes and hence we return 1.
9) The respective result will be **stored in the memorization table.**
10) **If the substring is a palindrome then the value of count is incremented by 1** it repeats till all the substring are checked and finally the value of the count is returned

Example

Consider string s= “aabcca”

- i=0 : “a”,“aa”,“aab”,“aabc”,“aabcc”, “aabcca”
- i=1 : “a”,“ab”,“abc”,abcc”,“abcca
- i=2 : “b”,“bc”,“bcc”,“bcca”
- i=3 : “c”,“cc”,”cca”
- i=4 : “c”,“ca”
- i=5 : “a”

- Here in the above example the strings like **abcc** are not recomputed as the result of the sub-problem is already computed when calculating the result of a**abcc**a.
- When the algorithm checks strings aa , cc the value of count is incremented by 1 and finally reaches 8 which is the total number of palindromic substrings of the given string “aabcca”.
- **a,a,b,c,c,c,a,aa,cc**

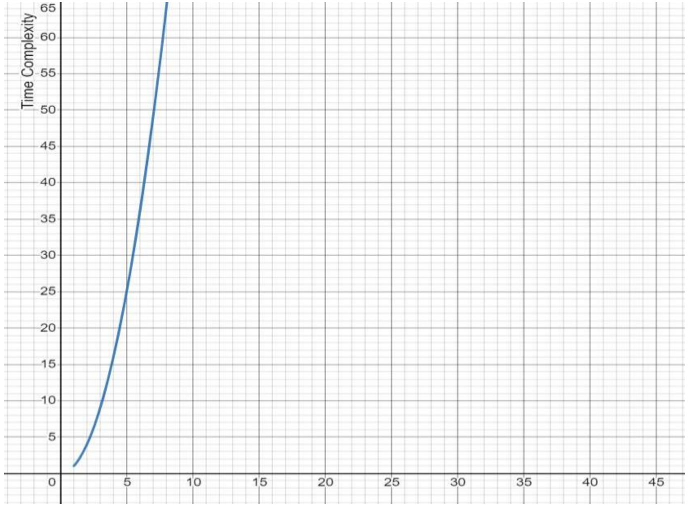
Pseudo Code :

```
Function isPal(string s, int i, int j):

        if i > j then
            return 1;
        end if

        if dp[i][j] != -1 then
            return dp[i][j];
        end if

        if s[i] != s[j]  then
            return dp[i][j] = 0;
        end if

        return dp[i][j] = isPal(s,i+1,j-1);
```

```
Function countSubstrings(string s):

        int n ← s.length();
        int count ← 0;

        for  i ←  0 to n-1 do
            for j ←  i  to n-1 do
                if isPal(s, i, j) then
                    count++;
                end if

        return count;
```

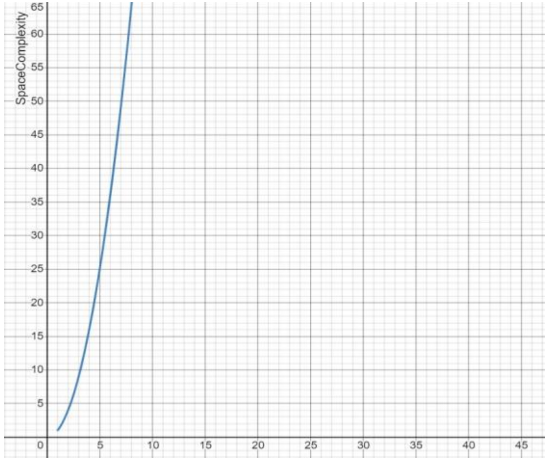
Time Complexity

- The algorithm will have a **O(n^2)** time complexity where n is the length of the string.
- This is because all possible substrings are being visited and are checked whether they are a palindrome or not. The total number of substrings is n*(n+1)/2 which is in the order of n^2. Using the memorization table, it can be checked whether a substring is palindrome or not in O(1). Hence, the total time complexity is **O(n^2)**.
- Graph:



Space Complexity

- The algorithm will have a **O(n^2)** space complexity where n is the length of the string.
- This is because a memorization table (n x n 2d array) is used to store the results of the sub-problems.
- Therefore, **O(n^2) auxiliary space** is required.
- Graph:



Conclusion

- We proposed a solution to calculate the number of palindromic substrings of a given string using dynamic programming approach .
- The approach has a time complexity of O(n^2) which is found through the experimental study as well as the asymptotic analysis of the graph.

References

- https://www.geeksforgeeks.org/count-palindrome-sub-strings-string/
- Introduction to Algorithms by Thomas.H.cormen