DESIGN AND ANALYSIS OF ALGORITHMS

Group 18

IIT2019237 - Ishneet Sethi

IIT2019238 - Chandramani Kumar

IIT2019239 - Mrityunjaya Tiwari

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.

APPROACH-1(Recursion)

Brute Force

- The idea is to recursively check for the substrings that are palindrome .
- We call recursive function CountPS on the string with low as 0 and high as len(string) -1.
- If length is 1,i.e low = high, we return 1 since it's a palindrome.
- In recursion if low crosses high then it will be an invalid condition conditions we need to return 0.
- If string from low to high is palindrome, we will set cnt =1.
- In each call we will count :
  a) all palindromes from "low+1" to "high"
  b) all palindromes from "low" to "high-1"
  c) because of the above two recursive calls, since we have counted twice all palindromes from "low+1" to "high-1", so we subtract one count.
- Finally we will return cnt.

ALGORITHM - 1

Pseudo Code

```
function main( )
    get string
    long long int n ← string.size()
    long long int res
    res ← CountPS(string,0,n-1)
    print res
```

```
function CountPS(string,l,r)
    if(l>r)
        return 0;
     if(l==r)
        return 1;
     long long int cnt ← 0
    if (ispalindrome(str,l,r))
        cnt ← 1

     cnt ← cnt + CountPS(string,l+1,r)
    cnt ← cnt + CountPS(string,l,r-1)
   cnt ← cnt - CountPS(string,l+1,r-1)

   return cnt
```

```
function ispalindrome(string s,int i,int j)
    i ← 0, j ← n-1
    while(i<=j)
        if(s[i++]!=s[j--])
        return 0;
    end

   return 1;
```

APPROACH-2

Two-loop Strategy

- The idea is to consider all the substrings and check if they form a palindrome or not.
- To do this we are implementing the two for-loops to consider all the substrings.
- Basically the first for-loop determines the start index of the substring and the second for-loop determines the end index of the substring.
- Hence one by one considering all the substrings and checking if it's a palindrome we increment the count.
- To check if the substring is a palindrome or not, we use the ispalindrome function explained already in the above mentioned approaches.
- Finally the total number of palindrome substrings present in the string is printed.

ALGORITHM - 2(Two loops)

Pseudo Code

```
function ispalindrome(string s,int
i,int j)
    i ← 0, j ← n-1
    while(i<=j)
        if(s[i++]!=s[j--])
        return 0;
    end
   return 1
```

```
function main( )
      get string
      long long int n ← string.size()
      long long int count
      count ←0
      for i ←0 to n
        for j ← to n
           if(ispalindrome(s,i,j)
              count ←count+1
  print count
```

APPROACH-3(Dynamic Programming)

- To do this we are implementing the bottom up approach of dynamic programming.
- We are creating a 2D matrix with string length as the number of rows and columns.
- The upper half of the matrix separated by the main diagonal corresponds to all the substrings possible for the given string. Further, the cells on each diagonal in the upper half correspond to substring of the same length.
- Basically, the cells on the main diagonal correspond to those substring whose lengths are 1 and similarly length increases as diagonal no increase. We will be marking the cells on the main diagonal of the matrix as true because the single letter string is palindrome by default.
- For the cells on the second diagonal in the upper half, we will check if both characters in the string are the same (marked cell true) or not (marked cell false).
- For rest of the diagonals, we will check similarity of first and last character of the substring and also check if the string between the first and the last character is palindrome (marked true) or not (marked false).
- Then in between string palindrome check can be done by looking the cell at left-down.
- Meanwhile, we will be incrementing the counter if cells' value comes out to be true.

ALGORITHM - 3 (Dynamic Programming)
Pseudo Code

```
function main( )
    get string
    long long int res
    res ←
    count_palindromes(strin
    g)
    print res
```

```
function count_palindromes(string)
    long long int cnt ← 0

    long long int len ← string.size()
vector<vector<bool> >  dp(len, vector<bool>
(len))

    for gap ← 0  to len-1  do
       for i ← 0 , j ← gap  to len-1  do
        if  (gap  ← 0) then
            dp[i][j]  ←  true
        else if ( gap ← 1) then
            if ( string[i] == string[j] )
          dp[i][j] ← true
            else
          dp[i][j] ← false
        else
            if ( (string[i]
== string[j]) and
(dp[i + 1][j - 1]) )
            dp[i][j] ←
true
            else
          dp[i][j] ← false

    if ( dp[i][j] ==true )
       cnt ← cnt+1

return cnt
```

APPROACH-4

Manacher's Algorithm

- The idea is to count the number of even length and odd length palidromic substrings separately; the idea for each is basically the same, but there are enough variations that it is worth separating the two entirely.

- The idea is that we don't have need to repeat calculations on elements we have "seen before".

- If (when going from left to right) we have seen elements at index R and we are at index i,

  we know there is some index k s.t. R is the right endpoint of k's longest palindrome centered around it.

- Thus, one can check that we can map i to some index j (where j is on the other half of k's longest palindrome and is the same distance from the left endpoint instead of the right endpoint), we can use the longest palindrome length at index j to give a lower bound on the longest palindrome at i (notice that the characters around both of them within the longest palindrome k is contained in are the same - hence we already have computed everything we need unless if we extend pass the boundary).

ALGORITHM - 4 (Manacher's Algorithm)

Pseudo Code

```
function main( )
    get string
    long long int n ← string.size()
    long long int answer
    answer ← 0

vector<long long int>d1(n)
  for i ←0 to n,l ←0,r ← -1
    long long int  k ← (i > r) ? 1 : min(d1[l + r - i], r - i + 1)
        while (0 <= i - k && i + k < n && s[i - k] == s[i + k])
            k←k+1
        end
    d1[i] ←k--
    if (i + k > r)
        l ← i - k
        r ←i + k

vector<long long int>d2(n)
for i ←0 to n,l ←0,r ← -1
long long int  k ← (i > r) ? 0 : min(d2[l + r - i+1], r - i + 1)
    while (0 <= i - k-1 && i + k < n && s[i - k-1] == s[i + k])
        k←k+1
    end
d1[i] ←k--
    if (i + k > r)
        l ← i - k-1
        r ←i + k
for i←0 to n
answer←answer+d1[i]/2
+1
for i←0 to n
answer←answer+d2[i]/2

print answer
```

Illustration

Input : string = acca

Length of the given string = 3

What tabulation table depicts :

a	**c**	**c**	**a**
a	**ac**	**acc**	**acca**
0	**c**	**cc**	**cca**
0	**0**	**c**	**ca**
0	**0**	**0**	**a**

Illustration Continued ...

First, we will create a 2d matrix dp[4][4]
Initially dp[][] matrix and cnt be :

dp[4][4] = { 0, 0, 0, 0
0, 0, 0, 0
0, 0, 0, 0
0, 0, 0, 0 }
cnt = 0

Clearly, there are 4 diagonals which we will traverse.
Diagonal 1 : connecting (0,0) to (3,3)
Diagonal 2 : connecting (0,1) to (2,3)
Diagonal 3 : connecting (0,2) to (1,3)
Diagonal 4 : connecting (0,3) to (0,3)

Now, using algorithm 2 for first diagonal, the updated matrix and cnt will be :
Since "a" , "c", "c", "a" are palindromes, so mark true

dp[4][4] = { 1, 0, 0, 0
0, 1, 0, 0
0, 0, 1, 0
0, 0, 0, 1 }
cnt = 4

Now, for all other diagonals algorithm 2 implementation is demonstrated as :

Illustration Continued …

Diagonal 2 :
For dp[0][1] :
Since , ‘a’ != ‘c’ so mark false
For dp[1][2] :
Since , ‘c’ == ‘c’ so mark true
For dp[2][3] :
Since , ‘c’ != ‘a’ so mark false
Updated dp[][] matrix and cnt :
dp[4][4] = { 1, 0, 0, 0
0, 1, 1, 0
0, 0, 1, 0
0, 0, 0, 1 }
So, cnt = 5

Diagonal 3 :
For dp[0][2] :
Since , ‘a’ != ‘c’ so mark false
For dp[1][3] :
Since , ‘c’ != ‘a’ so mark false

Updated dp[][] matrix and cnt :
dp[4][4] = { 1, 0, 0, 0
0, 1, 1, 0
0, 0, 1, 0
0, 0, 0, 1 }
So, cnt = 5

Diagonal 4 :
For dp[0][3] :
Since, ‘a’ == ‘a’ so look for
dp[0+1][3-1]
Since, dp[1][2] = true, so mark true

Updated dp[][] matrix and cnt :
dp[4][4] = { 1, 0, 0, 1
0, 1, 1, 0
0, 0, 1, 0
0, 0, 0, 1 }
So, cnt = 6

Output: 6.

ALGORITHM ANALYSIS
ALGORITHM 1

Time Complexity:

- In this approach,for each recursive step we check if the substring is palindrome or not.
- If it is palindrome we set cnt to 1 and recurse and add for the remaining substrings for (i+1,j) and (i,j-1) and subtract the common palindrome in string range (i+1,j-1).
- Since for each recursive call we make 3 recursive calls hence the total time complexity would be exponential i.e 3^(N).

So,T = O(3^(N)).

Space Complexity:

- Our memory complexity is determined by the number of return statements because each function call will be stored on the program stack.
- To generalize, a recursive function's memory complexity is O(recursion depth). Hence in this case space complexity would be O(N).

ALGORITHM ANALYSIS
ALGORITHM 2

Time Complexity :

In this approach, we are following some steps :

1) Traversing each row and column of the upper half of the dp[][] matrix to check whether the substring that corresponds to that cell is palindrome or not.

Clearly, time complexity gets affected by the above step.
Assuming, time taken for comparisons to be unit.
Let's suppose step 1 time complexity be T

Since traversing each rows and columns ,
So, T = O(N * N)
` => T ~ O(N*N)

Best Case Complexity: O(N*N)

Worst Case Complexity: O(N*N)

Average Case Complexity : O(N*N)

Space Complexity :

- Assuming, variables created don't affect the space used calculation much.
- Since we are creating a 2D matrix of the size N X N, so we are using N*N space .Hence, the space complexity of the algorithm 2 will be O(N*N).

ALGORITHM ANALYSIS
ALGORITHM 3

Time Complexity :

In this approach,we consider two for-loops,
The first for-loop determines the start index of the substring and the second for-loop determines the end index of the substring.
Since it takes two for-loops to consider all the substrings the time complexity for computing the substrings would be O(N^2).
For every computed substring we also have to check if it's a palindrome or not which requires a traversal of half the length of the substring,hence the time complexity would be O(N).
Since it is being done for all the substrings the total time complexity would be O(N^3);

=> T ~ O(N*N*N)

Space Complexity :

- For the space analysis of the algorithm,some variables are used having constant space requirement.
- As far as the algorithm is concerned we do not use any extra array/string/container to find the result.Hence the overall space complexity is constant i.e O(k).

ALGORITHM ANALYSIS
ALGORITHM 4

Time Complexity :

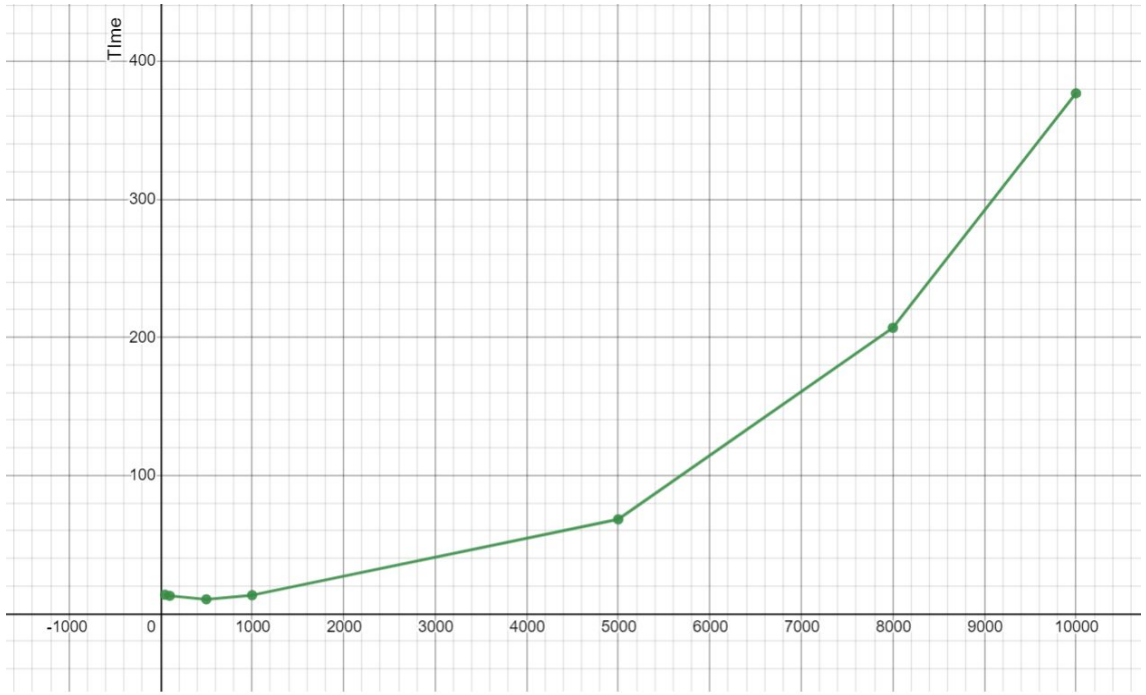
At the first glance it's not obvious that this algorithm has linear time complexity, because we often run the naive algorithm while searching the answer for a particular position.We can notice that every iteration of trivial algorithm increases r by one. Also r cannot be decreased during the algorithm. So, trivial algorithm will make O(N) iterations in total. Also, other parts of Manacher's algorithm work obviously in linear time.

Thus, we get O(N) time complexity.

Space Complexity :

- For the space analysis of the algorithm we have used two vectors/arrays to store odd length and even even length palindromes apart from the input string and some variables having constant space requirement.Hence the overall space complexity is O(N).

EXPERIMENTAL RESULT



Time complexity graph for aposteriori analysis. (Keeping N on X-axis)

CONCLUSION

The algorithms discussed in this paper can be used to find the number of palindromic substrings. In the matter of comparison between these two algorithms,in Approach 1, time complexity i.e (O(3^N)) and in approach 2 it would be (O(N^3)). Both would be higher as compared to Algorithm 3 i.e ((O(N^2)).)

So we can conclude that Dynamic Programming Solution i.e the Algorithm 3 is optimal and efficient among them.

However, there exists a better standard algorithm named Manacher's algorithm with time and space complexity of O(n). We have discussed about it also as algorithm 4.

THANK YOU