Finding the number of non-empty substrings that are palindromes from a given string

Harsh Meena - IIT20190005,
Asha Jyothi Donga - IIT2019006,
Sampada Kathar - IIT2019007

Date: 28-03-2021

Abstract

In this paper we have devised an algorithm which finds the number of non-empty substrings that are palindromes from a given string

1 Problem

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.

2 Keywords

Strings, Palindromes, Dynamic Programming

3 Introduction

Dynamic Programming: It is an optimization of the recursion technique which stores the outputs of repeated inputs to the recursion so that a subproblem which has already been solved doesn't need to be solved again.
Palindromes: Palindromes are words which are the same read forwards to backwards and backwards to forwards.

4 Algorithm Design

To find the number of non-empty substrings that are palindromes from a given string:

1. Our base condition is $i > j$. Initial values are $i = 0$, and $j = n - 1$ where n is the length of the substring. Return 1 if this condition is met.
2. Check whether the characters are equal for all i and j, if not then return 0.
3. Increment i and decrement j and repeat the process. Do this for all values of i and j

5 Algorithm Analysis

```
bool isPal(string s, int i, int j) {
    if (i > j)
        return 1;
    if (dp[i][j] != -1)
        return dp[i][j];
    if (s[i] != s[j])
        return dp[i][j] = 0;
    return dp[i][j] = isPal(s, i + 1, j - 1);
}

int palindrome_count(string s) {
```

```
    memset(dp, -1, sizeof(dp));
    int n = s.length();
    int count = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (isPal(s, i, j))
                count++;
        }
    }
    return count;
}
```

5.1 Time Complexity Analysis

Using this reccursive dynamic programming approach, the function is called a total of maximum $n^2 + c$ (here c is constant) times to calculate each dp value. Thus the worst case complexity of this algorithm will be $O(n^2)$.
The best case occurs when either n is equal to 1, i.e., when string is of length 1. Thus, the best case time complexity is $\Omega(1)$

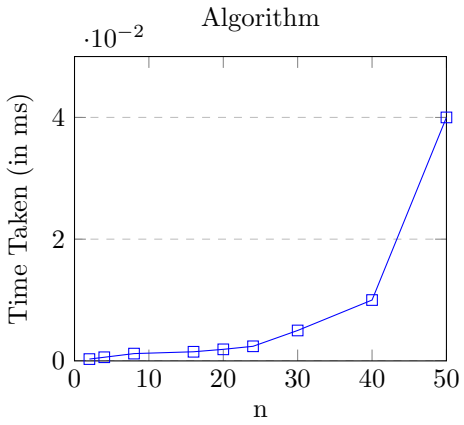
5.2 Space Complexity

This algorithm has a space complexity of $O(n^2)$ for the dp table.

6 Experimental Analysis

In the following table some cases are plotted for the first algorithm,

n	Time Taken (in ms)
2	0.0003
4	0.0006
8	0.00012
16	0.0015
20	0.0019
24	0.0024
30	0.005
40	0.010
50	0.04



The above graph shows the variance of Time with respect to size of the second array, and follows our theoretically calculated TC of $O(n^2)$.

7 Conclusion

From this paper we can conclude that top down dynamic programming algorithms, which make use of memoization are quite efficient and should be used to solve recursion problems when there are repeating sub problems.

8 References

[1]vaibhav29498, 'substring palindroms', GeeksforGeeks, 2018. [Online]. [Accessed: 1-Feb-2021]
[2]' Dynamic Programming', tutorialspoint. [Online]. [Accessed: 1-Feb-2021]
[3]GeeksforGeeks, 'reccursive dp — Set 1 (Introduction)', GeeksforGeeks, 2018. [Online]. [Accessed: 1-Feb-2021]