Assignment 4

Michele Fattoruso - Z1840898

October 2018

1 Longest Palindrome Sub-sequence

To solve the problem of finding the longest palindrome sub-string, we Initially try to find a recurrence relation in the problem, and then we can try to solve it with a dynamic programming approach.

1.1 Explanation

First of all, a string is defined Palindrome when we can read it the same way both backward and forward. We can start to identify the base cases of the problem.

- If a string is a one character string, then it's obviously palindrome
- If a string is composed of two character, then is palindrome if both characters are equal
- If string has size greater than 2,it is palindrome if the first and last character are equal, and the inner sub-string, composed of the string with stripped the first and last character is palindrome.

This will help us to identify the longest sub-string, so a contiguous sequence of character that form a palindrome string. What we are trying to identify instead is the longest sub-sequence, so even a non contiguous sequence.

1.2 Recurrence

To resolve this problem, we will have to modify our initial recurrence. The base cases will be the same, but the recurrence step will be different. Let's consider our recursive function name is "LP".

- if(s.length < 2)
  return s.length
- if(s.length $==$ 2 and s[i]$==$ s[j]) //with i,j beginning and end of string
  return s.length

- if(s.length > 2 and s[i]== s[j])

  return 2 + max(LP(s.substr(i+1,j),LP(s.substr(i,j-1))

- if(s.length > 2 and s[i]!= s[j])

  max(LP(s.substr(i+1,j),LP(s.substr(i,j-1))

this strategy will return us the size of the longest palindrome sub-sequence in the string.

The problem with this approach, is that for every time we try to calculate a sub-string, we will have to recalculate the same values over and over, as the same sub-string will be analyzed more than once. This is where we can use dynamic

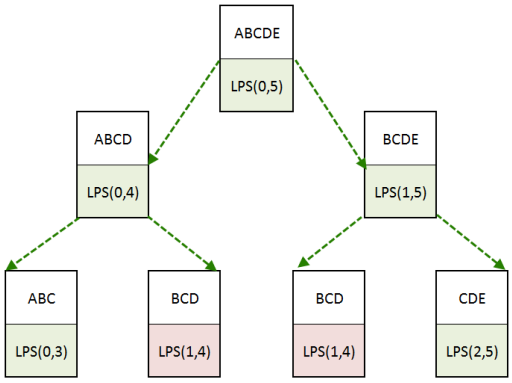

Figure 1: Example of recurrence for the computation of a palindromic sub-sequence. The red boxes represents the recurring computation, so the substrings which value gets calculated more than once

programming to avoid redundant computation. What we need to store, is the value of the maximum palindrome sub-sequence found for each sub-string, given indexes i and j as starting and ending position for the sub-string. To store this information, what we will need is a 2D array, where for each possible sub-sequence starting at i and ending at j, we store in the array position [i][j] the value of the maximum found palindrome sub-sequence.

We can solve this algorithm with a bottom up approach, by starting from the smallest sub-string, and then solving bigger problems. What we initially know is that a single character string is a palindrome sub-string, so we initially initialize the whole structure to zero, with the elements in position [i][i] to 1.

We then loop over the string, by considering first all the strings of length 2, then length 3 and so on until we reach the final case there the length of the sub-string is equal to the length of the whole string. At every iteration, we check if the starting and ending character of the sub-string are equal, and if so, we save in the 2D array in the position [i][j] the value of the longest inner

palindrome, so the array value in position [i+1][j-1] + 2, if not, we store just the maximum of the inner sub-sequence.

Start \ End		B	B	A	B	A
		0	1	2	3	4
B	0	1	2	2	3	3
B	1		1	1	3	3
A	2			1	1	3
B	3				1	1
A	4					1

Figure 2: Example of Final result for the computation of a palindromic sub-sequence using dynamic programming

Once we calculate the whole 2D array, we will have in the position [0][length] the value of the maximum palindrome sub-sequence in the string

1.3 Code

```
int LongestPalindromeSubsequence(string s) {
    int length = s.length();
    if(length < 2){
        return length;
    }

    vector<vector<int>> longest(length, vector<int>(length,0));

    for(int i=0;i<length;i++){
        longest[i][i]=1;
    }

    for(int k=1;k < length;k++ ){
        int i,j;
        for(i=0,j=k;j<length;i++,j++){

            if(s[i]==s[j]){
                longest[i][j]= longest[i+1][j-1]+2;
            }else{
                longest[i][j] = max(longest[i+1][j],
                                    longest[i][j-1]);
            }
        }
    }
    return longest[0][length-1];
}
```

1.4 Complexity

Analyzing the algorithm, we can see that the space complexity of the algorithm is equal to $\mathcal{O}(n^2)$ where n is equal to the length of the string. Also,thanks

to dynamic programming, we are not going to spend computational time by recalculating more than once the same sub-string value, as the values will be stored in the matrix.The algorithm calculates all the values of a triangular superior matrix of size n^2, so we execute roughly $n^2/2$ computation. The Time complexity of our algorithm will be $\mathcal{O}(n^2)$

2 Coin changing

The current problem, requires an algorithm that given a number of coins, and a total, will return the minimum number of coins required that sum up to a defined amount.

2.1 Explanation

The problem can be divided in two sub-problems, every time we try to calculate the number of coins necessary to sum up to the total amount. Given a coin C, and an amount A, the total number of coins necessary can be a combination of the coins including C, or excluding C.

- if we include C, the minimum number of coins will be the amount of coins necessary to sum up to A-C + 1
- if we exclude C, the min minimum number of coins will be the number of coins necessary to sum up to the current amount, excluded the coin C

Given this recurrence, we can easily figure out that, most of the values gets calculated more than once, for example, given amount = 6 and coins = [2,3,4], we will have the following recurrence tree:

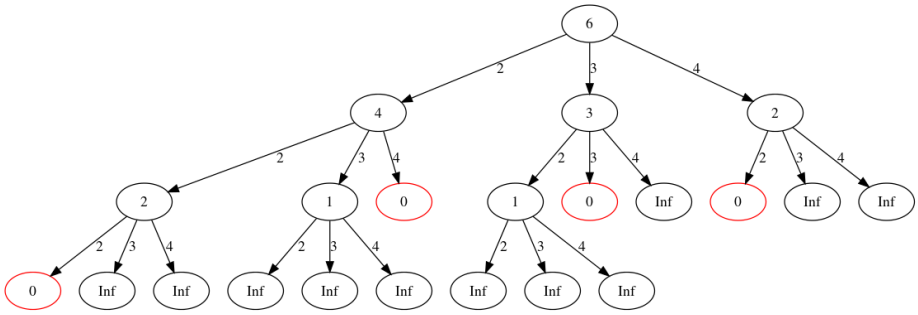

Figure 3: Example of recurrence for the calculation of the minimum amount of coins needed to sum up to 6

As we can see from the image, we calculate the coin change of zero 4 times, the coin change of 1 and 2 two times. And this is just for a small example.

Once spotted this recurrence, we can try to apply dynamic programming to store already computed information, in a storage space. In out case we will use

a one dimensional array of integers, which will store at index i, the minimum amount of coins needed to sum up to i;

Said so, we can try to resolve the problem thought a bottom up approach.

A bottom up approach will solve first the smallest cases, and with that build up to the more complex cases, until we reach our solution. Our approach will start from the fact that, if the amount is zero, then the minimum amount of coins needed will be zero, and if the total amount is a negative number, we will return -1 just for correctness.

2.2 Code

We initially initialize an array of size amount + 1 set to MAX, and then we set the amount equal to 0 as 0 coins needed. After that we perform a double loop over all the coins, and for each coin we loop over all the possible amounts up to the input amount.

At each iteration, we check if the current coin can be used for the current amount, by simply checking if the coin is smaller or equal than the current amount. If it's smaller or equal then we compute the value of the current value, by trying the two combinations of sum with and without the current coin, and we store the one with the fewer used coins through the minimum function.

```
int coinChange(vector<int>& coins, int amount) {
    int coinsSize=coins.size();
    int maxAmount = amount + 1;
    int minCoins[amount + 1];
    minCoins[0]=0;

    for(int i=1;i<=amount;i++){
        minCoins[i]=maxAmount;
    }

    //calculate all the amounts up to the desired amount
    for(int currAmount = 1; currAmount <= amount; currAmount++ ){
        //Loop over all the coins,and get the minimum amount of it
        for(int coinIndex = 0; coinIndex < coinsSize; coinIndex++){
            int currentCoin = coins[coinIndex];
            if(currentCoin <= currAmount){
                minCoins[currAmount] = min( minCoins[currAmount]
                        , minCoins[currAmount - currentCoin] + 1);
            }

        }
    }

    return minCoins[amount] > amount ? -1 : minCoins[amount];
}
```

2.3 Example

Amount = 6, Coins = [2,3]

Initialization: | 0 | | 7 | | 7 | | 7 | | 7 | | 7 |

Amount = 1, Coin = 1 | 0 | 1 | 7 | 7 | 7 | 7 | 7 |
Position 1 updated from 7 to 1

Amount = 1, Coin = 2 | 0 | 1 | 7 | 7 | 7 | 7 | 7 |
Position 1 not updated because the coin 2 is bigger than the amount 1

Amount = 2, Coin = 1 | 0 | 1 | 2 | 7 | 7 | 7 | 7 |
Position 2 updated from 7 to 2

Amount = 2, Coin = 2 | 0 | 1 | 1 | 7 | 7 | 7 | 7 |
Position 2 updated from 2 to 1

Amount = 3, Coin = 1 | 0 | 1 | 1 | 2 | 7 | 7 | 7 |
Position 3 updated from 7 to 2

Amount = 3, Coin = 2 | 0 | 1 | 1 | 2 | 7 | 7 | 7 |
Position 3 updated from 2 to 2

Amount = 4, Coin = 1 | 0 | 1 | 1 | 2 | 3 | 7 | 7 |
Position 4 updated from 7 to 3

Amount = 4, Coin = 2 | 0 | 1 | 1 | 2 | 2 | 7 | 7 |
Position 4 updated from 3 to 2

Amount = 5, Coin = 1 | 0 | 1 | 1 | 2 | 2 | 3 | 7 |
Position 5 updated from 7 to 3

Amount = 5, Coin = 2 | 0 | 1 | 1 | 2 | 2 | 3 | 7 |
Position 5 updated from 3 to 3

Amount = 6, Coin = 1 | 0 | 1 | 1 | 2 | 2 | 3 | 4 |
Position 6 updated from 7 to 4

Amount = 6, Coin = 2 | 0 | 1 | 1 | 2 | 2 | 3 | 3 |
Position 6 updated from 4 to 3

Return 3

2.4 Complexity

Given the fact that we use an array to store intermediate steps for the computation, our algorithm has a space complexity of $\mathcal{O}(S)$, where s is the amount value to be calculated. Also, our algorithm doesn't compute any redundant operation, due to the fact that it gets the previously calculated value from an array, which has constant access time. To get the final result, we perform a double loop, one for the length of the coin array, that we can consider of length n, and one for each amount from 1 to S, so the time complexity of our algorithm is equal to $\mathcal{O}(n * S)$