Project Proposal-Revised

Problem: There are two players- A (Myself) and B (My friend) playing a game of coins lined up in front of us. Each of us gets alternate turns to remove coin either from the left-most side or right most side. The winner of the game is the player who has a higher score which is the sum of the coins which we collect in our pile. Assuming both myself and my friend is playing optimally, and I am given the chance to start the game, how much money am I guaranteed to win even if my friend is playing optimally. (This is problem 3 from the suggested problems)

Input: n: number of coins

coin: a list of length n. i and j represents left and right most coin value, also, low, and high in case of greedy approach represent the left and right most coin value of the list coin.

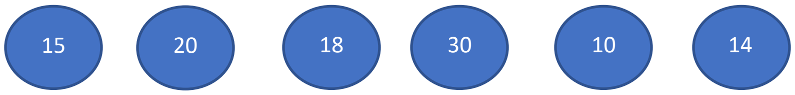
Output: The amount, Player A(myself) is guaranteed to win.

For greedy approach, the amount for both Player A and B are shown as in some cases the greedy method does not produce the optimal solution and there are cases where Player B wins.

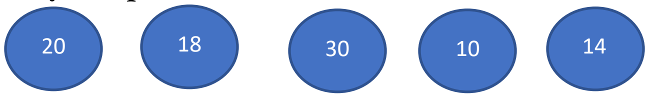
Assumption: To keep the game fair, assume there is a row of even number of coins so that the starting player does not get to choose an extra coin.

Scenario I:

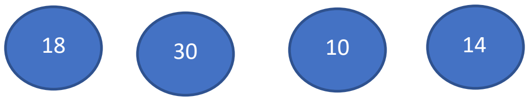
Input:



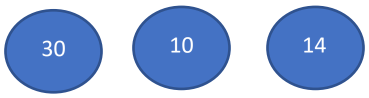
Player A picks: 15



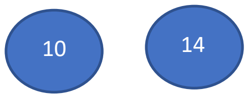
Player B picks: 20



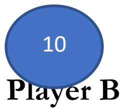
Players A picks: 18



Player B picks: 30



Player A picks: 14



Player B picks: 10

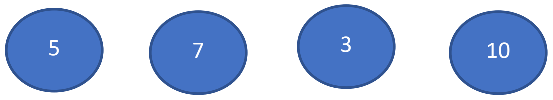
The **Total** score by **Player A:** (15 + 18 +14) = **47**

The **Total** score by **Player B:** (20 + 30 + 10) = **60**

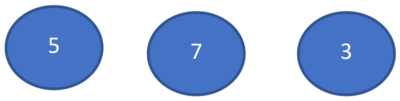
The Total score of Player B is more compared to Player A. So, the second player wins.

Scenario II:

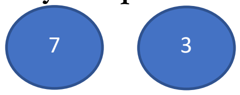
Input:



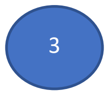
Player A picks: 10



Player B picks: 5



Player A picks: 7



Player B picks: 3

The **Total** score by **Player A:** (10 + 7) = 17

The **Total** score by **Player B:** (5 + 3) = 8

The Total score of Player A is more compared to Player B. So, the first player wins.

Brute Force Approach (Revised)- As per feedback, I implemented my previous brute force approach to my greedy approach. And the current (revised) brute force is implemented with reference to the solution provided.

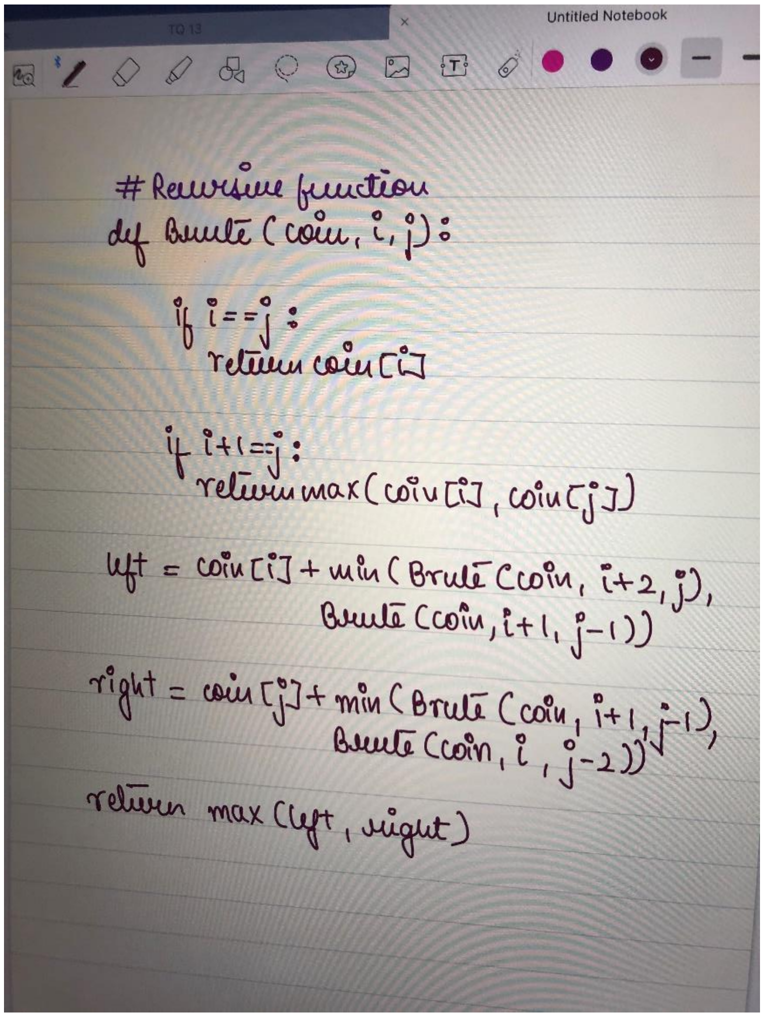
The idea here is to find the optimal strategy that would make Player A win, knowing that the opponent is playing optimally. Player A has two choices for coin [i…j], where i and j denote the left and right of the line, respectively.

1. If Player A chooses the left coin i, the opponent (Player B) is left to choose from [i+1, j].
    - If Player B chooses left coin i + 1, recur for [i+2, j]
    - If Player B chooses right coin j, recur for [i +1, j -1]
2. If Player A chooses the right coin j, then Player B is left to choose from [i, j-1]
    - If Player B chooses the left coin i, recur for [i+1, j-1]
    - If Player B chooses the right coin j-1, recur for [i, j-2]

The opponent (Player B) is playing optimally, he/she will try to minimize the player's (Player A) point i.e., the opponent will make a choice that will leave the player with minimum coins.

optimalsolution(i,j)=	**coin[i], (if i=j)**
	max(coin[i], coin[j]) (if i+1=j)
	max(coin[i], min (optimalsolution (coin, i + 2, j), optimalsolution (coin, i + 1, j – 1)), coin[j] + min (optimalsolution (coin, i +1, j – 1), optimalsolution (coin, i, j-2)))

Pseudocode:


TQ 13
Untitled Notebook

```
#Recursive function
def Brute(coin, i, j):

    if i==j:
        return coin[i]

    if i+1==j:
        return max(coin[i], coin[j])

left = coin[i] + min(Brute(coin, i+2, j),
                     Brute(coin, i+1, j-1))

right = coin[j] + min(Brute(coin, i+1, j-1),
                      Brute(coin, i, j-2))

return max(left, right)
```

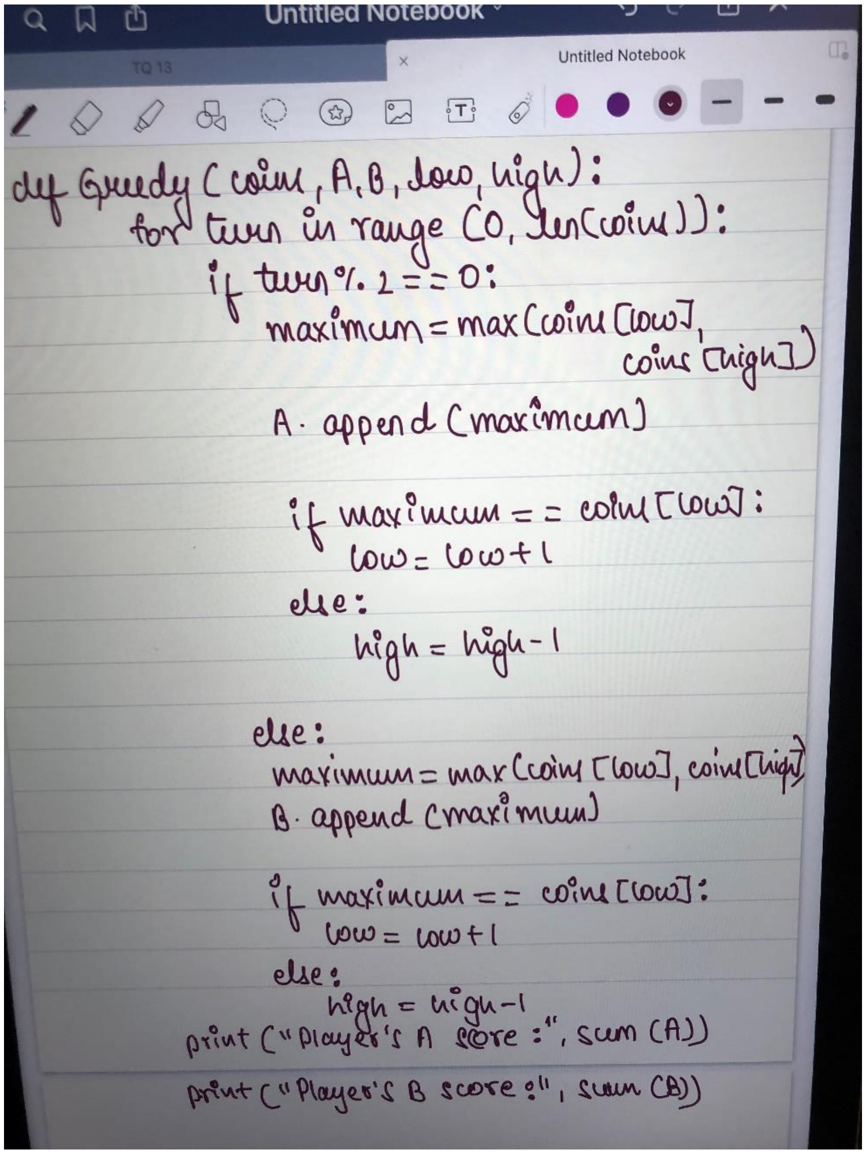
Time Complexity: The time complexity of the above approach is exponential, O(2^n) and occupies space in the call stack. So, it takes long time to run this on a list of size more than 10.

Problem with Brute force approach:

The problem has optimal substructure also the problems can be broken down into smaller subproblems, which can be further broken down. There are overlapping subproblems, so basically, we are solving same subproblem over and over.

Greedy Approach (Revised)-: **I applied my previous brute force approach to my greedy approach as suggested.**

In this approach, both players A and B choose the coin which has the maximum value during their turn. They keep on repeating this approach until no coins are left.



```
def Greedy (coins, A, B, low, high):
    for turn in range (0, len(coins)):
        if turn % 2 == 0:
            maximum = max (coins [low],
                           coins [high])
            A. append (maximum)

            if maximum == coins [low]:
                low = low + 1
            else:
                high = high - 1

        else:
            maximum = max (coins [low], coins [high])
            B. append (maximum)

            if maximum == coins [low]:
                low = low + 1
            else:
                high = high - 1
    print ("Player's A score :", sum (A))
    print ("Player's B score :", sum (B))
```

Time Complexity: O(n), since there is single for loops, not the nested one

Problem with Greedy approach:

This approach is not optimal as it only takes the current values into consideration and ignores the value which could be available in the next move. It does not guarantee a win or maximize the winnings of Player A. Taking a maximum at a particular point might unlock a much bigger number for Player B to choose and this would allow Player B to win.

Dynamic Approach:

This approach would make sure that Player A wins with the maximum possible score even when Player B would play optimally. Both the players will try to reduce the chances of winning of the opponent and will be able to do so with the following approaches, F(i,j) represent the maximum value, Player A can collect from the ith coin to jthcoin

1. Player A chooses the ith coin, value $V_{i`}$, then Player B can either choose (i+1) th coin or jth

coin. The Player B is also playing optimally and intends to choose the coin which would leave the Player A with minimum value. The Player A can collect the value V_i + min (F (i+2, j), F (i+1,j-1) where [i+2, j] is the range of array indices available to Player A if Player B chooses V_i + 1 and [i+1, j-1] is the range of array indexes available if opponent chooses the jth coin.

2. The Player A can even choose the jth coin with the value $V_{j,}$ the Player B either chooses ith coin or (j-1) th coin. The Player B intends to choose the coin which leaves the Player A with minimum value, i.e. the Player A can collect the value V_j +min(F(i+1,j-1), F(I,j-2)) where [i,j-2], is the range of array indices available for the Player A if the Player B picks the jth coin and [i+1,j-1] is the range of indices available to the Player A if the Player B picks up the ith coin.

The recursive solution based on above two choices; we would be taking the maximum of two choices:

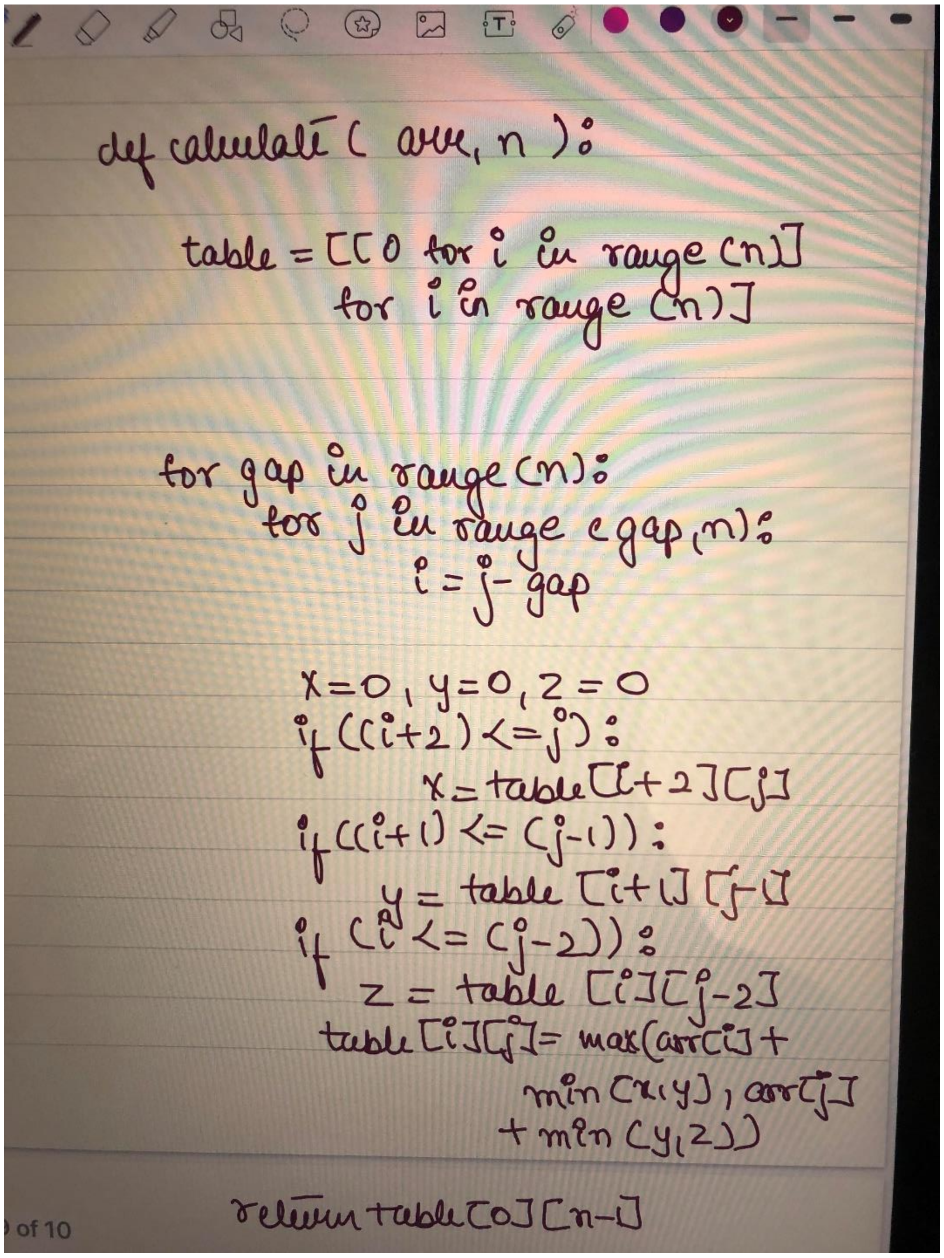
$F(i,j) = \text{Max}(V_i + \min(F(i + 2, j), F(i + 1, j – 1)), V_j + \min(F(i + 1, j -1), F(i, j-2))$

Base Cases:

$F(i,j) = V_i$, if j==i

$F(i,j) = \max(V_{i,} V_j)$, if j==i+1

Using Bottom-up approach, Pseudocode:



```
def calulate ( arr, n ):

    table = [[0 for i in range (n)]
              for i in range (n)]


    for gap in range (n):
        for j in range (gap, n):
            i = j - gap

            x = 0, y = 0, z = 0
            if ((i+2) <= j):
                    x = table[i+2][j]
            if ((i+1) <= (j-1)):
                    y = table [i+1] [j-1]
            if (i <= (j-2)):
                    z = table [i][j-2]
            table[i][j] = max(arr[i] +
                      min (x, y), arr[j]
                      + min (y, z))

    return table[0][n-1]
```

Time Complexity: There is use of two nested loop because of which the time complexity would be O(n^2)