Lecture 28 : Stocks II

We are given an array 'prices' where price of stocks are given, we can at a time buy one stock and we have to sell it before buying another stock, we can repeat this process as many times as want, to maximise the ~~result~~ profit

prices[] = [7, 1, 5, 3, 6, 4]

Output = 7 [(1, 5) + (3, 6)]

Now try to understand and observe that every stock we have two option Pick / Not pick. But we need an extra variable which will tell us, if we can ~~by~~ buy the current stock on that particular day or not.

We have to maximize our output, thus we take maximum of all cases.

Think if we are buying a stock we have to SUBTRACT that stock's price from ~~as~~ ans. ~~Add~~ And we'll ADD when we sell.

Maintain a ~~by~~ 'buy' variable if buy = 1, purchase it (or) ignore it, else buy = 0 sell it (or) Don't sell it.

Base if (ind == n), then irrespective we have bought (or) sell the stock, 'return 0' for no loss.

Note : After every call keep updating the 'buy' variable, accordingly.

Memoisation

```
int f(int ind, int buy, vector<int>& arr,
                        vector<vector<int>>& dp) {

    if(ind == arr.size())
        return 0;
    if(dp[ind][buy] != -1)
        return dp[ind][buy];

    if(buy == 1)
        dp[ind][buy] = max(-arr[ind] + f(ind+1, 0, arr, dp),
                           f(ind+1, 1, arr, dp));
    else
        dp[ind][buy] = max(arr[ind] + f(ind+1, 1, arr, dp),
                           f(ind+1, 0, arr, dp));

    return dp[ind][buy];
}
```

Time Complexity = O(N x 2)

Space Complexity = O(N x 2) + O(N)

Tabulation

Same procedures ① Base Case Manipulation.
② Write variables (i, buy)
③ Copy Recurrence.

```
int solve ( vector<int> arr ){
    int n= arr.size();
    vector<vector<int>> dp(n+1, vector<int>(2,0));

    dp[n][0]=0 , dp[n][1]=0; // base cases

    for(int i=n-1; i>=0; i++){
        for(int j=0; j<=1; j++){
            if(j==1)
                profit = max(-arr[i]+dp[i][0], dp[i][i]
            else
                profit= max(arr[i]+ dp[i][1], dp[i][0]);

                dp[i][j]= profit;
        }
    }
    return dp[0][1];
}
```

Time comp → O(N x 2)
Space comp → O(N x 2).

We can space optimize it by using prev and curr.

Lecture 29: Stocks III

The question is exactly the same as last question the only difference is that we can ~~one~~ (buy and sell) stocks at max '2' times.

That means we are allowed to perform transactions only twice

1 Transaction = ~~By~~ (Buy And Selling) of Stock.

Thus we can use the same and just add one more variable like the knapsack problem. To keep a track of No of Transactions made.
Try to understand only the case : when we had bought a stock, and when we are planning to sell it only then it is counted as A TRANSACTION, otherwise it won't

Base Case will add one more case when (CAP == 0)
That means we have reached the limit to purchase and thus we can return 0;

Time Complexity = $O(N \times 2 \times 3)$
Space Complexity = $O(N \times 2 \times 3) + O(N)$
↓
Auxiliary Stack Space.

Memoization

Create a dp[][][] to store ans.

```
int f(int ind, int buy, int cap, vector<int>& arr,
                    vector<vector<vector<int>>>& dp) {

    if (ind==n || cap==0)
        return 0;

    if(buy==1)
        dp[ind][buy][cap] = max(-arr[ind] + f(ind+1, 0, cap, arr,
                                              dp),
                                f(ind+1, 1, cap, arr, dp));
    else
        dp[ind][buy][cap] =
            max(arr[ind]+f(ind+1, 1, cap-1, arr, dp),
                f(ind+1,0,cap, arr, dp));

    return dp[ind][buy][cap];
};
```

Tabulation.

Same Rules : ① Base Case.
② Variables (i, buy, cap)
③ Copy Recurrence

Time Complexity — O(N x 2 x 3).
Space Complexity — O(N x 2 x 3).

```
int Solve (vector<int> & arr) {
    int n = arr.size();
    vector<vector<vector<int>>> dp(n+1, vector<vector<int>>(3,
                                   vector<int>(3, 0)));

    for(int i = n-1 ; i >= 0 ; i--) {
        for(int j = 0; j <= 1 ; j++) {
            for(int z = 1 ; z <= 2 ; z++) {
                if (j)
                    dp[i][j][z] = max(-arr[i] + dp[i+1][0][z],
                                      dp[i+1][1][z]);
                Else.
                    dp[i][j][z] = max(arr[i] + dp[i+1][1][z-1],
                                      dp[i+1][0][z] );
            }
        }
    }
    return dp[0][1][2]; // original State where we
                        made F calls.
}
```

If the Question is moulded, and we are given 'K' transactions instead of '2', just replace values and everything else will remain the same.

Space optimize by using curr[][], prev[][]

Lecture 30 → Buy & Sell Stocks with a Cooldown.

Complete same as Stock II, it's just that after a complete transaction, we have to serve a cooldown period of 1 stock.

prices[] = [1, 2, 3, 0, 3]
output = 3
↳ (2-1) cooldown[3] (0, 3)

So, in the same code in Stock II, in the else condition where we complete the TRANSACTION by selling we just do ind+2.

```
int f(int ind, int buy, vector<int>& arr) {
    if (ind >= n) → change
        return 0;
    if (dp[ind][buy] != -1)
        return dp[ind][buy];

    if (buy)
        dp[ind][buy] = max(-arr[ind]+f(ind+1, 0, arr),
                           f(ind+1, 1, arr));
    else
        dp[ind][buy] = max(arr[ind] + f(ind+2, 0, arr),
                           f(ind+1, 0, arr));

    return dp[ind][buy];
}
```

only change