07/06/2023

Q1 Coin change problem (Leetcode 322)

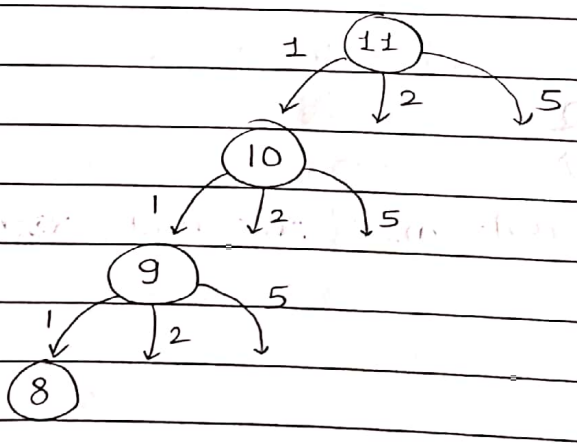
i/p → coins = [1, 2, 5]

amount = 11

o/p → 3 (5 + 1 + 5)

↑ minimum no. of coins

If we are not able to make the amount, we have to return -1.

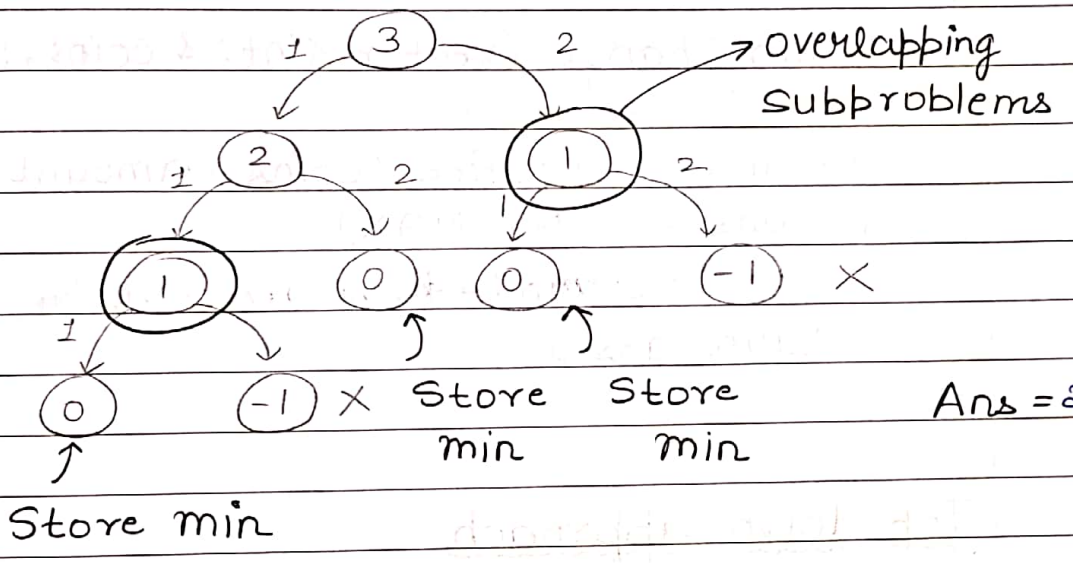


In short at each node there are calls which is equal to the size of coins. Here we have 3 coins and hence 3 calls are there for each node. We can say that we have to run a for loop from 0 to no. of coins.

Dry run

It is important to note that once the amount becomes 0, we have successfully got the answer but we need to store the minimum no. of coins.

coins → [1, 2]
target / amount = 3



Pattern Recursive call for each and every value of the coins array (Simple for loop pattern)

Recursive code

```
int solveRec (vector <int> & coins, int amount) {
    // If we have to make 0 amount, we need 0 coins
    if (amount == 0)
        return 0;
    if (amount < 0)
    // Mark this as from here we won't get answer
        return INT_MAX;
    int mini = INT_MAX;
    for (int i=0; i< coins.size(); i++) {
        int ans = solveRec (coins, amount - coins[i]);
        if (ans != INT_MAX) {
        // Update mini as we have a valid answer
    To include previous coin →
        mini = min (mini, 1+ans);
    }
```

```
    }
    return mini;
}
int coinChange(vector<int>& coins, int amount){
    //Call the function
    int ans = solveRec(coins, amount);
    if(ans == INT_MAX)
        return -1; //Directed by question
    return ans;
}
```

Top down approach

```
int solveTopDown(vector<int>& coins, int
amount, vector<int>& dp) {

    if(amount == 0)
        return 0;
    if(amount < 0)
        return INT_MAX;
    //Step 3: Check if ans already exists
    if(dp[amount] != -1)
        return dp[amount];
    int mini = INT_MAX;
    for(int i=0; i<coins.size(); i++){
        int ans = solveTopDown(coins,
        amount - coins[i], dp);
        if(ans != INT_MAX) {
            mini = min(mini, 1+ans);
        }
    }
    //Step 2: Update in dp array
```

```
    dp[amount] = mini;
    return dp[amount];
}
int coinChange(vector<int>& coins, int amount){
    //Step 1: Create dp array
    vector<int> dp(amount+1, -1);
    int ans = solveTopDown(coins, amount, dp);
    if(ans == INT_MAX)
        return -1;
    return ans;
}
```

→ amount changing in recursion

TC = O(amount)
SC = O(n+n)

Bottom up approach (Tabulation)

We need to know by which value should we initialize the dp array here. dp array is containing minimum no. of coins & hence store INT_MAX initially.

```
int solveTab(vector<int>& coins, int amount){
    //Step 1: Create dp array
    vector<int> dp(amount+1, INT_MAX);
    //Step 2: Observe base case of top-down
    dp[0] = 0;
    //Step 3: Reverse flow in top-down approach
    for(int i=1; i<=amount; i++){
        for(int j=0; j<coins.size(); j++){
            if(i-coins[j]>=0 && dp[i-coins[j]]!=INT_MAX)
            {
                int ans = dp[i-coins[j]];
                dp[i] = min(dp[i], 1+ans);
            }
        }
    }
```

↑ Valid index

Ignore INT_MAX ans ↑

```
        }
    }
    return dp[amount];
}

int coinChange(vector<int>& coins, int amount){

    int ans = solveTab(coins, amount);
    if(ans == INT_MAX)
        return -1;
    return ans;
}
```

Dry run

coins → [1, 2, 5]
amount → 7

0	1	1	2	2	1	2	2
0	1	2	3	4	5	6	7

dp[0] = 0

dp[1] = min(0+1, X, X) = 1 ← negative index

negative index

dp[2] = min(1+1, 0+1, X) = 1
dp[3] = min(1+1, 1+1, X) = 2
dp[4] = min(1+2, 1+1, X) = 2
dp[5] = min(2+1, 2+1, 0+1) = 1
dp[6] = min(1+1, 2+1, 1+1) = 2
dp[7] = min(2+1, 1+1, 1+1) = 2

Final answer

$dp(i) = min(x, y, z)$;

$x \rightarrow$ go one index behind
$y \rightarrow$ go two index behind
$z \rightarrow$ go five index behind

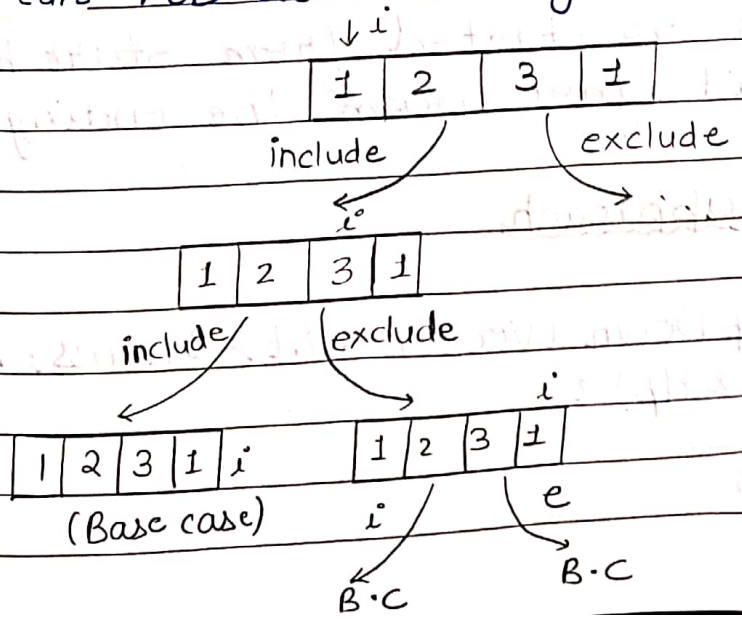
$TC = O(amount \times n)$ ← coins.size()
$SC = O(n)$

Space optimization
Here space optimization is not possible as dp[i] depends on dp[i - coins[j]] and coins[j] is not a certain value. Here we can't decide no. of variables.

Q2 House robber problem (Leetcode 198)

i/p → [1, 2, 3, 1]
o/p → 4 (1+3 = 4)

A robber can't rob adjacent houses on same night. We have to return maximum money robber can rob at same night.



include ⇒ i moves by 2 index
exclude ⇒ i moves by 1 index

Recursive code

```
int solveRec (vector<int>& nums, int n){
    // Base case : Invalid house & hence nothing to steal
    if (n < 0)
        return 0;
    if (n == 0)
        return nums[0];
    // Include call
    int include = nums[n] + solveRec(nums, n-2);
    //Exclude call
    int exclude = 0 + solveRec(nums, n-1);
    return max(include, exclude);
}
int rob (vector<int>& nums){
    int n = nums.size() - 1;
    return solveRec(nums, n);
}
```

Note → In dry run we started from start but in code we did start from the ending index.

Top down approach

```
int solveTopDown (vector<int>& nums, int n, vector<int>& dp){
    // Base case
    if (n < 0)
```

```
    return 0;
if (n == 0)
    return nums[0];
// Check if answer exists already (Step - 3)
if (dp[n] != -1)
    return dp[n];
// Include call
int include = nums[n] + solveTopDown(nums, n-2,
dp);
// Exclude call
int exclude = 0 + solveTopDown(nums, n-1, dp);
// Step 2 : Store ans in dp array
dp[n] = max(include, exclude);
return dp[n];
}

int rob (vector<int> &nums) {

int n = num.size() - 1;  → n changing in recursion
vector<int> dp(n+1, -1);
return solveTopDown(nums, n, dp);
}
```

TC = O(n)
SC = O(n+n)

Bottom up approach

```
int solveTab(vector<int> &nums, int n) {
    // Step 1 : Create dp array
vector<int> dp(n+1, 0);  → will work with INT_MIN also
// Step 2 : Observe base case of top down
dp[0] = nums[0];
// Step 3 : Reverse flow of top-down
for (int i=1; i<=n; i++) {
```

```
        int temp = 0;
        if (i-2 >=0)
            temp = dp[i-2];
        int include = temp + nums[i];
        int exclude = 0 + dp[i-1];
        dp[i] = max(include, exclude);
    }
    return dp[n];
}
int rob(vector<int>& nums) {
    int n = nums.size() - 1;
    return solveTab(nums, n);
}
```

TC = O(n)

Dry run

nums → | 1 | 2 | 3 | 1 |
0 1 2 3

dp[i] → money earned coming upto ith house.

n = 3

We have to create dp array of size = 4.

1	2	4	4
0	1	2	3

dp[0] = 1 → nums[0] (Base case)

dp[1] = max(2, 1) = 2

↑ 0 + nums[1] ↖ 0 + dp[0]

dp[2] = max(1+3, 2) = 4

temp ↗ ↑ nums[2] ↖ 0 + dp[1]

$dp[3] = max(2+1, 0+4) = 4$

temp = 2, nums[3] = 1, dp[2] = 4

Space optimization

Here dp[i] depends on dp[i-1] and dp[i-2]. Hence we can simply do with the help of 3 variables.

```
int spaceOpt(vector<int> &nums, int n){
    int prev2 = 0;
    int prev1 = nums[0];
    int curr = 0;
    for(int i=1; i<=n; i++){
        int temp = 0;
        if(i-2 >= 0)                    // -> dp[i-2]
            temp = prev2;
        int include = temp + nums[i];
        int exclude = 0 + prev1;        // -> dp[i-1]
        curr = max(include, exclude);   // updating variables
        prev2 = prev1;
        prev1 = curr;
    }
    return prev1;
}
```

TC = O(n)
SC = O(1)

Dry run

1	2	3	1
0	1	2	3

* i = 1

prev2 = 0, prev1 = 1

include = 2
exclude = 1
curr = 2

* i = 2
prev2 = 1 , prev1 = 2
temp = 1
include = 4
exclude = 2
curr = 4

* i = 3
prev2 = 2 , prev1 = 4
temp = 2
include = 3
exclude = 4
curr = 4

Now prev2 = 4, prev1 = 4 & hence we need to return prev1.