Dynamic Programming - 1

- Karun Karthik

Contents

Introduction

Dynamic programming is a technique to solve problems by breaking it down into a colle ction of sub-problems, solving each of those sub-problems just once and storing these solutions inside the cache memory in case the same problem occurs the next time.

Dynamic Programming is mainly an optimization over plain recursion .
Wherever we see a recursive solution that has repeated calls for same inputs, we can o ptimize it using Dynamic Programming.
This simple optimization reduces the time complexities from exponential to polynomial.

There are two different ways to store our values so that they can be reused at a later i nstance. They are as follows:
1. Memoization or the Top Down Approach.
2. Tabulation or the Bottom Up approach.

In Memoization we start from the extreme state and compute result by using values th at can reach the destination state i.e the base state.

In Tabulation we start from the base state and then compute results all the way till the extreme state.

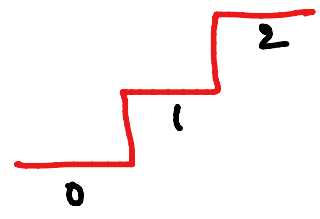
Note: To store the intermediate results we can use Array, Matrix, Hashmap etc., all we need is data storage and retrieval with a specific key.

How to find the use case of Dynamic Programming?

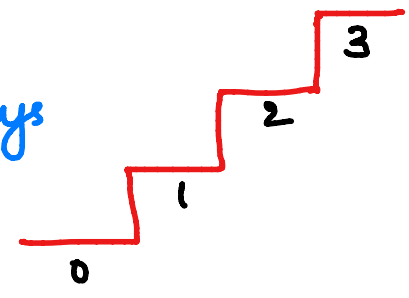
You can use DP if the problem can be,
1. Divided into sub-problems
2. Solved using a recursive solution
3. Containing repetitive sub-problems

① Climbing Stairs → Given a value 'N', find the number of ways to reach N & jumps possible are ONE or TWO.

Eg n = 2 ⇒ $0 \xrightarrow{1} 1 \xrightarrow{1} 2$; $0 \xrightarrow{2} 2$ } for N = 2 we have 2 ways



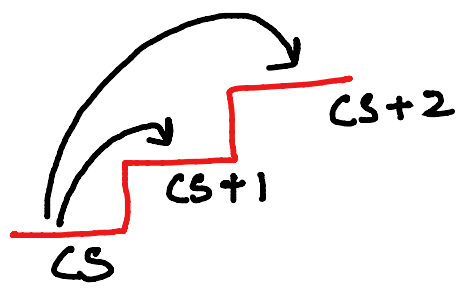
n = 3 ⇒ $0 \xrightarrow{1} 1 \xrightarrow{1} 2 \xrightarrow{1} 3$; $0 \xrightarrow{1} 1 \xrightarrow{2} 3$; $0 \xrightarrow{2} 2 \xrightarrow{1} 3$ } for N = 3 we have 3 ways

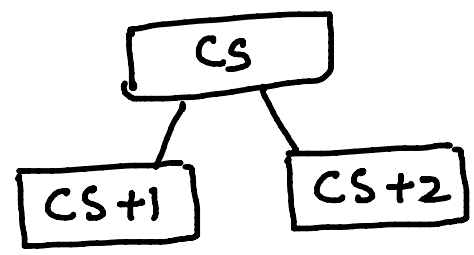


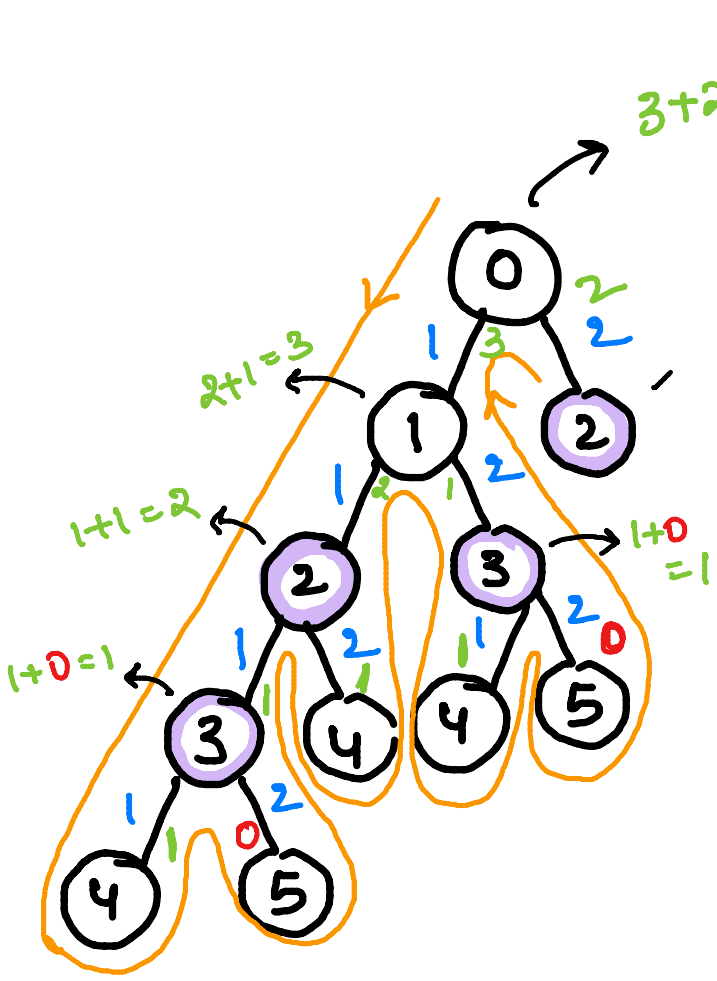
n = 4

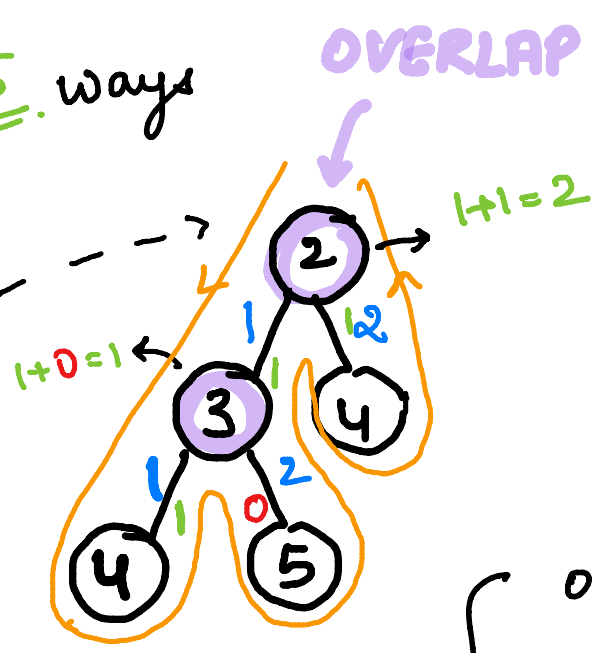
↳ for every stair we have 2 cases ie

⇒









if CS == n
return 1

if CS > n
return 0

5 ways

$0 \xrightarrow{1} 1 \xrightarrow{1} 2 \xrightarrow{1} 3 \xrightarrow{1} 4$

$0 \xrightarrow{1} 1 \xrightarrow{1} 2 \xrightarrow{2} 4$

$0 \xrightarrow{1} 1 \xrightarrow{2} 3 \xrightarrow{1} 4$

$0 \xrightarrow{2} 2 \xrightarrow{1} 3 \xrightarrow{1} 4$

$0 \xrightarrow{2} 2 \xrightarrow{2} 4$

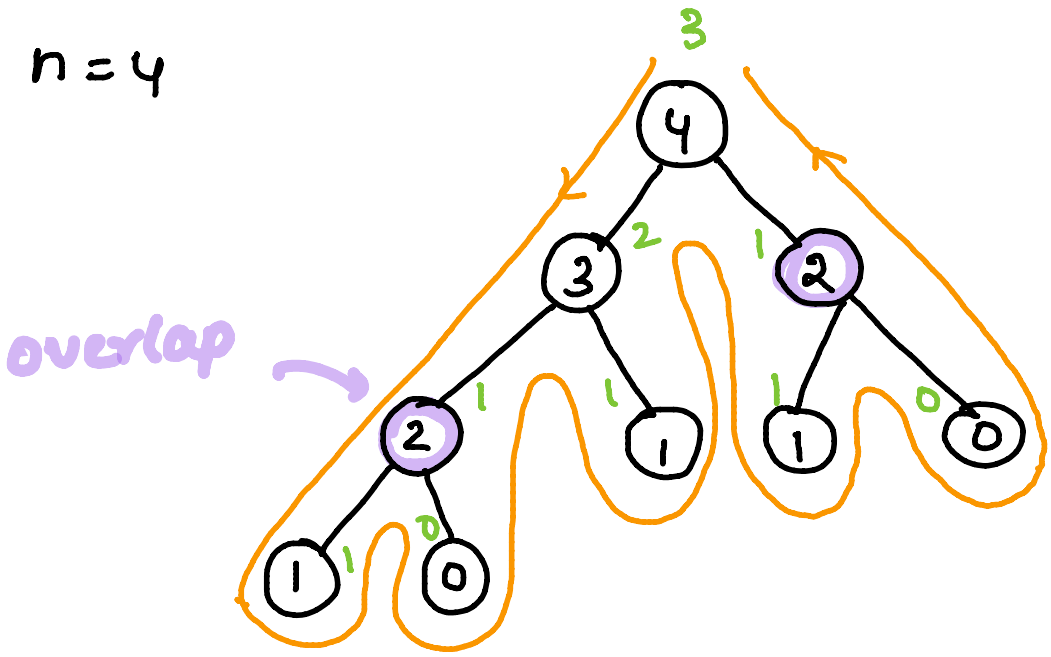
* Here we can see for ②, ③ the subproblem is being done multiple times, we can solve using dp.

Code →

```cpp
class Solution {
public:
    int totalWays(int currentStair, int targetStair, unordered_map<int,int> &memo){

        if(currentStair==targetStair){
            return 1;
        }

        if(currentStair > targetStair){
            return 0;
        }

        int currentKey = currentStair;

        if(memo.find(currentKey)!=memo.end()){
            return memo[currentKey];
        }

        int oneStep = totalWays(currentStair+1, targetStair, memo);
        int twoStep = totalWays(currentStair+2, targetStair, memo);

        memo[currentKey] = oneStep+twoStep;

        return oneStep+twoStep;

    }

    int climbStairs(int n) {
        unordered_map<int,int> memo;
        return totalWays(0,n,memo);
    }
};
```

② Fibonacci Number → $f(n) = f(n-1) + f(n-2)$ & $f(0) = 0$, $f(1) = 1$, $n \geq 0$.

Eg → $n = 4$



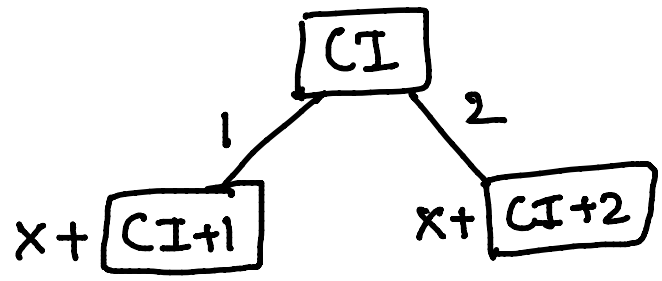
Code →

```cpp
class Solution {
public:
    int helper(int n, unordered_map<int,int>&memo){

        if(n<=1){
            return n;
        }

        int currentKey = n;

        if(memo.find(currentKey)!=memo.end()){
            return memo[currentKey];
        }


        int a = helper(n-1,memo);
        int b = helper(n-2,memo);

        memo[currentKey] = a+b;
        return memo[currentKey];
    }


    int fib(int n) {
        unordered_map<int,int>memo;
        return helper(n,memo);
    }
};
```

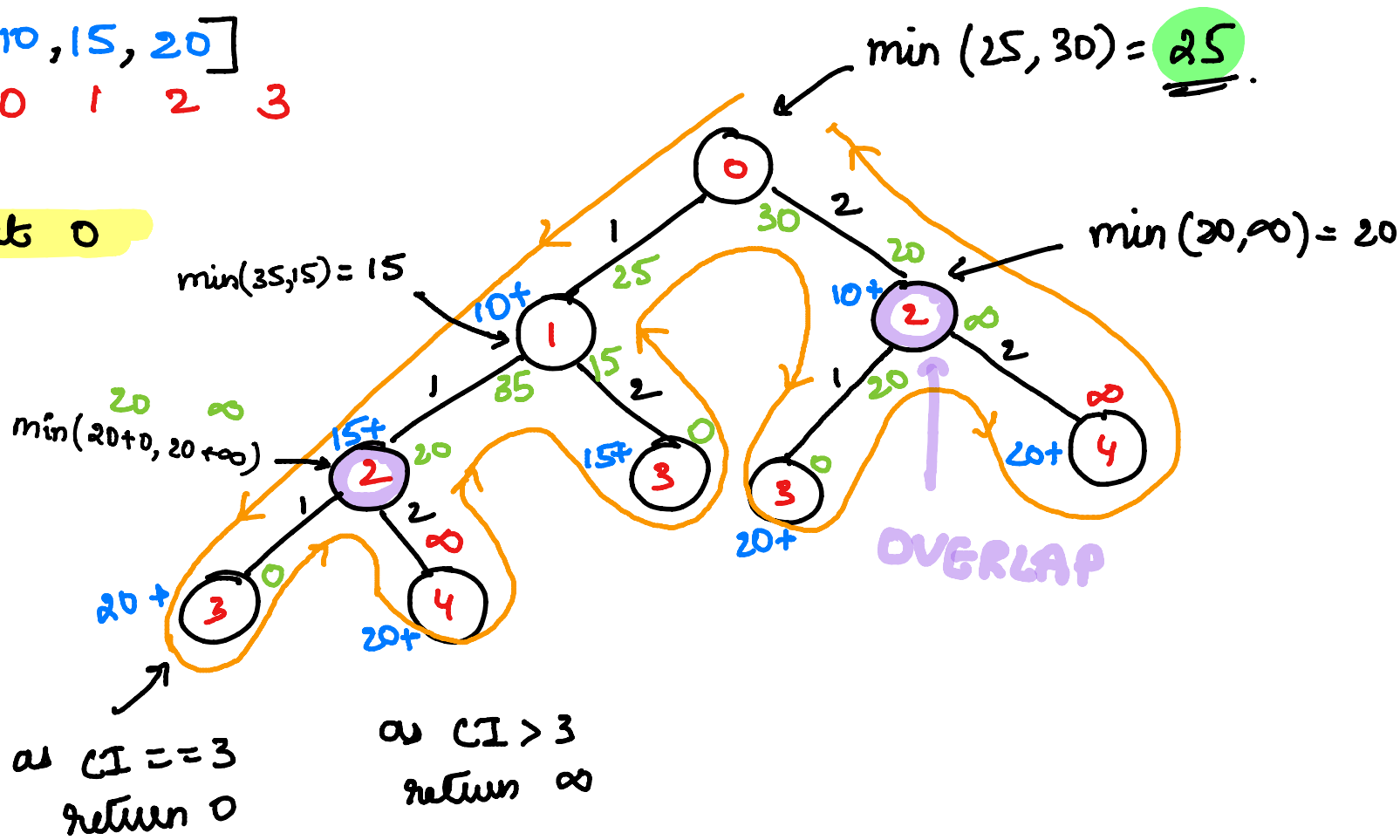
③ Min Cost Climbing Stairs →

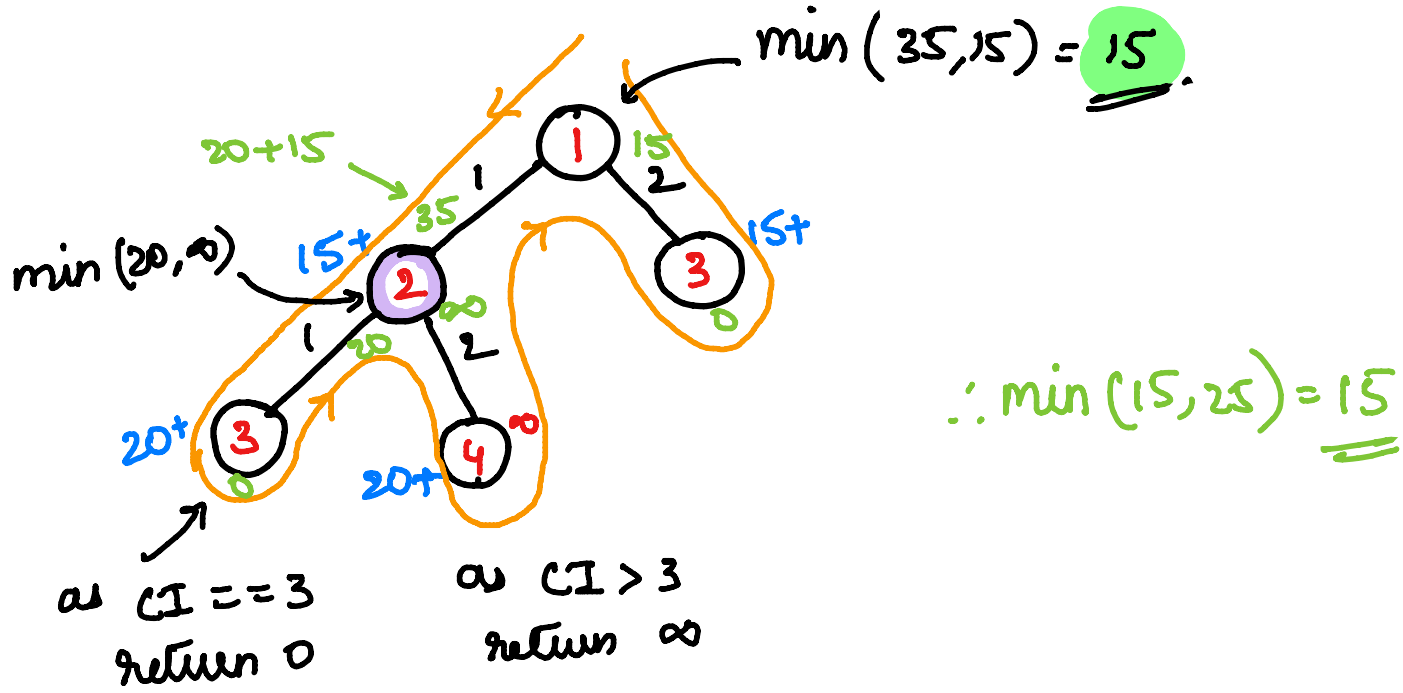
Given costs array, find min cost to reach the end, starting from 0 or 1 & making 1 or 2 jumps.

$\therefore$ costs = [_ _ X _ _ _]
CI



Eg cost = [10, 15, 20]
0 1 2 3





$\therefore$ min (15, 25) = 15

Code →

```cpp
class Solution {
public:

    int minCost(vector<int>&cost, int currentIndex, unordered_map<int,int> &m){

        if(currentIndex == cost.size()){
            return 0;
        }

        if(currentIndex > cost.size()){
            return 1000;    // large values, serves as INFINITY
        }

        if(m.find(currentIndex)!=m.end()){
            return m[currentIndex];
        }

        int oneJump = cost[currentIndex] + minCost(cost,currentIndex+1, m);
        int twoJump = cost[currentIndex] + minCost(cost,currentIndex+2, m);

        m[currentIndex] = min(oneJump, twoJump);
        return m[currentIndex];
    }

    int minCostClimbingStairs(vector<int>& cost) {
        unordered_map<int,int> m;
        return min( minCost(cost,0,m), minCost(cost,1,m));
    }
};
```

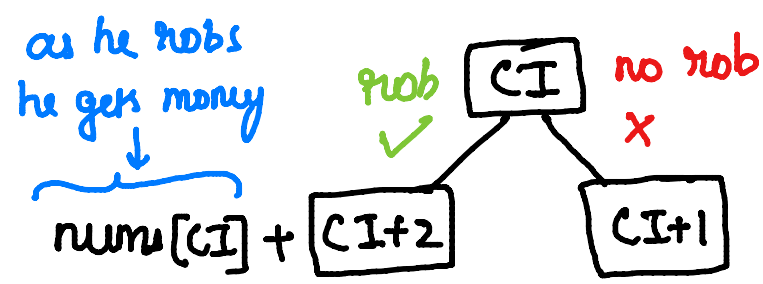
④ House Robber → Given an array of no. representing money, find max amount, that can be robbed without choosing the adjacent houses.

Eg nums = [2, 7, 9, 3, 1]
0 1 2 3 4

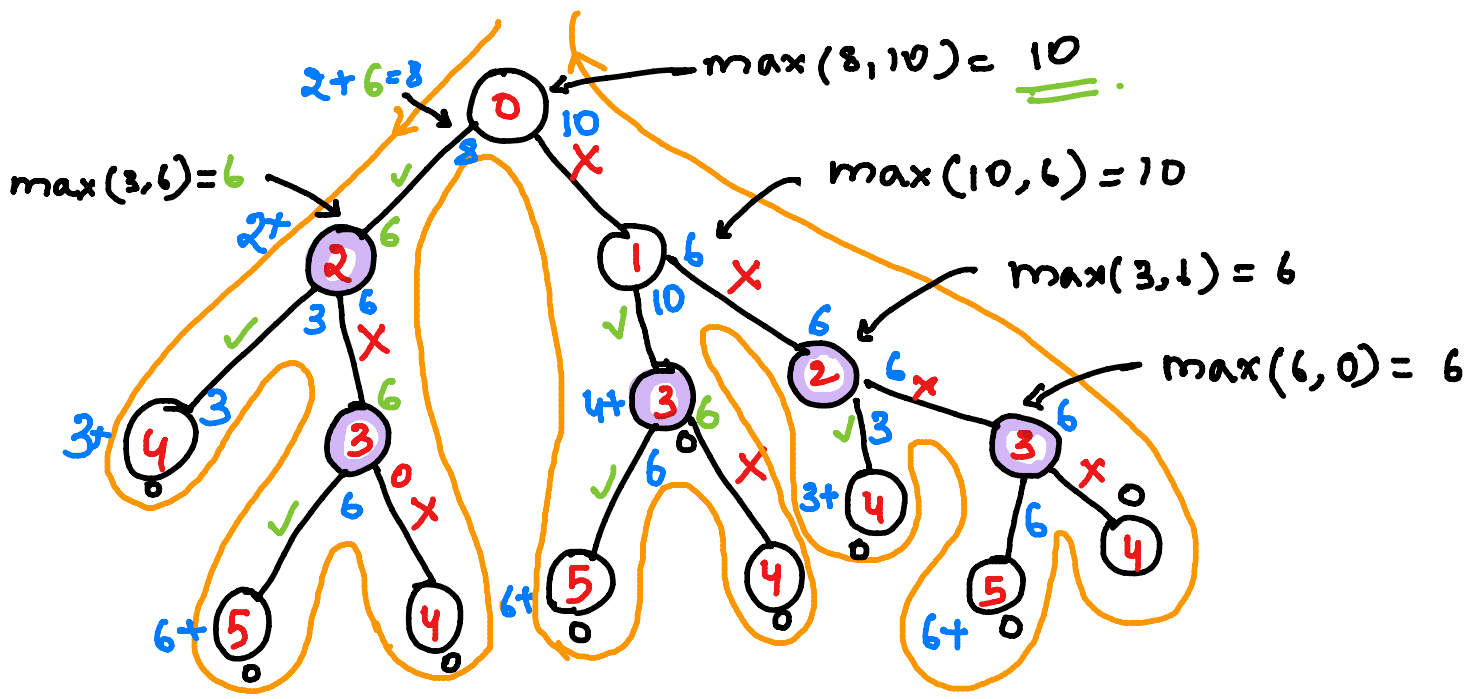
⇒ max amount = 2 + 9 + 1 = 12.
0 → 2 → 4

→ if robber robs house, then amount = nums[CI] & CI = CI + 2 (to avoid adjacent)
else CI = CI + 1

ie nums = [_ _ CI _ _ _ _]



Eg [2, 4, 3, 6]
0 1 2 3



as CI > 3
return 0

∴ at every node find max (left, right) & add it's value to the nums[CI] if selected, else continue.

code →

```cpp
class Solution {
public:

    int helper(vector<int>&nums, int currentIndex, unordered_map<int,int>&m){

        if(currentIndex >= nums.size()){
            return 0;
        }

        int currentKey = currentIndex;

        if(m.find(currentKey)!=m.end()){
            return m[currentKey];
        }

        int rob = nums[currentKey] + helper(nums, currentIndex+2, m);
        int noRob = helper(nums, currentIndex+1, m);

        m[currentIndex] = max(rob, noRob);

        return m[currentIndex];
    }

    int rob(vector<int>& nums) {
        unordered_map<int,int> m;
        return helper(nums,0,m);
    }
};
```

⑤ House Robber - II →

In this problem, the approach will be similar to previous one, but the houses are in circle, which means that

* if we start from 1st house, then we can't rob the last house.
* if we start from 2nd house, then we can rob the last house.
* and return max value between 1st house & 2nd house
* if only 1 house is present, then rob it directly.

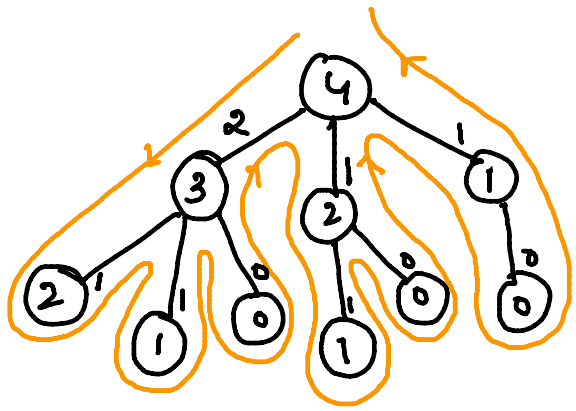
Code →

```cpp
class Solution {
public:

    int helper(vector<int>&nums, int currentIndex, int lastIndex, unordered_map<int,int>&m){

        if(currentIndex > lastIndex){
            return 0;
        }

        int currentKey = currentIndex;

        if(m.find(currentKey)!=m.end()){
            return m[currentKey];
        }

        int rob = nums[currentKey] + helper(nums, currentIndex+2, lastIndex, m);
        int noRob = helper(nums, currentIndex+1, lastIndex, m);

        m[currentIndex] = max(rob, noRob);

        return m[currentIndex];

    }

    int rob(vector<int>& nums) {

        int n = nums.size();
        if(n==1)    return nums[0];

        unordered_map<int,int> memo1,memo2;
        // we can start robbing from 2 houses
        int firstHouse = helper(nums, 0, n-2, memo1);
        int secondHouse = helper(nums, 1, n-1, memo2);
        return max(firstHouse, secondHouse);
    }
};
```

⑥ N-th Tribonacci → given n, find T_n

$T_{n+3} = T_n + T_{n+1} + T_{n+2}$ ∀ $n \geq 0$ $T_0 = 0$, $T_1 = 1$, $T_2 = 1$.

Eg n = 4



4 (2+1+1)

code →

```cpp
class Solution {
public:

    int helper(int n, unordered_map<int,int> &m){
        if(n<=1){
            return n;
        }

        if(n==2){
            return 1;
        }

        int currentNum = n;

        if(m.find(currentNum)!=m.end()){
            return m[currentNum];
        }

        int a = helper(n-1,m);
        int b = helper(n-2,m);
        int c = helper(n-3,m);

        m[currentNum] = a+b+c;

        return m[currentNum];
    }

    int tribonacci(int n) {
        unordered_map<int,int>m;
        return helper(n,m);
    }
};
```

(7) 0 - 1 Knapsack Problem → find max profit such that the weight of all items <= capacity.

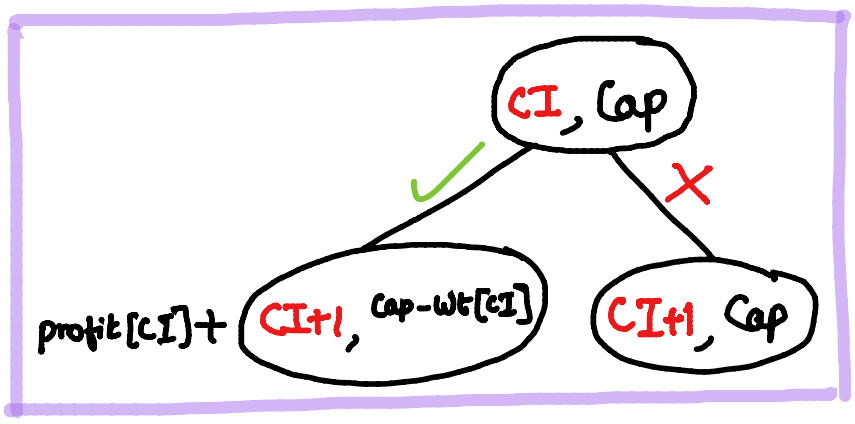
```
          0  1  2  3
wt      = [3, 4, 6, 5]
profits = [2, 3, 1, 4]
capacity = 8
```

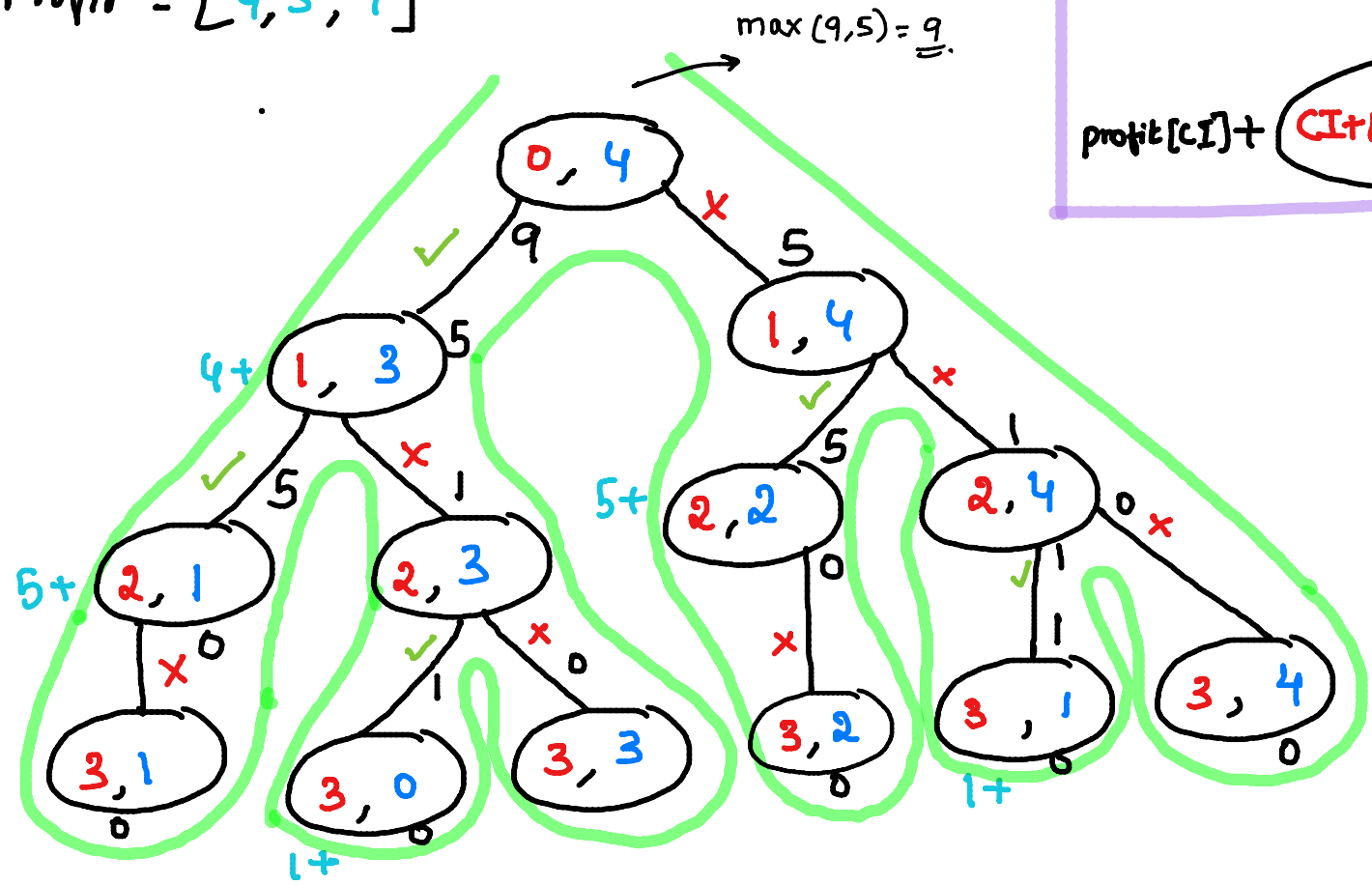
→ if we select 0 & 3,
then total weights = wt[0] + wt[3]
= 3 + 5 = 8.

& profits are 2 + 4 = 6 That's max profit possible

Eg

```
          0  1  2
Wt     = [1, 2, 3]    capacity = 4
Profit = [4, 5, 1]
```





∴ at every step,

→ if selecting an index then reduce capacity by wt[CI]
& add profit[CI] to result

→ if not selecting, increment CI by 1

→ find max(left, right)

code →

```cpp
class Solution
{
    public:

    int helper(int W, int wt[], int val[], int n, int curr,
                                unordered_map<string,int> &memo){
        if(curr==n) return 0;

        // Instead of Matrix we can use strings as unique keys
        string currKey = to_string(curr)+"_"+to_string(W);

        if(memo.find(currKey)!=memo.end()) return memo[currKey];

        int currWt = wt[curr];
        int currVal = val[curr];

        int selected = 0;
        if(currWt<=W){
            selected = currVal + helper(W-currWt, wt, val, n, curr+1, memo);
        }

        int notSelected = helper(W, wt, val, n, curr+1, memo);

        memo[currKey] = max(selected, notSelected);
        return memo[currKey];
    }


    int knapSack(int W, int wt[], int val[], int n)
    {
       unordered_map<string,int> memo;
       return helper(W, wt, val, n, 0, memo);
    }
};
```

(8) Partition Equal Subset Sum →

given an array, find if it can be divided into two subsets whose sum is equal.

Eg nums = [1, 5, 11, 5] can be divided into S1 = {1, 5, 5} & S2 = {11}
& sum of S1 == sum of S2 ∴ returns True.

∴ initially find sum of elements in array.

1) if sum is odd then return False

2) if sum is even, then proceed.

→ find a subset whose value == sum/2
which means that the other subset will have value == sum/2.

→ lets say ts = sum/2 (ts is target sum)

→ At every index, we have 2 choices

1) if we select then ts = ts − nums[CI] (CI: current index)
&
CI = CI + 1

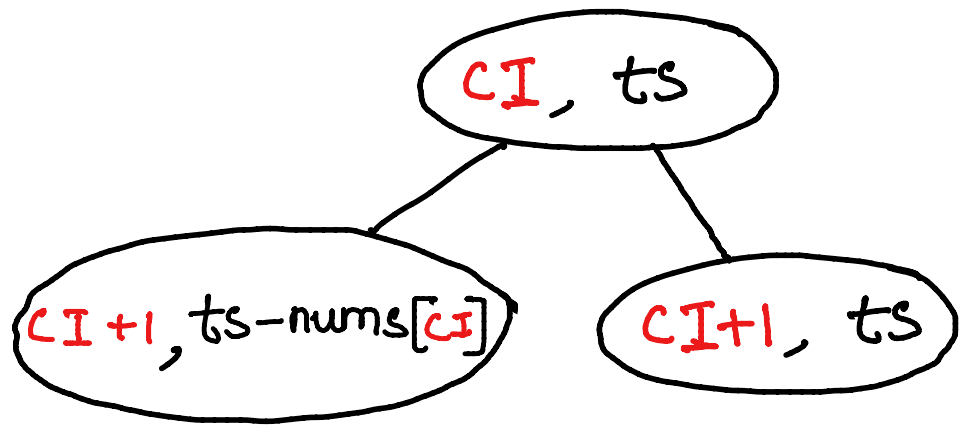
2) if we donot select then ts = ts (i.e remains same)
CI = CI + 1

3) return OR of left & right branch.

⇒ nums = [1, 5, 11, 5] (0, 1, 2, 3)

sum = 22

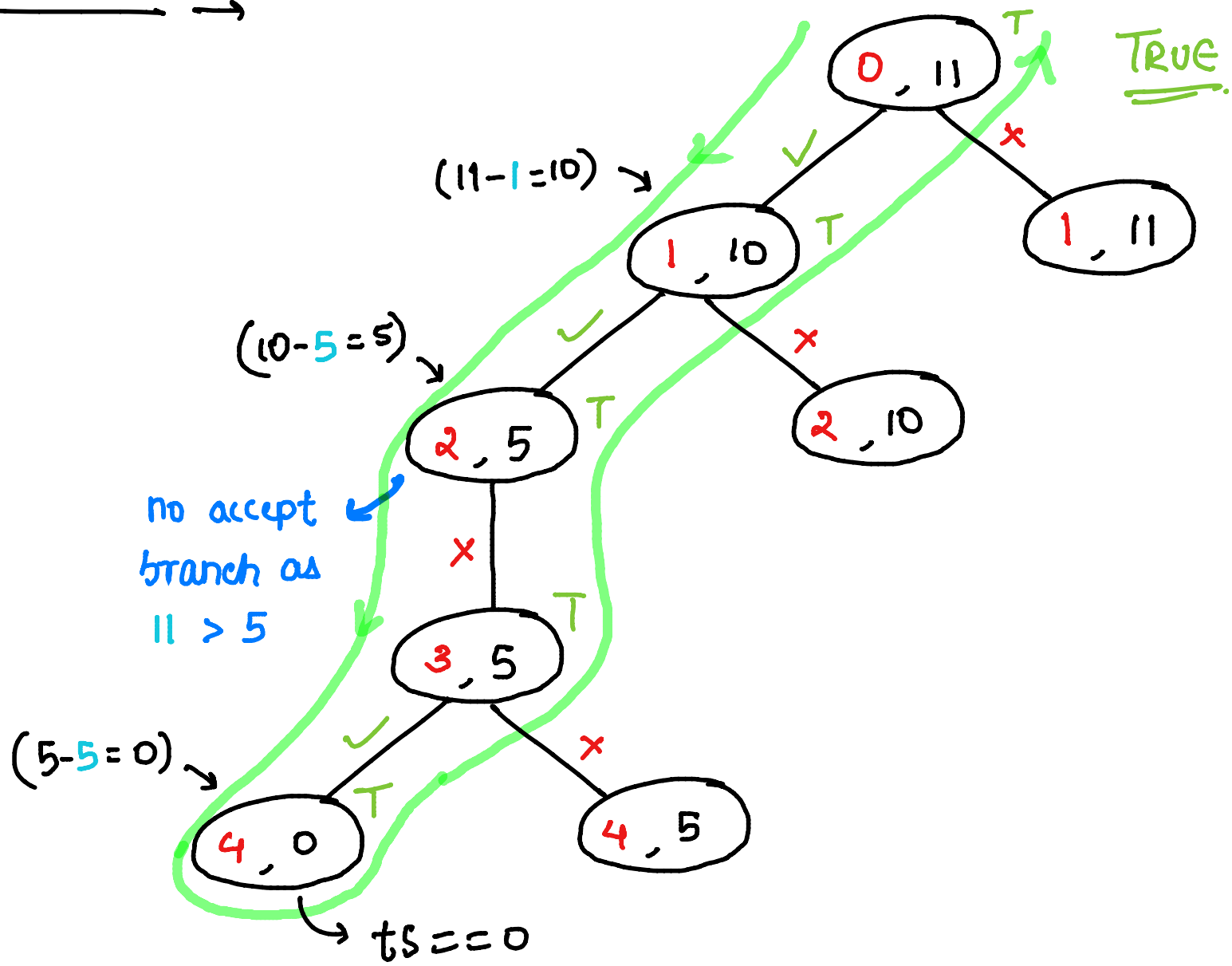
ts = 11



Here sum % 2 == 0

∴ dividing into 2 subsets is possible.

Explanation →



⇒ that subset is found

& return True

as we are using OR, one True branch is sufficient

Code →

```cpp
class Solution {
public:

    bool isPossible(int targetSum,int currentIndex, vector<int>&nums,
                                    unordered_map<string, bool> &memo){

        if(targetSum == 0)
            return true;

        if(currentIndex >= nums.size())
            return false;

        string currentKey = to_string(currentIndex)+"_"+to_string(targetSum);

        if(memo.find(currentKey)!=memo.end()){
            return memo[currentKey];
        }

        bool possible = false;

        if(nums[currentIndex]<=targetSum)
            possible = isPossible(targetSum-nums[currentIndex], currentIndex+1, nums, memo);

        // if already Possible then return True directly
        if(possible){
            memo[currentKey] = possible;
            return true;
        }

        bool notPossible = isPossible(targetSum, currentIndex+1, nums, memo);

        memo[currentKey] = possible||notPossible;
        return memo[currentKey];
    }

    bool canPartition(vector<int>& nums) {

        int total = 0;
        for(auto it:nums) total+= it;

        if(total%2!=0)  return false;

        unordered_map<string, bool> memo;
        return isPossible(total/2,0, nums,memo);
    }
};
```

(9) Target Sum →

given an array & target, find the number of ways to reach target by using + or - before each element in array.

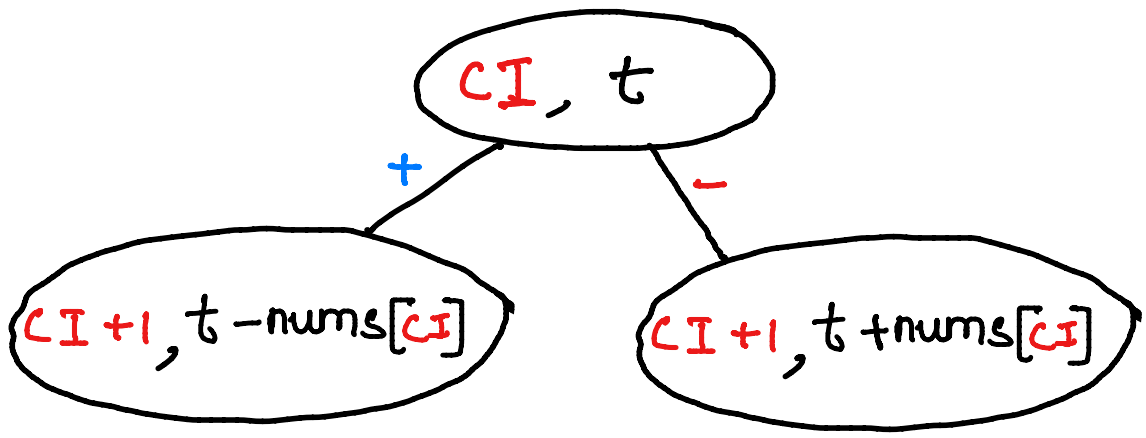
Eg

Input: nums = [1,1,1,1,1], target = 3
Output: 5
Explanation: There are 5 ways to assign symbols to make the sum of nums be target 3.
-1 + 1 + 1 + 1 + 1 = 3
+1 - 1 + 1 + 1 + 1 = 3
+1 + 1 - 1 + 1 + 1 = 3
+1 + 1 + 1 - 1 + 1 = 3
+1 + 1 + 1 + 1 - 1 = 3

→ at every index we can use + or - sign

if + then $t = t - (+nums[CI]) \Rightarrow t - nums[CI]$

if - then $t = t - (-nums[CI]) \Rightarrow t + nums[CI]$



→ at every node, return the sum of values from left & right. Because we need to find the total number of ways.

code →

```cpp
class Solution {
public:
    int totalWays(int currentIndex, vector<int>&nums, int target, unordered_map<string,int> &memo){

        if(target==0 and currentIndex==nums.size()){
            return 1;
        }

        if(currentIndex>=nums.size() and target!=0){
            return 0;
        }

        string key = to_string(currentIndex)+"_"+to_string(target);

        if(memo.find(key)!=memo.end()){
            return memo[key];
        }

        int plus = totalWays(currentIndex+1, nums, target-nums[currentIndex],memo);

        int minus = totalWays(currentIndex+1, nums, target+nums[currentIndex],memo);

        memo[key] = plus+minus;

        return plus+minus;
    }

    int findTargetSumWays(vector<int>& nums, int target) {
        unordered_map<string,int> memo;
        return totalWays(0,nums,target,memo);
    }
};
```

(10) Count number of subsets with given difference →

→ This is similar to Target Sum.

given the difference between two subsets, and an array find no. of subsets with the difference.

Approach →

Let say $S1 - S2 =$ difference (given) — (1)

we can calculate sum of every element, say sum

& it can be said that for 2 subsets S1 & S2

$S1 + S2 =$ Sum. — (2)

Now (1) + (2) $\Rightarrow$ $2(S1) =$ difference + sum

$S1 =$ (difference + sum)/2.

→ Implement Target sum with target value = S1.

(11) Delete and Earn →

You are given an integer array `nums`. You want to maximize the number of points you get by performing the following operation any number of times:

- Pick any `nums[i]` and delete it to earn `nums[i]` points. Afterwards, you must delete **every** element equal to `nums[i] - 1` and **every** element equal to `nums[i] + 1`.

Return *the **maximum number of points** you can earn by applying the above operation some number of times.*

Eg nums = [2, 2, 3, 3, 3, 4]

→ if we start deleting 2, then result = 2+2 = 4

then nums = [3, 3, 3, 4]

& we need to delete all 2+1 & 2-1 ⇒ nums = [4]

→ if we delete 4, then result = 4+4 = 8.

& nums = [ ]

(or)

→ if we start deleting 3, then result = 3+3+3 = 9.

then nums = [2, 2, 4]

& we need to delete all 3-1 & 3+1 ⇒ nums = [ ]

∴ result = 9.

(or)

→ if we start deleting 4, then result = 4

then nums = [2, 2, 3, 3, 3]

& we need to delete all 4+1 & 4-1 ⇒ nums = [2, 2]

→ if we delete 2, then result = 4+4 = 8.

& nums = [ ]

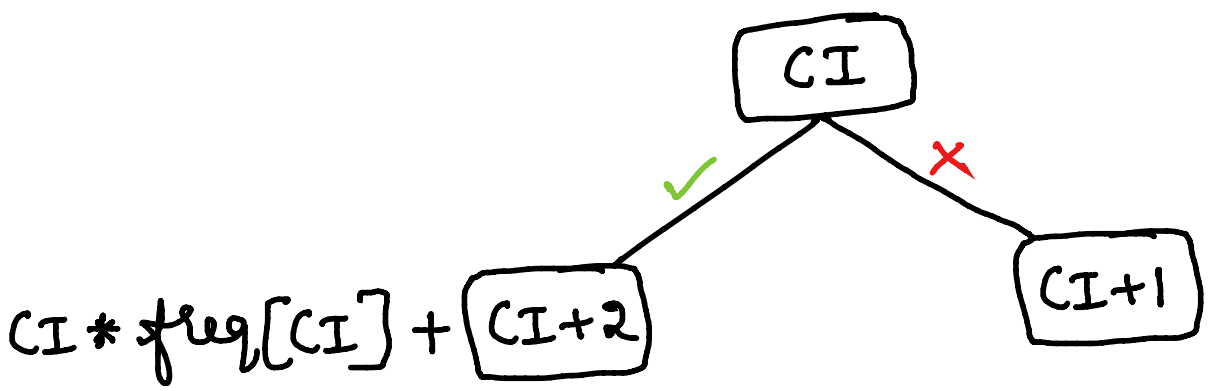
→ we can store frequency of each element and use the similar approach

nums = [2, 2, 3, 3, 3, 4]

freq =

0	1	2	3	4
0	0	2	3	1

CI



Code →

```cpp
class Solution {
public:

    int maxPoints(vector<int>& freq, int currentIndex, unordered_map<int,int>&memo){

        if(currentIndex >= freq.size()) return 0;

        int key = currentIndex;

        if(memo.find(key) != memo.end()) return memo[key];

        int Delete = currentIndex*freq[currentIndex] + maxPoints(freq, currentIndex+2, memo);
        int NotDelete = maxPoints(freq, currentIndex+1, memo);

        memo[key] = max(Delete, NotDelete);

        return memo[key];
    }

    int deleteAndEarn(vector<int>& nums) {

        int maxi = *max_element(nums.begin(), nums.end());
        vector<int> freq(maxi+1, 0);

        for(auto i: nums)   freq[i]++;

        unordered_map<int,int> memo;

        return maxPoints(freq, 0, memo);
    }
};
```

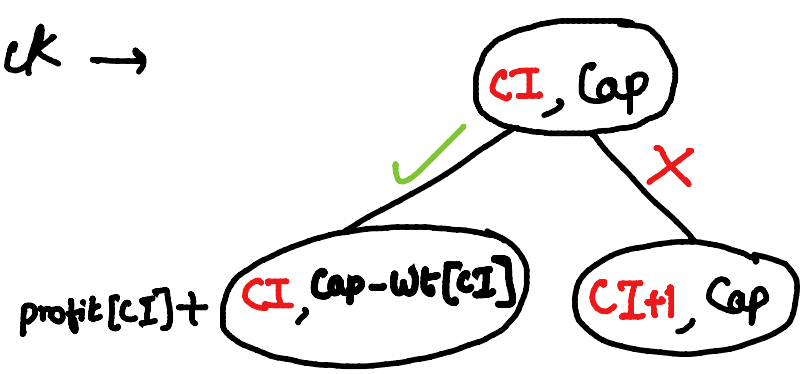
(12) Unbounded Knapsack → Similar to 0-1 knapsack but allows us to choose an item more than once

Eg
wt = [2, 1] (0, 1)
values = [1, 1]
capacity = 3

if bounded knapsack then profit = 2 (2, 1) (1+1)

if unbounded knapsack then profit = 3 (1, 1, 1) (1+1+1)

for unbounded knapsack →



Code →

```cpp
class Solution{
public:
    int helper(int W, int wt[], int val[], int N, int curr, vector<vector<int>>&memo){

        if(W==0)    return 0;
        if(curr==N) return 0;

        if(memo[curr][W]!=-1)   return memo[curr][W];

        int currWt = wt[curr];
        int currVal = val[curr];

        int selected = 0;
        if(currWt<=W){
            selected = currVal + helper(W-currWt, wt, val, N, curr, memo);
        }

        int notSelected = helper(W, wt, val, N, curr+1, memo);

        memo[curr][W] = max(selected, notSelected);
        return memo[curr][W];
    }

    int knapSack(int N, int W, int val[], int wt[])
    {
        vector<vector<int>> memo( N , vector<int> (W+1, -1));
        return helper(W, wt, val, N, 0, memo);
    }
};
```

(13) Coin Change II → (Similar to unbounded knapsack)

given an array of coins & amount, find total number of ways / combinations to make up that amount.

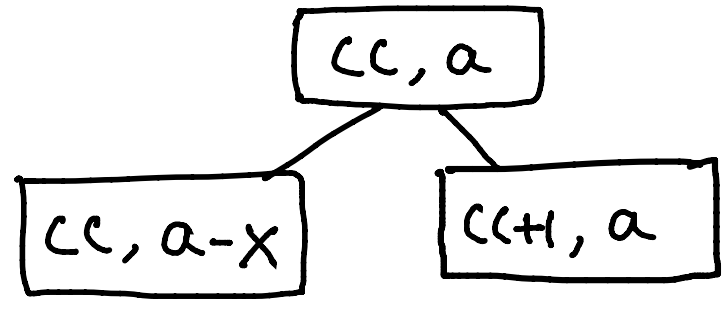
Eg coins = [1, 2, 5]

amount = 5

4 ways
- 1+1+1+1+1
- 1+1+1+2
- 1+2+2
- 5

coins = [_ x _ _ _] x = coins[cc]

current coin: cc, a → amount



code →

```cpp
class Solution {
public:

    int totalWays(int currentIndex, vector<int>& coins, int amount,vector<vector<int>>&memo){
        if(amount == 0) return 1;   // amount==0 means that target is reached so return 1
        if(currentIndex >= coins.size()) return 0;  //if index is out of bounds then return 0

        if(memo[currentIndex][amount]!=-1) return memo[currentIndex][amount];

        int consider = 0;
        if(coins[currentIndex]<=amount){
            consider = totalWays(currentIndex,coins,amount-coins[currentIndex],memo);
        }
        int notConsider = totalWays(currentIndex+1,coins,amount,memo);

        memo[currentIndex][amount] = consider+notConsider;
        return memo[currentIndex][amount];
    }

    int change(int amount,vector<int>& coins) {
        vector<vector<int>>memo(coins.size()+1,vector<int>(amount+1,-1));
        return totalWays(0,coins,amount,memo);
    }
};
```

(14) Coin Change → (Similar to unbounded knapsack)

Given an array of coins & amount, find fewest number of coins to make up that amount, return -1 if its not possible

Eg coins = [1, 2, 5]

amount = 11

for 11 ⇒ 1+....+1 (11 times) = 11

1+2+2+2+2+2 = 11

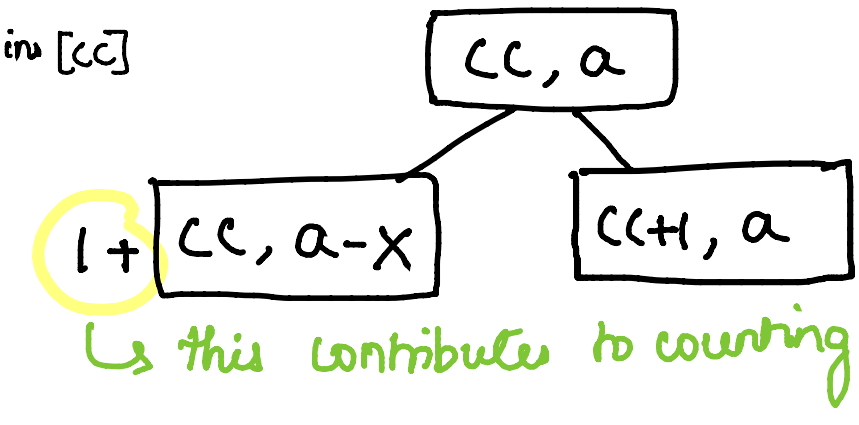
1+5+5 = 11

out of all the ways last has min coins.

coins = [_ x _ _ _]

x = coins[cc]

cc, a → current coin, amount



↳ this contributes to counting coins.

code →

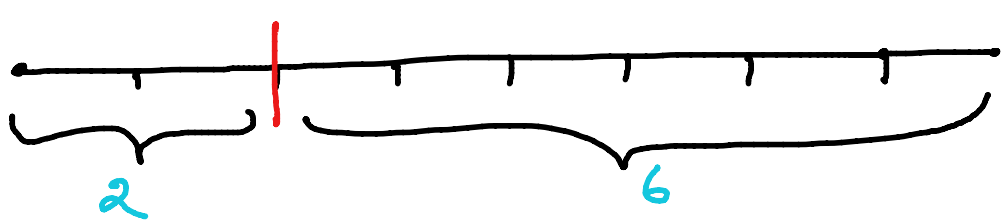
```cpp
class Solution {
public:

    int minimumCoins(int currentIndex, vector<int>& coins, int amount,vector<vector<int>>&memo){
        if(amount == 0)    return 0;
        if(currentIndex >= coins.size())    return 100000;  //Any Max Value outside boundary

        if(memo[currentIndex][amount]!=-1)  return memo[currentIndex][amount];

        int consider = 100000;
        if(coins[currentIndex]<=amount){
            consider = 1 + minimumCoins(currentIndex, coins, amount-coins[currentIndex], memo);
        }

        int notConsider = minimumCoins(currentIndex+1, coins, amount, memo);

        memo[currentIndex][amount]= min(consider,notConsider);
        return memo[currentIndex][amount];
    }

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

        vector<vector<int>>memo(coins.size()+1,vector<int>(amount+1,-1));
        int ans = minimumCoins(0, coins, amount, memo);

        return (ans==100000)? -1 : ans;
    }
};
```

(15) Rod Cutting →

Given a rod of length N and array of prices. find the max value that can be obtained by cutting rod.

Eg N = 8 prices = [1, 5, 8, 9, 10, 17, 17, 20]
(1 2 3 4 5 6 7 8 / 0 1 2 3 4 5 6 7)

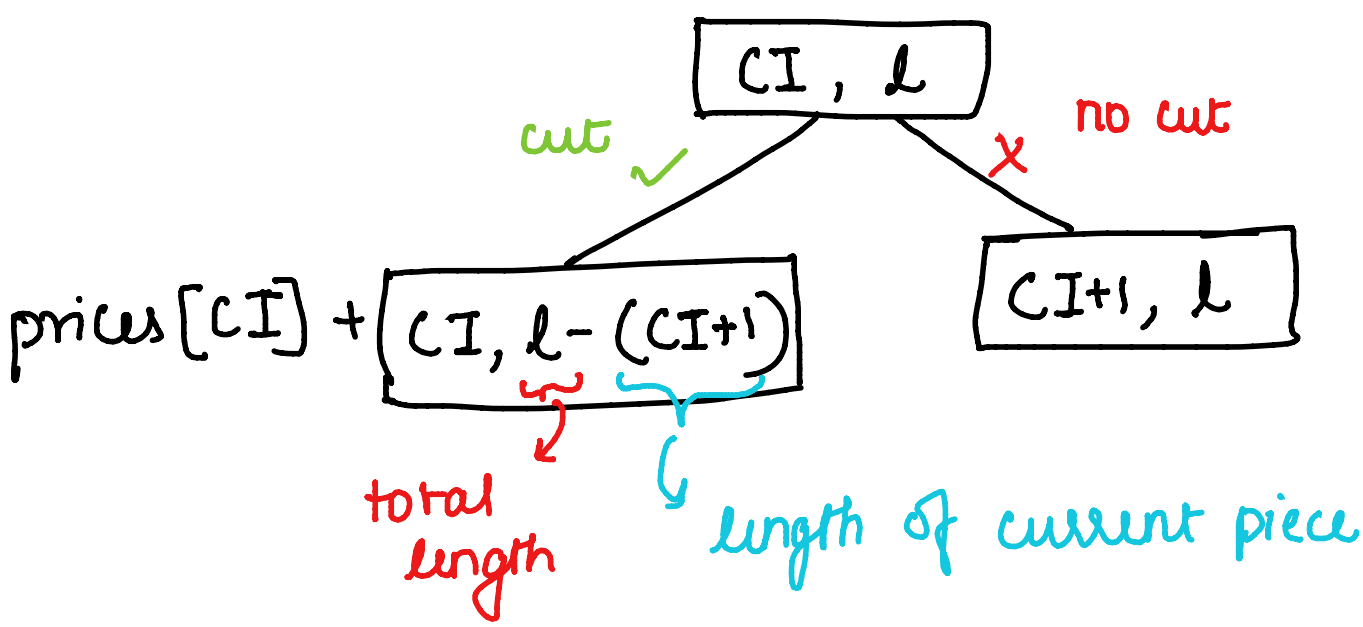
* price of a piece is prices[CI], whose length is CI+1



if we cut our rod into 2 pieces of length 2, 6 we get max value of 5+17 is 22.

→ there might be other ways, but this particular configuration returns max value.

* At any instance length of current piece is CI+1



code →

```cpp

class Solution{
  public:
    int maxProfit(int price[],int currentIndex, int n, vector<vector<int>>&memo){
        if(n==0)   return 0;
        if(currentIndex>=n)   return 0;

        if(memo[currentIndex][n]!=-1)   return memo[currentIndex][n];

        int selected = 0;
        if(currentIndex+1<=n){
            selected = price[currentIndex]+maxProfit(price, currentIndex, n-(currentIndex+1), memo);
        }
        int notSelected = maxProfit(price, currentIndex+1, n, memo);

        memo[currentIndex][n] = max(selected, notSelected);
        return memo[currentIndex][n];
    }
    int cutRod(int price[], int n) {
        vector<vector<int>> memo(n+1, vector<int>(n+1,-1));
        return maxProfit(price,0,n,memo);
    }
};
```

Dynamic Programming - 2

- Karun Karthik

Contents

(16) Best time to Buy & Sell Stock →

Given an array of prices, find the max profit if we are allowed to do one transaction

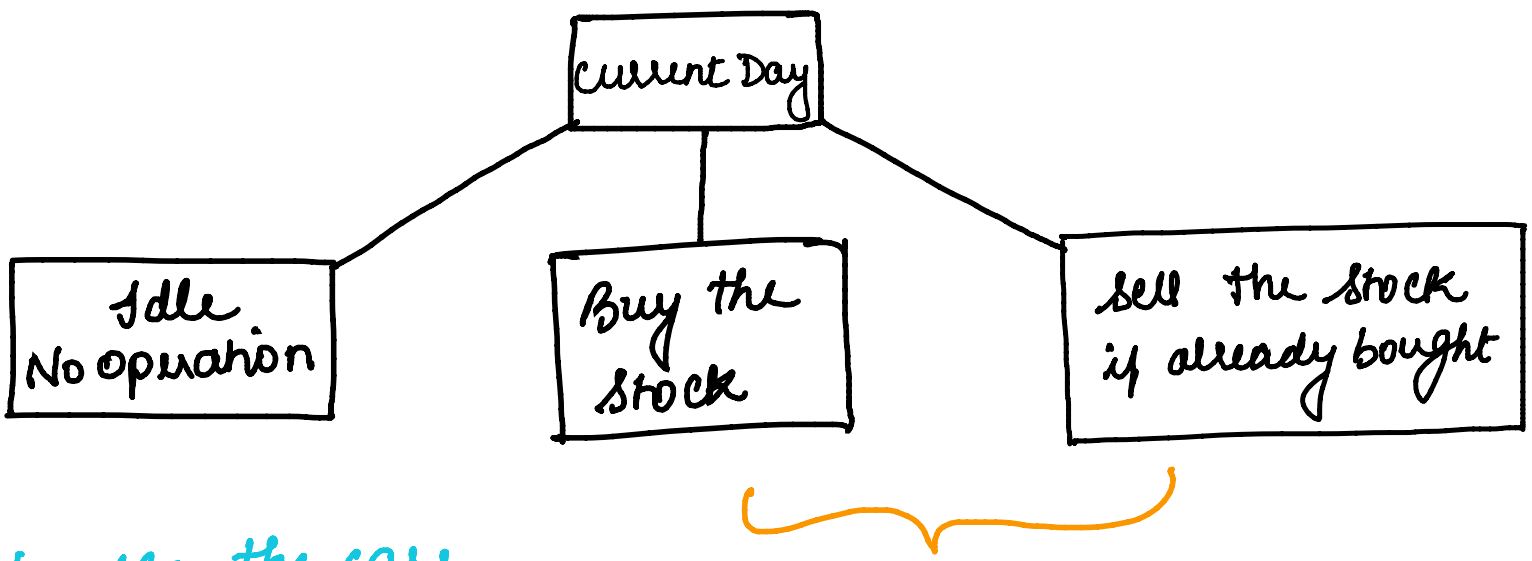
Eg

prices = [7, 1, 5, 3, 6, 4] → we get maxProfit when we buy at day 0 & sell on day 4

0 1 2 3 4 5

⇒ profit = 6 − 1 = 5.

Lets look at choices we have,



→ to handle the case that transaction could occur once, we use a variable called transaction = 1.

→ to handle these cases, we use a variable called canBuy.

→ once bought canBuy = false

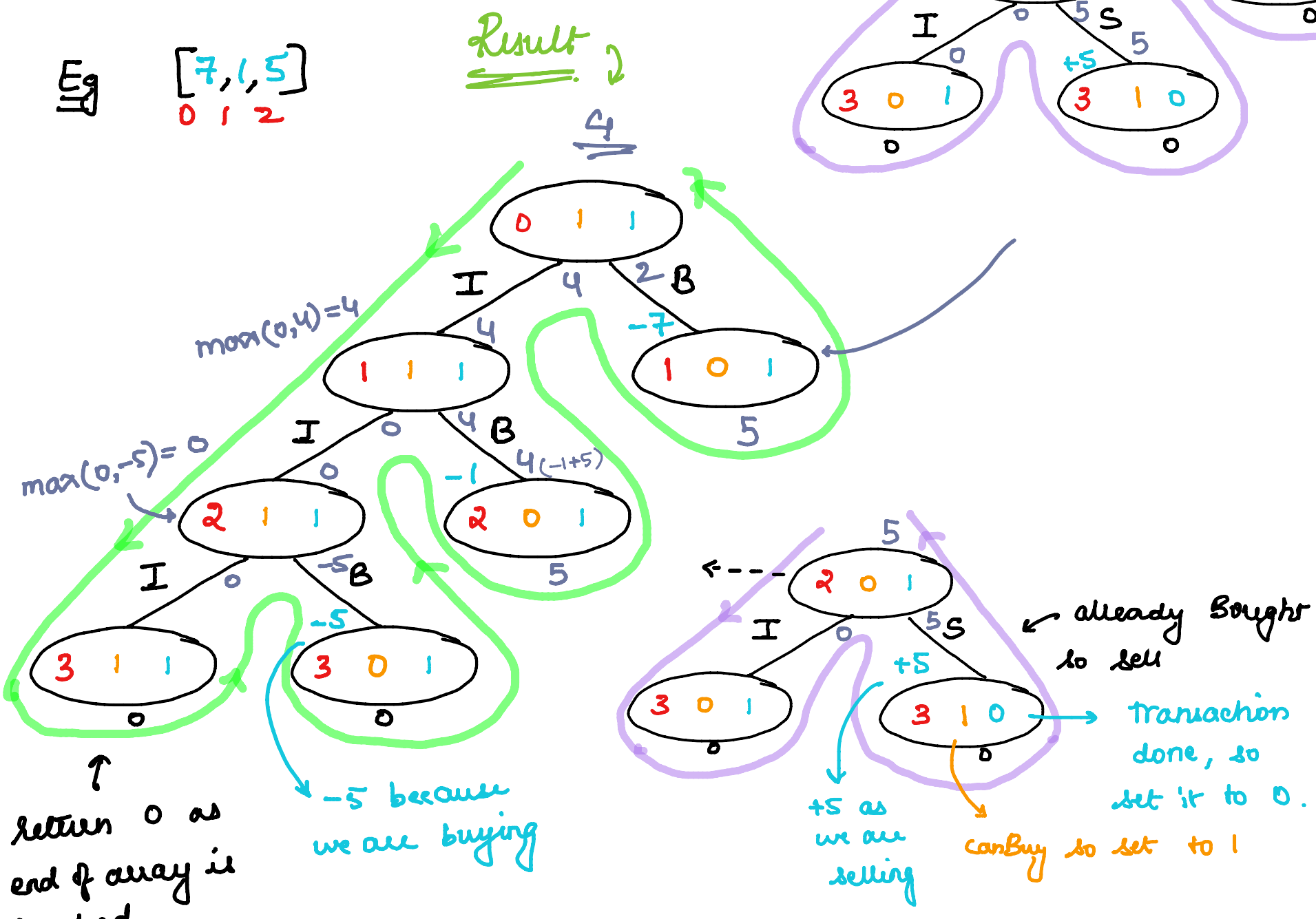
→ once sold canBuy = true

∴ Our recursive structure would be as follows →

current Day, canBuy, transaction

Eg [7,1,5]
0 1 2

Result



code →

```cpp
class Solution {
public:
    int find(vector<int> &prices, int currDay, int k, bool canBuy, vector<vector<int>> &memo){

        if(currDay >= prices.size() || k<=0 ) return 0;

        if(memo[currDay][canBuy] != -1) return memo[currDay][canBuy];

        if(canBuy)
        {
            int idle = find(prices, currDay+1, k, canBuy, memo);
            int buy = -prices[currDay] + find(prices, currDay+1, k, !canBuy, memo);
            return memo[currDay][canBuy] = max(buy, idle);
        }
        else
        {
           int idle = find(prices, currDay+1, k, canBuy, memo);
           int sell = prices[currDay] + find(prices, currDay+1, k-1, !canBuy, memo);
           return memo[currDay][canBuy] = max(sell, idle);
        }
    }
    int maxProfit(vector<int>& prices) {
        int n = prices.size();
        vector<vector<int>> memo(n,vector<int> (2,-1));
        // canBuy = true and transaction as k = 1
        return find(prices,0,1,true,memo);
    }
};
```

(17) Best time to Buy & Sell Stock - II →

→ In this we can have many transactions that can be done.

Eg → prices = [7, 1, 5, 3, 6, 4]
(indices: 0 1 2 3 4 5)

↳ Buy on 1 & sell on 2 — profit = 5 - 1 = 4

Buy on 3 & sell on 4 — profit = 6 - 3 = 3

Total Profit = 7 Ans

code →

Remove the parameter K i.e transaction limit.

```cpp
class Solution {
public:
    int find(vector<int> &prices, int currDay, bool canBuy, vector<vector<int>> &memo){

        if(currDay >= prices.size()) return 0;

        if(memo[currDay][canBuy] != -1) return memo[currDay][canBuy];

        if(canBuy)
        {
            int idle = find(prices, currDay+1, canBuy, memo);
            int buy = -prices[currDay] + find(prices, currDay+1, !canBuy, memo);
            return memo[currDay][canBuy] = max(buy, idle);
        }
        else
        {
           int idle = find(prices, currDay+1, canBuy, memo);
           int sell = prices[currDay] + find(prices, currDay+1, !canBuy, memo);
           return memo[currDay][canBuy] = max(sell, idle);
        }
    }
    int maxProfit(vector<int>& prices) {
        int n = prices.size();
        vector<vector<int>> memo(n,vector<int> (2,-1));
        // canBuy = true and transaction are infinite so ignore k
        return find(prices, 0, true, memo);
    }
};
```

(18) Best time to Buy & Sell Stock - III →

In this maximum profit has to be achieved by making atmost 2 transactions.

Eg prices = [3, 3, 5, 0, 0, 3, 1, 4]
(indices: 0 1 2 3 4 5 6 7)

↳ Buy on 4 & sell on 5 — profit = 3 - 0 = 3

Buy on 6 & sell on 7 — profit = 4 - 1 = 3

Total Profit = 6 Ans.

Code →

In the base condition is no. of transactions >= 2 then return 0.

↳ ie possible transactions are when it is = 0, 1

(Line 6)

```cpp
class Solution {
public:
    int find(vector<int> &prices, int currDay, int transaction, bool canBuy,
    vector<vector<vector<int>>> &memo){

        if(currDay >= prices.size() || transaction>=2) return 0;

        if(memo[currDay][canBuy][transaction] != -1) return memo[currDay][canBuy][transaction];

        if(canBuy)
        {
            int idle = find(prices, currDay+1, transaction, canBuy, memo);
            int buy = -prices[currDay] + find(prices, currDay+1, transaction, !canBuy, memo);
            return memo[currDay][canBuy][transaction] = max(buy, idle);
        }
        else
        {
           int idle = find(prices, currDay+1, transaction, canBuy, memo);
           int sell = prices[currDay] + find(prices, currDay+1, transaction+1, !canBuy, memo);
           return memo[currDay][canBuy][transaction] = max(sell, idle);
        }
    }
    int maxProfit(vector<int>& prices) {
        int n = prices.size();
        vector<vector<vector<int>>> memo(n, vector<vector<int>>(2,vector<int>(2,-1)));
        // canBuy = true and transactions are allowed 2 times
        return find(prices, 0, 0, true, memo);
    }
};
```

(19) Best time to Buy & Sell Stock - IV →

This is a generalised version of previous problem, instead of limiting it to 2 transactions, we need to allow atmost k transactions.

code →

Pass k as an argument & use it to limit transaction in base condition. (Line 6)

```
1  class Solution {
2  public:
3      int find(vector<int> &prices, int currDay, int transaction, int k, bool canBuy,
4      vector<vector<vector<int>>> &memo){
5
6          if(currDay >= prices.size() || transaction>=k) return 0;
7
8          if(memo[currDay][canBuy][transaction] != -1) return memo[currDay][canBuy][transaction];
9
10         if(canBuy)
11         {
12             int idle = find(prices, currDay+1, transaction, k, canBuy, memo);
13             int buy = -prices[currDay] + find(prices, currDay+1, transaction, k, !canBuy, memo);
14             return memo[currDay][canBuy][transaction] = max(buy, idle);
15         }
16         else
17         {
18            int idle = find(prices, currDay+1, transaction, k, canBuy, memo);
19            int sell = prices[currDay] + find(prices, currDay+1, transaction+1, k, !canBuy, memo);
20            return memo[currDay][canBuy][transaction] = max(sell, idle);
21         }
22     }
23     int maxProfit(int k, vector<int>& prices) {
24         int n = prices.size();
25         vector<vector<vector<int>>> memo(n ,vector<vector<int>>(2,vector<int>(k+1,-1)));
26         // canBuy = true and transactions are allowed atmost k times
27         return find(prices, 0, 0, k, true, memo);
28     }
29 };
```

(20) Best time to Buy & Sell Stock with Cooldown →

In this, cooldown means that we cannot buy a stock on the immediate day after it is sold.

=> the day after sold should be skipped.

code →

To skip day after sell, increment the currentDay by 2. (Line 18)

```cpp
class Solution {
public:
    int find(vector<int> &prices, int currDay, bool canBuy, vector<vector<int>> &memo){

        if(currDay >= prices.size()) return 0;

        if(memo[currDay][canBuy] != -1) return memo[currDay][canBuy];

        if(canBuy)
        {
            int idle = find(prices, currDay+1, canBuy, memo);
            int buy = -prices[currDay] + find(prices, currDay+1, !canBuy, memo);
            return memo[currDay][canBuy] = max(buy, idle);
        }
        else
        {
           int idle = find(prices, currDay+1, canBuy, memo);
           int sell = prices[currDay] + find(prices, currDay+2, !canBuy, memo);
           return memo[currDay][canBuy] = max(sell, idle);
        }
    }
    int maxProfit(vector<int>& prices) {
        int n = prices.size();
        vector<vector<int>> memo(n,vector<int> (2,-1));
        // canBuy = true & transaction = infinite so ignore k & while sell, currDay +=2
        return find(prices, 0, true, memo);
    }
};
```

(21) Best time to Buy & Sell Stock with Transaction Fee →

In this variation, we donot have limit on transaction but while making a transaction i.e selling it, some fee has to be paid i.e transaction fee.

code →

Deduct the fee from the selling day's amount.

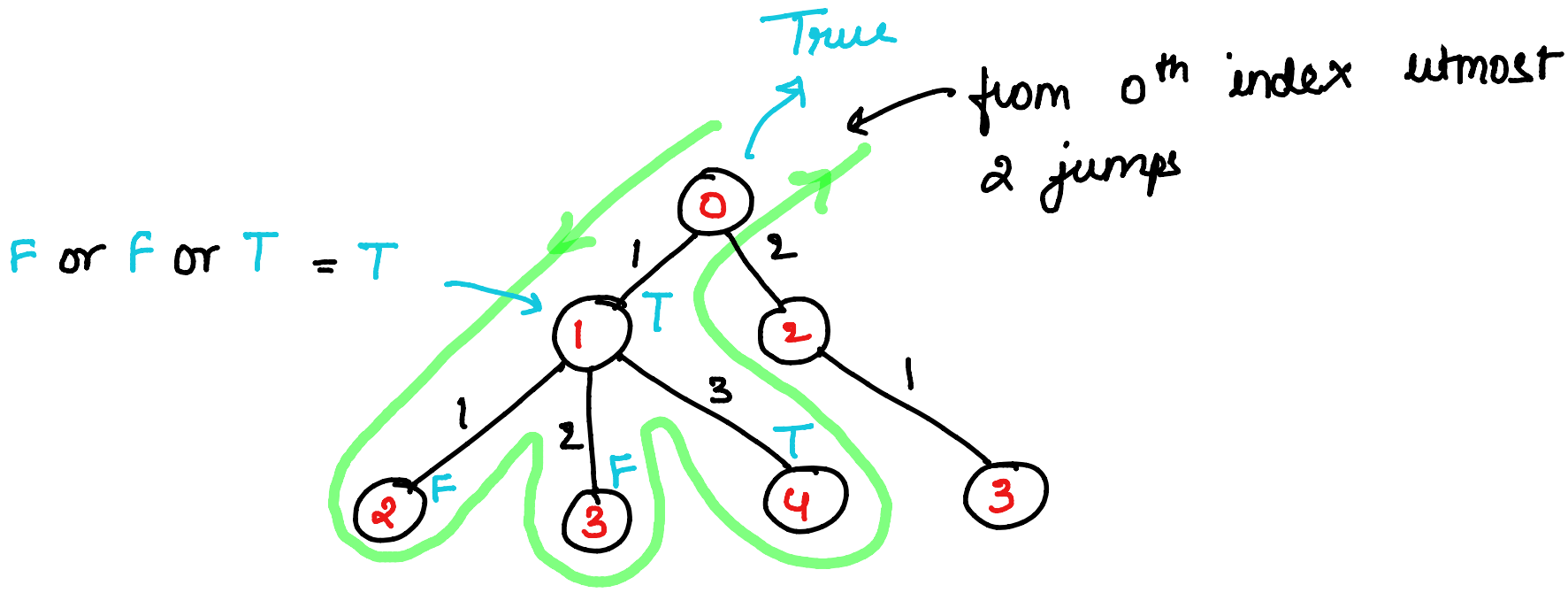
(Line 18)

```cpp
class Solution {
public:
    int find(vector<int> &prices, int currDay, int fee, bool canBuy, vector<vector<int>> &memo){

        if(currDay >= prices.size()) return 0;

        if(memo[currDay][canBuy] != -1) return memo[currDay][canBuy];

        if(canBuy)
        {
            int idle = find(prices, currDay+1, fee, canBuy, memo);
            int buy = -prices[currDay] + find(prices, currDay+1, fee, !canBuy, memo);
            return memo[currDay][canBuy] = max(buy, idle);
        }
        else
        {
           int idle = find(prices, currDay+1, fee, canBuy, memo);
           int sell = (prices[currDay]-fee) + find(prices, currDay+1, fee, !canBuy, memo);
           return memo[currDay][canBuy] = max(sell, idle);
        }
    }

    int maxProfit(vector<int>& prices, int fee) {
        int n = prices.size();
        vector<vector<int>> memo(n,vector<int> (2,-1));
        // canBuy = true & transaction = infinite so ignore k & while selling deduce fee
        return find(prices, 0, fee, true, memo);
    }
};
```

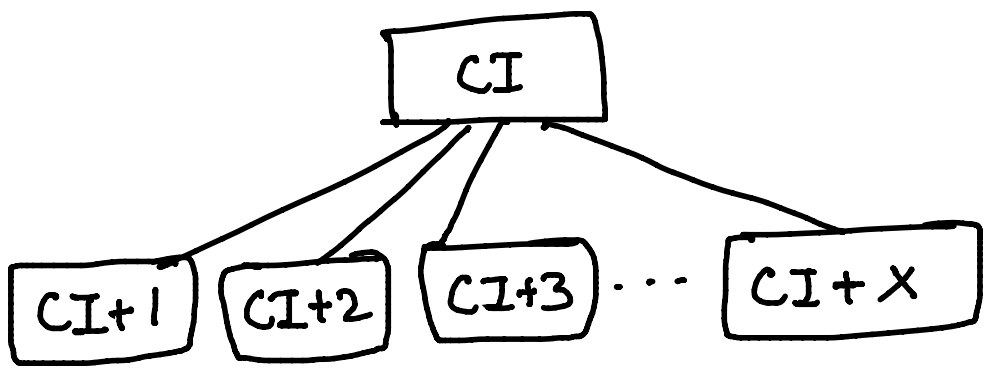
(22) Jump Game →

Given array of nums which indicate max number of jump from any index. Return true if you can reach last index.

Eg nums = [2, 3, 1, 1, 4]
0 1 2 3 4



Therefore, [_ _ X _ _ _]
CI



} can be implemented using a for loop.

Note: submitting DP solution gives TLE. This is just for understanding Optimal solution involves Greedy approach.

$$Tc \rightarrow O(\max(nums[i]) * n)$$

max time for for loop.

code →

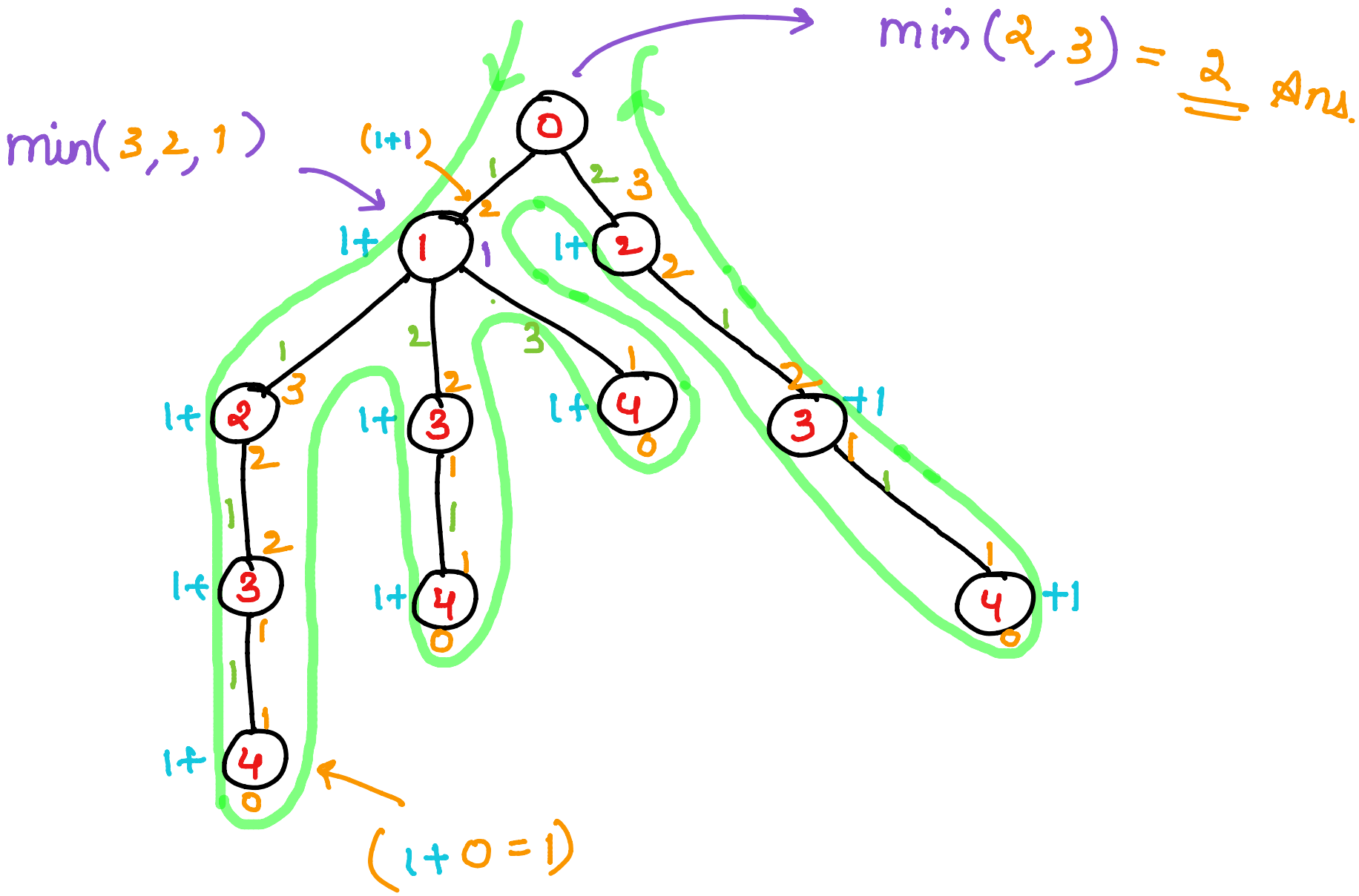
```cpp
class Solution {
public:
    bool isPossible(vector<int>&nums, int curr, unordered_map<int,bool>&memo)
    {
        if(curr >= nums.size()-1) return true;

        int currKey = curr;

        if(memo.find(currKey)!=memo.end()) return memo[currKey];

        int currJump = nums[curr];

        if(currJump >= nums.size() - curr)  return true;

        bool ans = false;

        for(int i=1; i<=currJump; i++){
            bool tempAns = isPossible(nums,curr+i,memo);
            ans = ans || tempAns;
        }
        return memo[currKey] = ans;
    }

    bool canJump(vector<int>& nums){
        unordered_map<int,bool>memo;
        return isPossible(nums, 0, memo);
    }
};
```

(23) Jump Game II →

Given array of nums which indicate max number of jump from any index. Reach last index in minimum number of moves.

Eg nums = [2, 3, 1, 1, 4]

0 1 2 3 4



→ If currentIndex >= lastIndex
then return 0.

while returning add 1 for counting ways!

Code →

```cpp
class Solution {
public:

    int minJumps(vector<int>& nums,int curr,vector<int>&memo)
    {
        if( curr >= nums.size()-1) return 0;

        int currKey = curr;
        if(memo[currKey]!=-1) return memo[currKey];

        int currJump = nums[curr];

        // some large value
        int ans = 10001;

        for(int i=1;i<=currJump;i++){
            int tempans = 1 + minJumps(nums,curr+i,memo);
            ans = min(ans, tempans);
        }
        return memo[currKey] = ans;
    }

    int jump(vector<int>& nums) {
        vector<int> memo(nums.size()+1,-1);
        return minJumps(nums, 0, memo);
    }
};
```

(24) Reach a given Score →

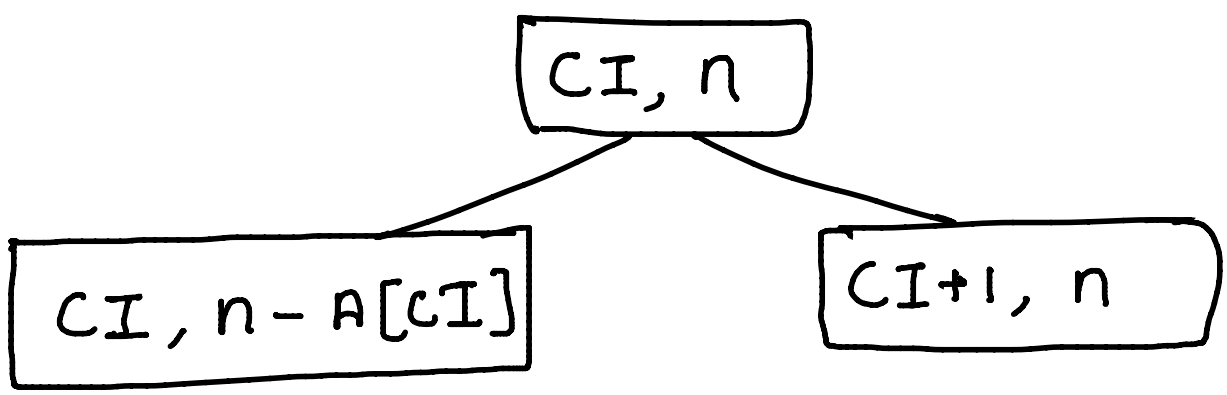
given 3 scores [3,5,10] & 'n'.
Return total number of ways to create n using the scores.

Eg n=8 then no. of ways to create 8 from [3, 5, 10]
is 1. (3+5)

n=13 then no. of ways to create 13 from [3, 5, 10]
is 2 (3+5+5) & (3+10)

n=20 then no. of ways to create 20 from [3, 5, 10]
is 4 (3+3+3+3+3+5) & (5+5+5+5)
& (5+5+10) & (10+10)

∴ let say A = [3,5,10] then
CI



code →

```cpp
typedef long long LL;

LL ways(int curr, LL n, vector<int>&score, vector<vector<int>>&vec)
{
    if(n==0) return 1;

    if(curr>=score.size()) return 0;

    if(vec[curr][n]!=-1) return vec[curr][n];

    LL consider = 0;

    if(score[curr]<=n)
      consider = ways(curr,n-score[curr],score,vec);

    LL notconsider = ways(curr+1,n,score,vec);

    return vec[curr][n] = consider + notconsider;
}

LL count(LL n)
{
    vector<int>score{3,5,10};
    vector<vector<int>>vec(score.size(),vector<int>(1001,-1));
    return ways(0,n,score,vec);
}
```

(25) Applications of Catalan Number →

Catalan Numbers are defined using the formula

$$C_n = \frac{(2n)!}{(n+1)!\,n!} = \prod_{k=2}^{n} \frac{n+k}{k} \quad \text{for } n \geq 0$$

this can be used recursively as follows,

$$C_{n+1} = \sum_{i=0}^{n} C_i C_{i-1} \quad \Big\} \; n \geq 0 \;\&\; C_0 = 1$$

→ $C_0 = \underline{1}$.

→ $C_1 = \underline{1}$.

→ $C_2 = C_0 \cdot C_1 + C_1 \cdot C_0 = 1 \cdot 1 + 1 \cdot 1 = \underline{2}$.

→ $C_3 = C_0 C_2 + C_1 C_1 + C_2 C_0 = 1 \cdot 2 + 1 \cdot 1 + 2 \cdot 1 = \underline{5}$.

→ $C_4 = C_0 C_3 + C_1 C_2 + C_2 C_1 + C_3 C_0$

$= 1 \cdot 5 + 1 \cdot 2 + 2 \cdot 1 + 5 \cdot 1 = \underline{14}$.

Appns →

1. No. of possible BST with n keys.
2. No. of valid combinations for N pair of parenthesis.

(26) N^{th} Catalan Number →

To find N^{th} catalan number we can use formula

$$C_{n+1} = \sum_{i=0}^{n} C_i C_{i-1} \quad \Big\} \; n \geq 0 \;\; \& \;\; C_0 = 1$$

↳ this can be implemented by

i) having base condition for $n==0$ & $n==1$

ii) using a loop to sum values from $i=0$ to n

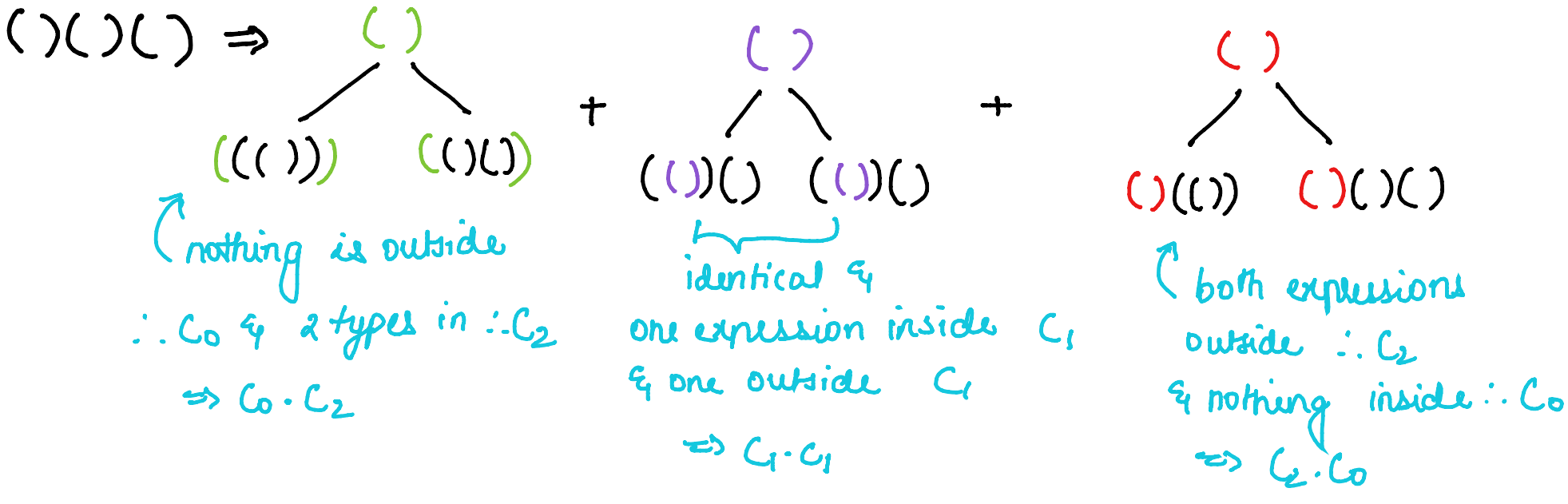
Code →

```cpp
class Solution
{
    public:
    cpp_int ncatalan(int n, vector<cpp_int>& memo) {
        if(n == 0 || n == 1) return 1;

        int curr = n;
        if(memo[curr]!=-1) return memo[curr];

        cpp_int catalan = 0;

        for(int i=0;i<n;i++) {
            catalan += ncatalan(i, memo)*ncatalan(n-i-1, memo);
        }

        memo[curr] = catalan;
        return memo[curr];
    }

    cpp_int findCatalan(int n)
    {
        vector<cpp_int> memo(1001,-1);
        return ncatalan(n, memo);
    }
};
```

(27) Number of valid Parenthesis Expression →

Given N, find total number of ways in which we can arrange N pair of parenthesis in a Balanced way.

Eg N=6 ⇒ ( )( )( ), ( )(( )), (( ))( ), ((( ))) ∴ Res = 4



⇒ $C_0 \cdot C_2 + C_1 \cdot C_1 + C_2 \cdot C_0 = C_3$ ⇒ for n we need to find ncatalan(n/2)

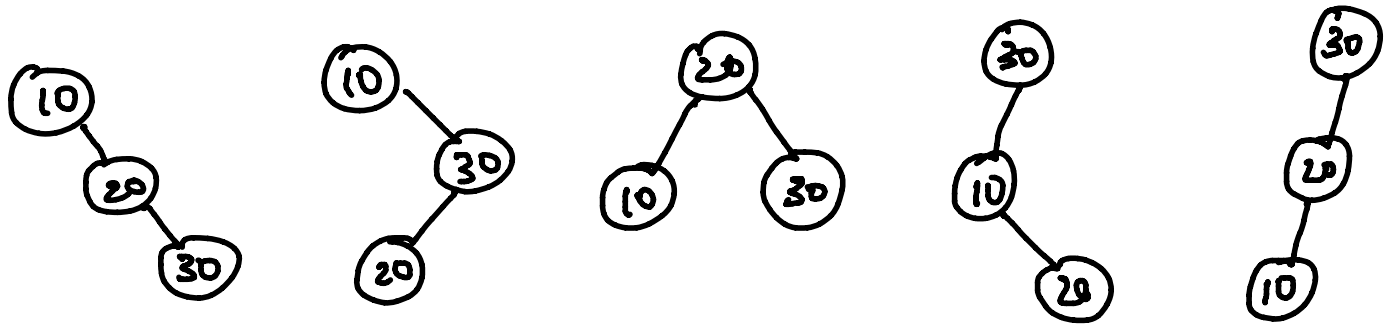
Code →

```cpp
#include<bits/stdc++.h>
using namespace std;

int ncatalan(int n, unordered_map<int,int>& memo) {
    if(n == 0 || n == 1) return 1;

    int curr = n;
    if(memo[curr]!=-1) return memo[curr];

    int catalan = 0;

    for(int i=0;i<n;i++) {
        catalan += ncatalan(i, memo)*ncatalan(n-i-1, memo);
    }

    memo[curr] = catalan;
    return memo[curr];
}

int countValidParenthesis(int n)
{
    unordered_map<int,int> memo;
    return ncatalan(n/2, memo);
}

int main(){
    int n;
    cin>>n;
    cout<<countValidParenthesis(n);
}
```

(28) Unique Binary Search Trees →

given integer N, return no. of unique BST that can be formed.

Eg n = 3 & lets say elements are [10, 20, 30]

⇒ 

∴ For n = 3, the result is 5.

∴ The catalan number gives us the result.

code →

```
class Solution {
public:

    int uniqueBST(int n, vector<int>& memo)
    {
        if(n==0||n==1) return 1;

        if( memo[n]!=-1)return memo[n];

        int ans = 0;
        for(int i=0;i<n;i++)
            ans += uniqueBST(i,memo)*uniqueBST(n-i-1,memo);

        return  memo[n] = ans;
    }

    int numTrees(int n) {
        vector<int> memo(n+1,-1);
        return uniqueBST(n, memo);
    }
};
```