Dynamic Programming - 2

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

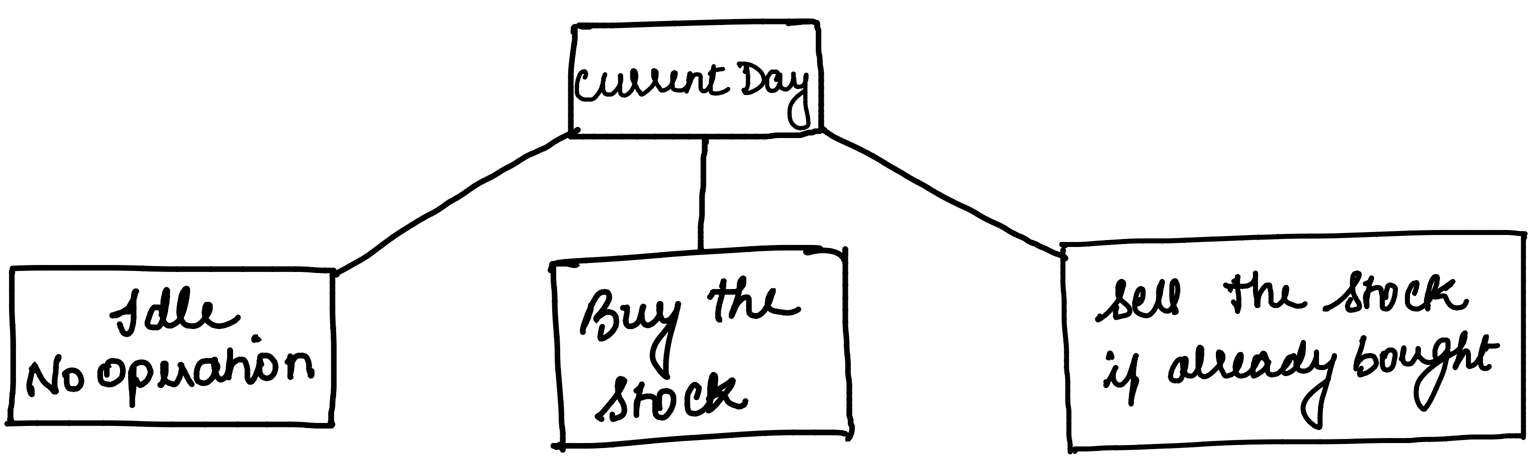
Eg

prices = [7, 1, 5, 3, 6, 4]
           0 1 2 3 4 5

→ we get maxProfit when we buy at day 0 & sell on day 4

⇒ 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.

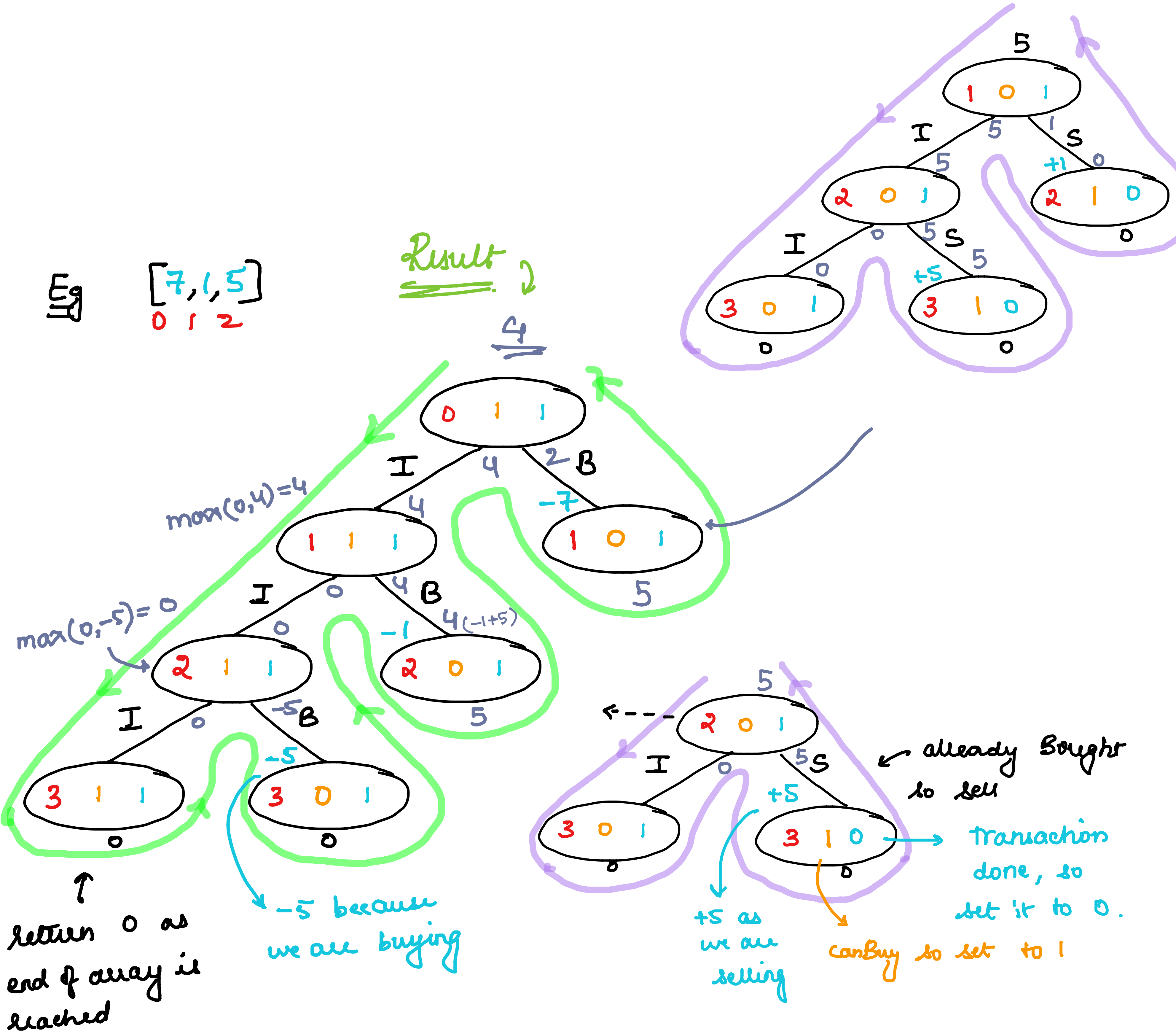
→ to handle these cases, we use a variable called canBuy.

→ 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
4
max(0,4)=4
max(0,-5)=0
I
B
S
-5 because we are buying
return 0 as end of array is reached
already bought so sell
transaction done, so set it to 0.
+5 as we are selling
canBuy so set to 1

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 = $\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \\ \end{array}$
$[7, 1, 5, 3, 6, 4]$

↳ 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.

(Line 6)

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

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

(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)

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

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

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

code →

Deduct the fee from the selling day's amount.

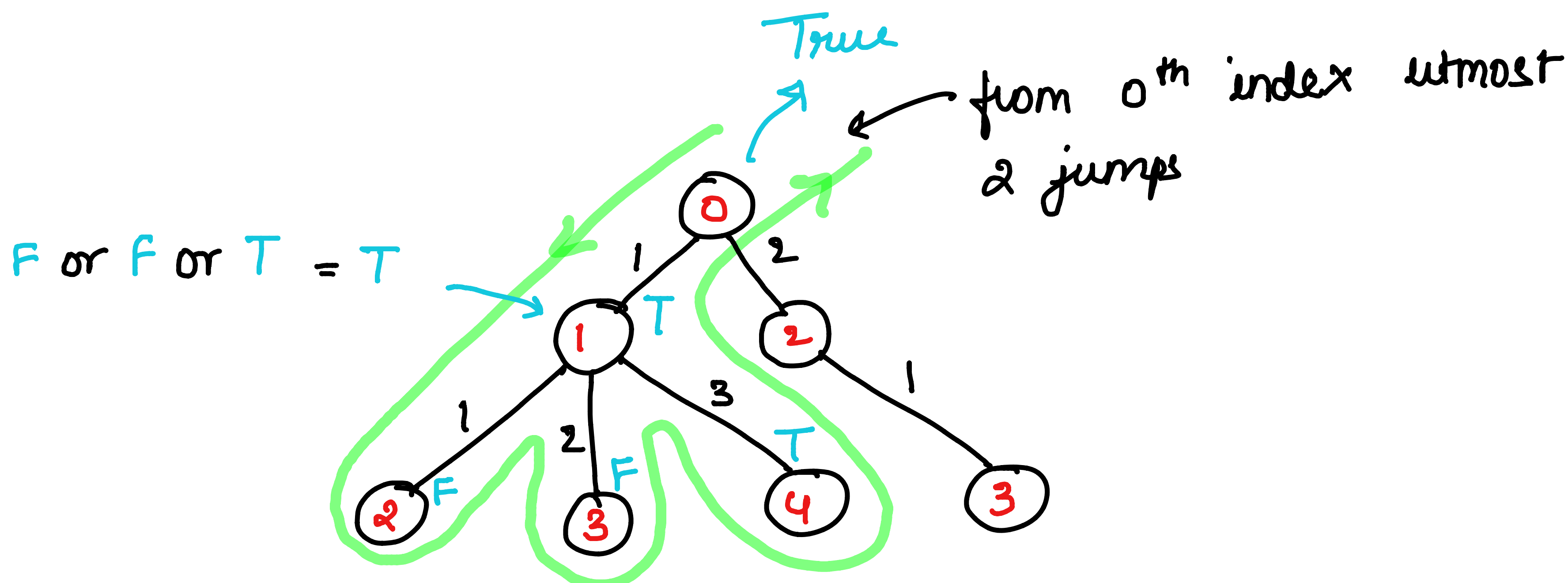
(Line 18)

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

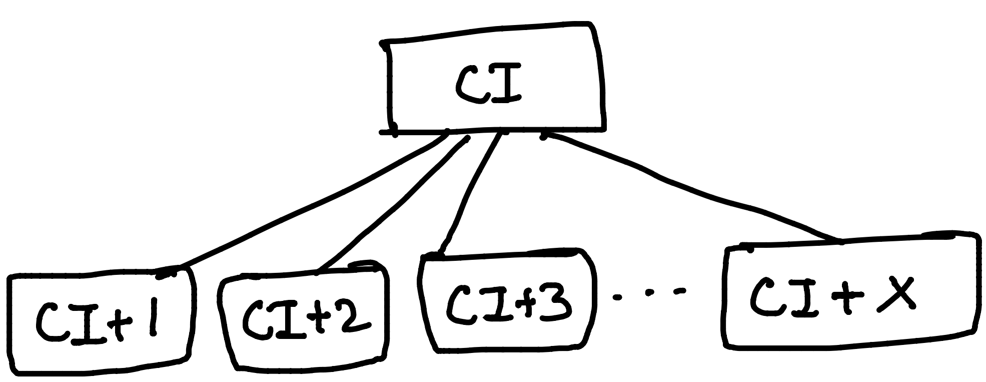
(22) Jump Game →

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, [_ _ $\underset{CI}{X}$ _ _ _]



can be implemented using a for loop.

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

$Tc \rightarrow O(\underbrace{max(nums[i])}_{\text{max time for forloop.}} * n)$

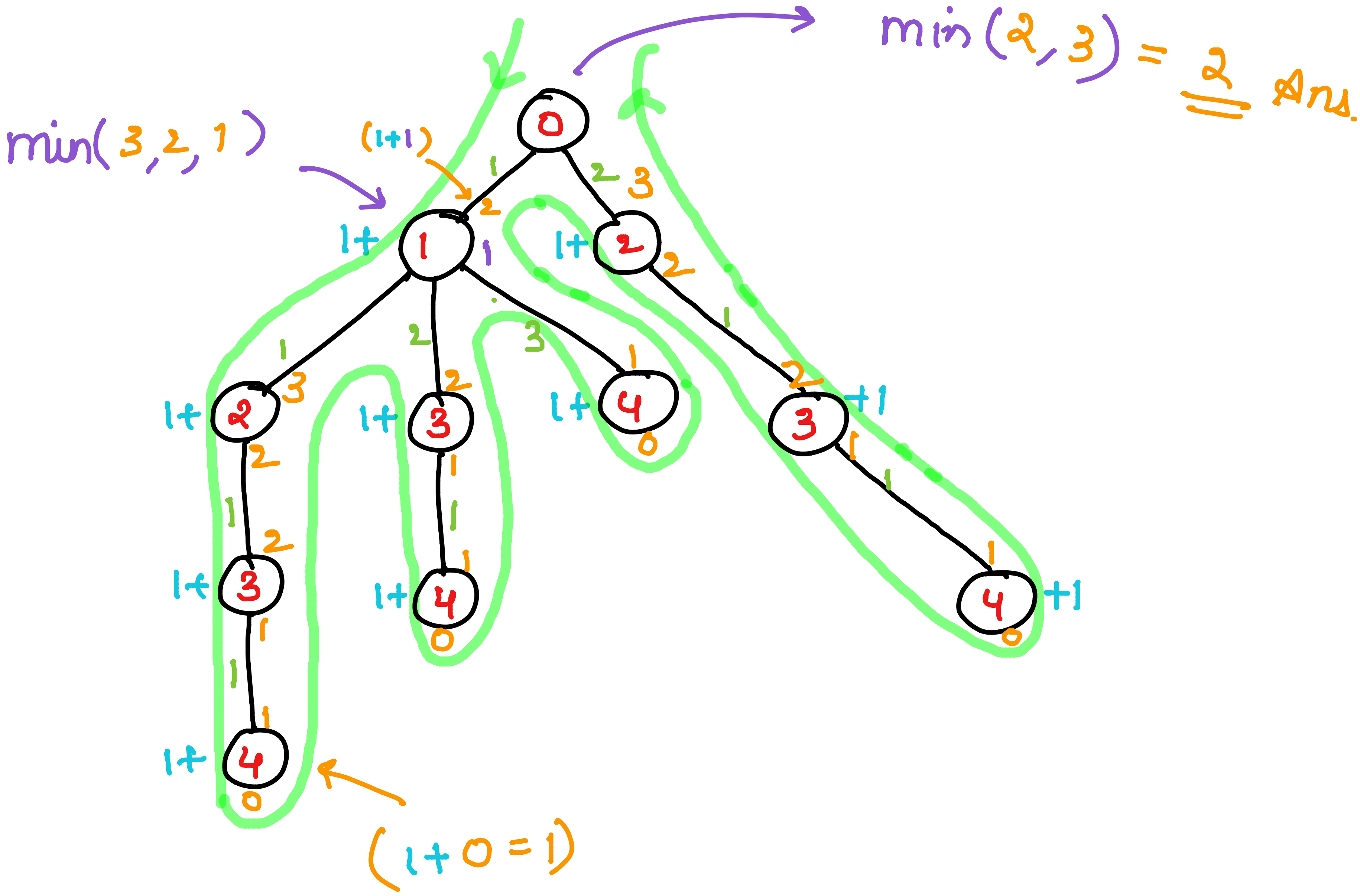
code →

```
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.

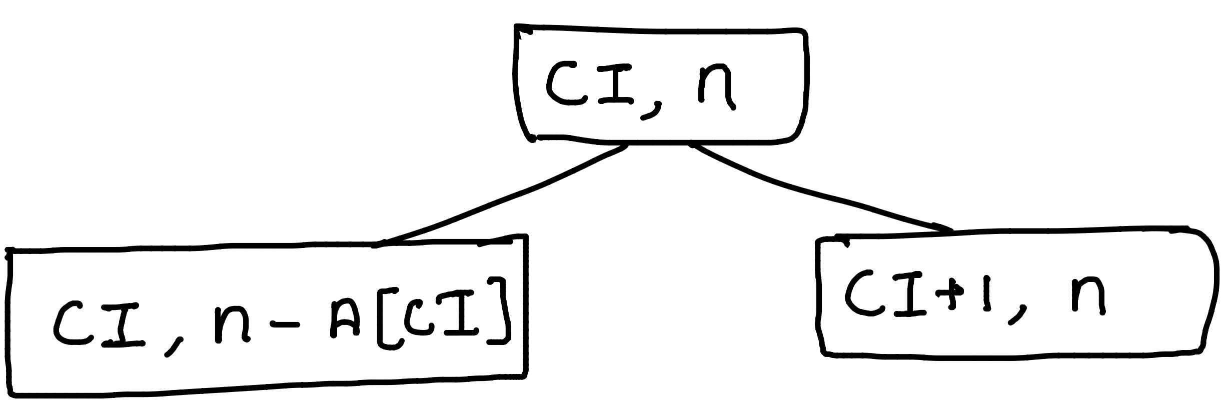
Eg n = 8 then no. of ways to create 8 from [3, 5, 10] is 1. (3+5)

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)

∴ lets 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,

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

→ $C_0 = \underline{\underline{1}}.$

→ $C_1 = \underline{\underline{1}}.$

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

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

→ $C_4 = C_0 C_3 + C_1 C_2 + C_2 C_1 + C_3 C_0$
$= 1 \cdot 5 + 1 \cdot 2 + 2 \cdot 1 + 5 \cdot 1 = \underline{\underline{14}}.$

App^{n}s →

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

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

↳ 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

( )( )( ) ⇒

( ) → ((( ))), (( )( ))
nothing is outside
∴ C_0 & 2 types in ∴ C_2
⇒ $C_0 \cdot C_2$

\+

( ) → (( ))( ), (( ))( )
identical &
one expression inside C_1
& one outside C_1
⇒ $C_1 \cdot C_1$

\+

( ) → ( )(( )), ( )( )( )
both expressions outside ∴ C_2
& nothing inside ∴ C_0
⇒ $C_2 \cdot C_0$

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

Code →

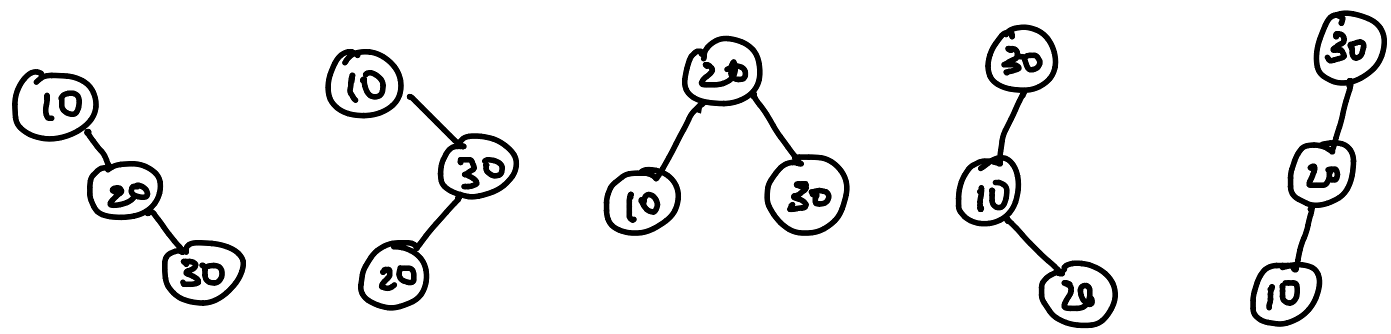
```cpp
#include<bits/stdc++.h>
using namespace std;

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 $\underline{\underline{5}}$.

∴ The catalan number gives us the result.

code →

```cpp
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);
    }
};
```