17/01/2024

DYNAMIC PROGRAMMING
CLASS - 6

1. Guess Number Higher or Lower II (Leetcode-375)

Merge Interval/Partitioning Pattern

Problem statement:
We are playing the Guessing Game. The game will work as follows:

1. *I pick a number between **1** and **n**.*

2. *You guess a number.*

3. *If you guess the right number, **you win the game**.*

4. *If you guess the wrong number, then I will tell you whether the number I picked is **higher or lower**, and you will continue guessing.*

5. *Every time you guess a wrong number **x**, you will pay **x** dollars. If you run out of money, **you lose the game**.*

*Given a particular n, return the minimum amount of money you need to **guarantee a win regardless of what number I pick**.*

What

Lets we have a Range

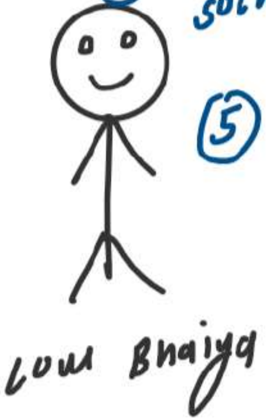
[i, j]

we are Breaking into Two part

[i, mid] [mid+1, j]

Combined Both part with a particular operations to get the Ans

Qustion Hai Kya -1 samJHTE Hai



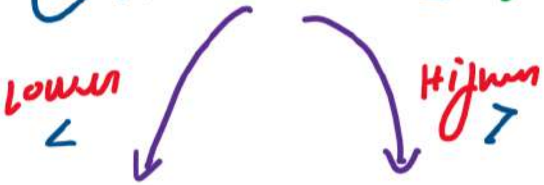
① Numbur socn liya Hai

⑤ Actua Nober kya Hai OR Lakshaya Bhaiya Ko kya panality Deni Hai YEH Lowi Bhaiya Batayenge According to WRONG Guess

② Numbur guss kar liya

③ Rijht Guess (WIN)

④ WRONG Guess (pay panality) ⑥



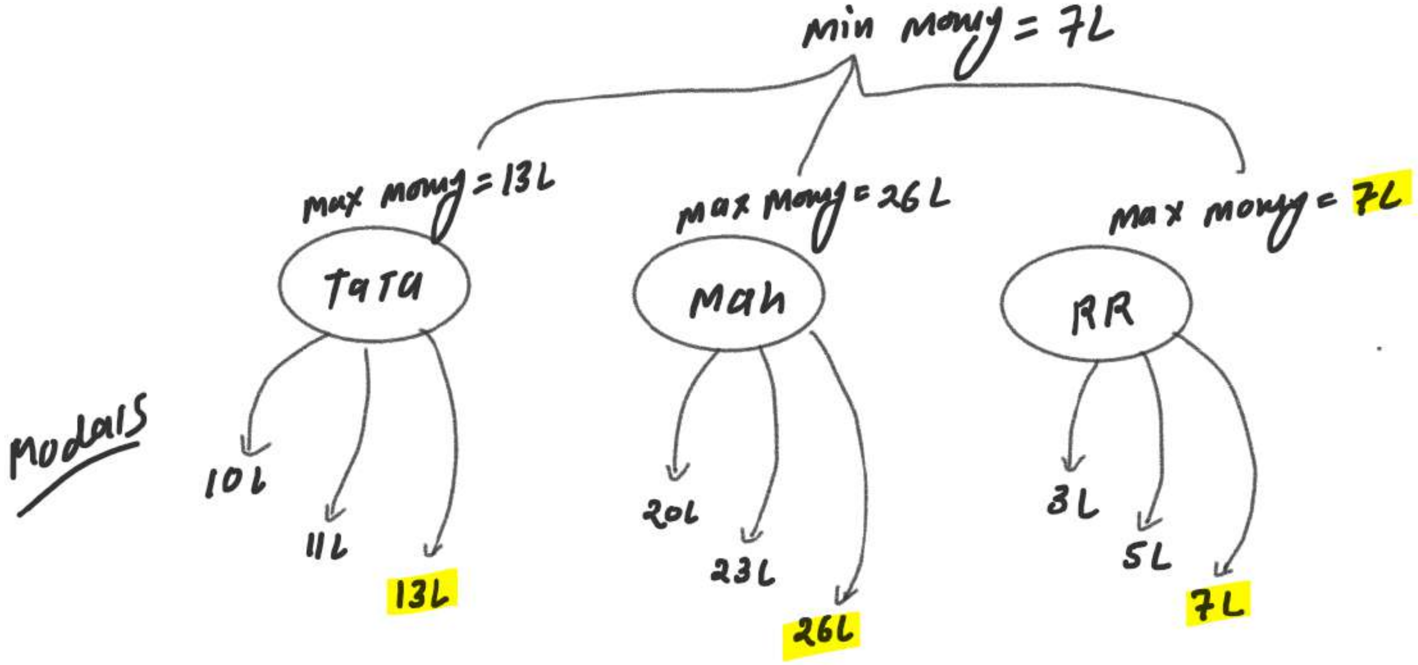
Find kya karna Hai ?

Minimum Amount kitna Ho Jisse Lakshay Bhaiya ki win Hone ke guarantee Ho

Qustion Hai Kya ?? SamJHTE Hai

Lakshay Bhaiya want to buy a Top modal Car.

→ To Bhaiya Ko Kitne minimum Money Ki Need Hai?



ANS

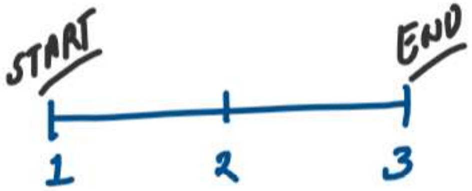
7 Lakh

Question Hai Kya <3 DRY RUN

Input: N = 3

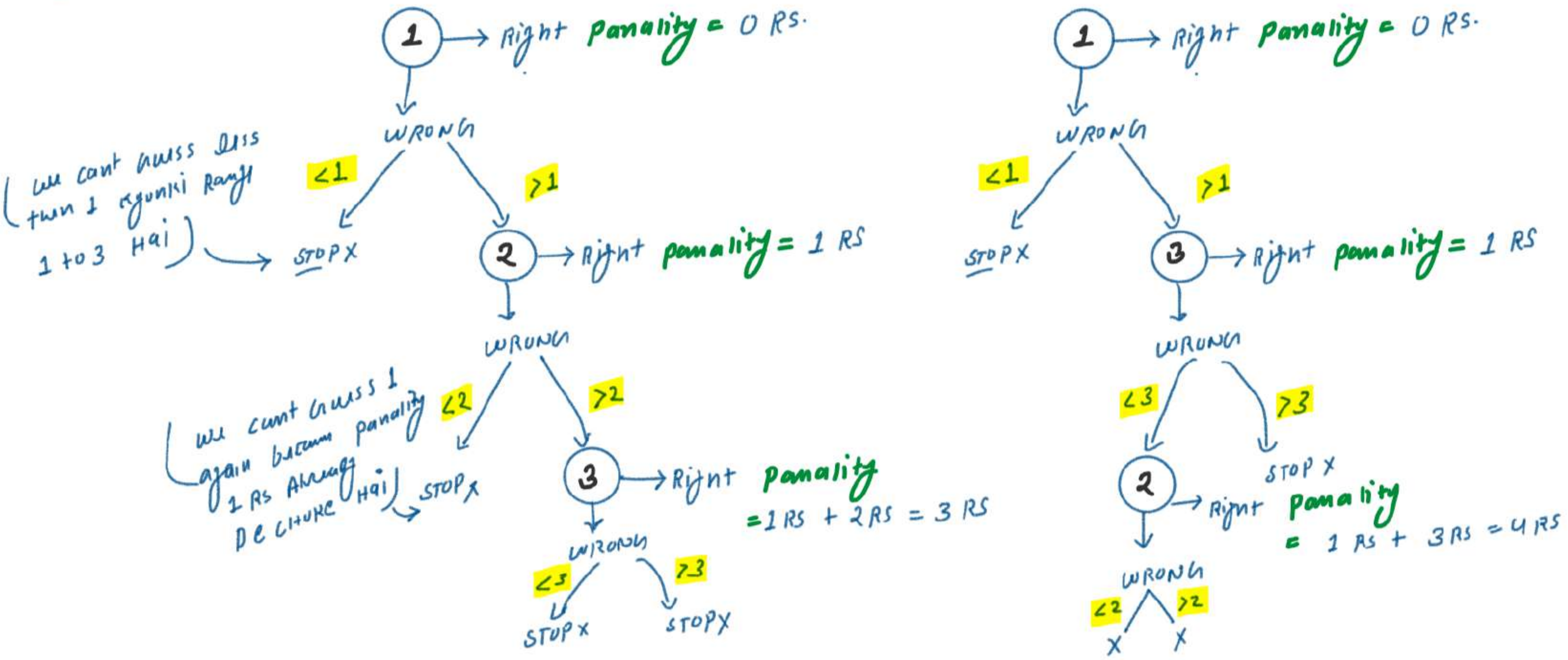
Output: 2

Range ∈ [1, N] ⇒ [1, 3]

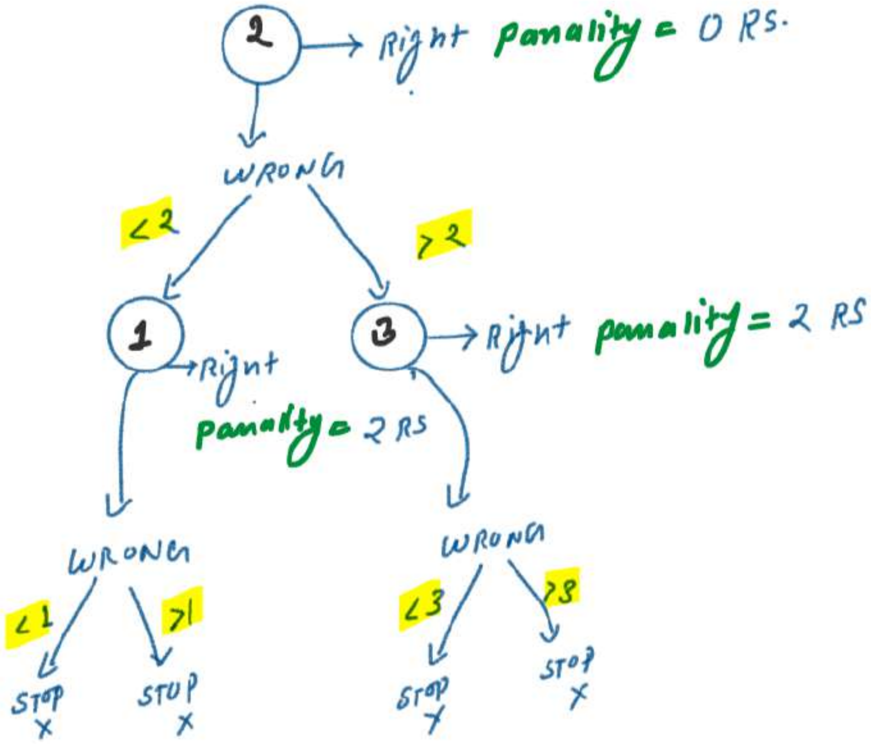


Case 1 start guessing from 1 Rs.

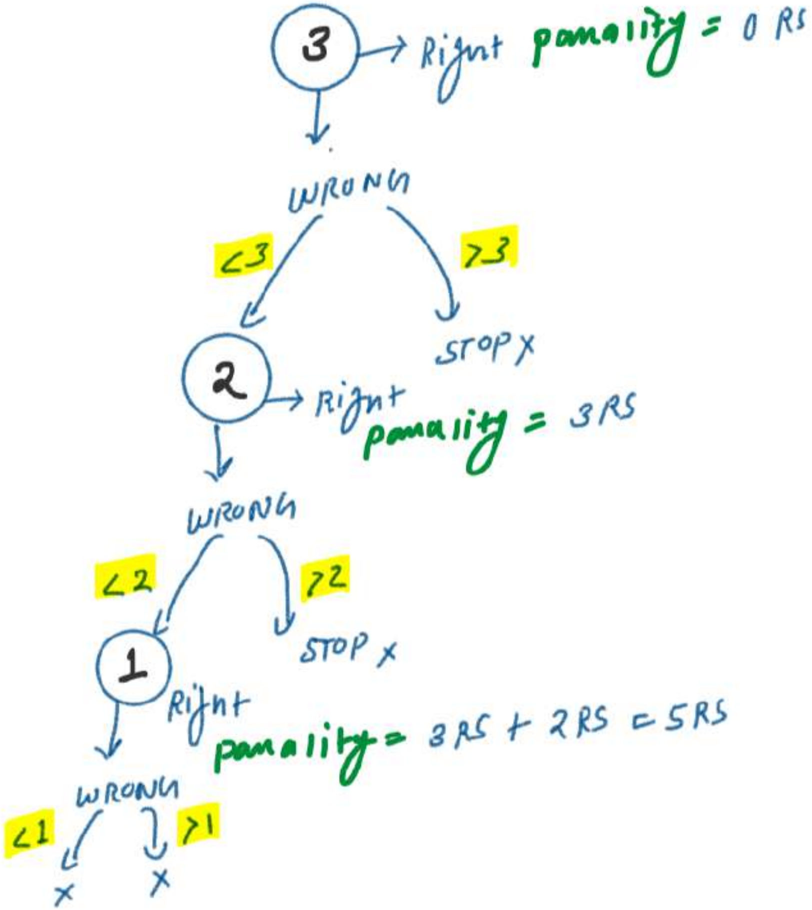
Case 2 start guessing from 1 Rs.


1
Right Panality = 0 Rs.
WRONG
<1
>1
(we cant guess less than 1 kyunki Range 1 to 3 Hai)
STOP X
2
Right panality = 1 RS
WRONG
(we cant guess 1 again becaus panality 1 Rs Already de chuke Hai)
<2
>2
STOP X
3
Right panality =1 RS + 2 RS = 3 RS
WRONG
<3
>3
STOP X
STOP X
1
Right Panality = 0 RS.
WRONG
<1
>1
STOP X
3
Right panality = 1 RS
WRONG
<3
>3
STOP X
2
Right Panality = 1 Rs + 3 Rs = 4 RS
WRONG
<2
>2
X
X

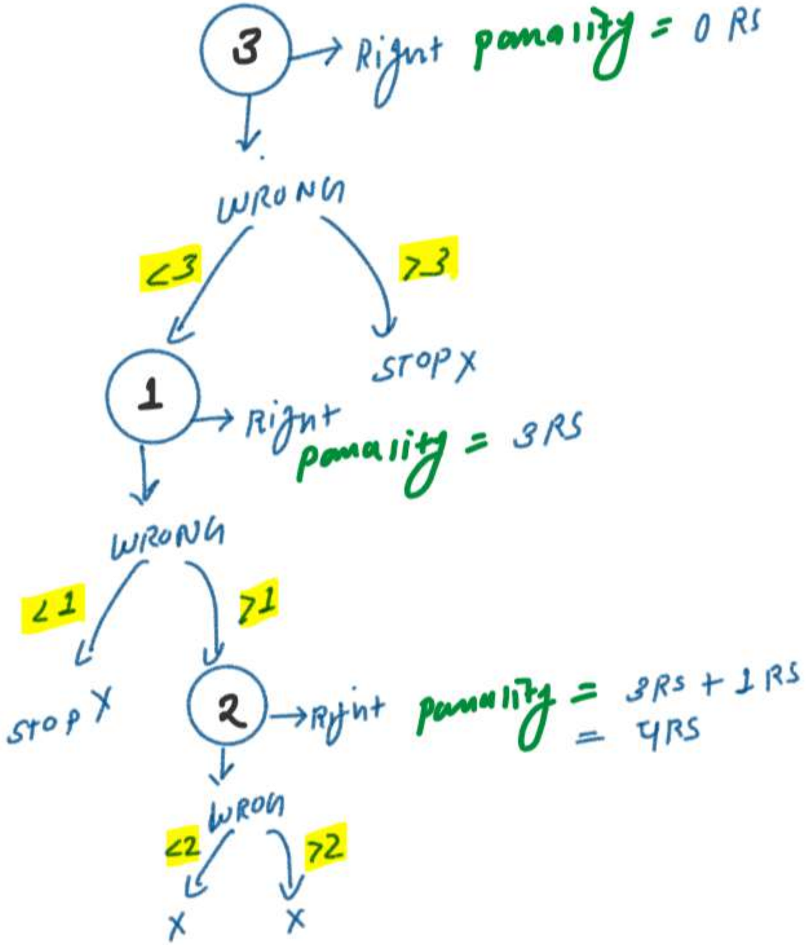
Case3 start Guessing from 2 RS.



Case4 start Guessing from 3 RS.



case5 start guessing from 3 RS.

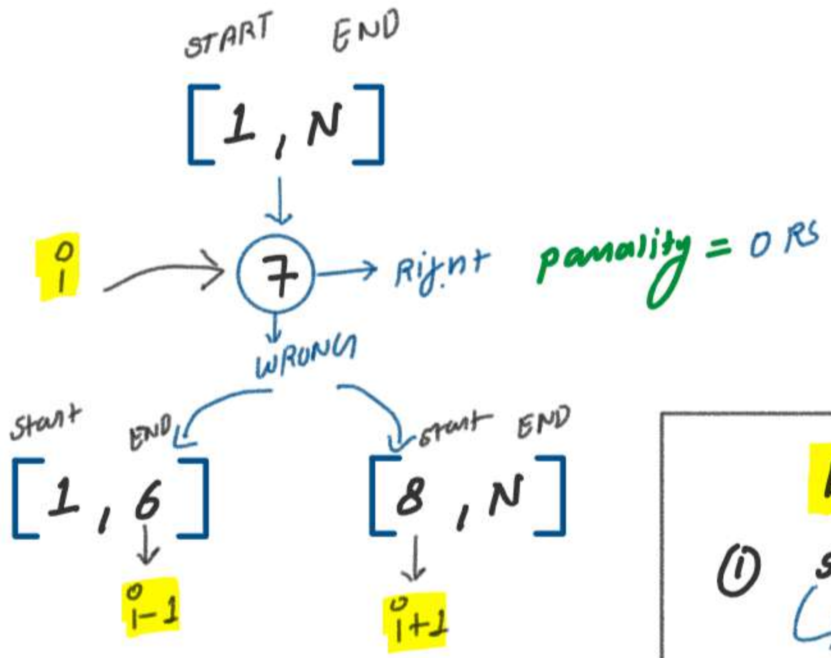


Ans	max RS
case1	3 RS
case2	4 RS
case3	2 RS
case4	5 RS
case5	4 RS

min RS

$\boxed{2 \text{ RS}}$

Final output



Again, we can guess from these Ranges

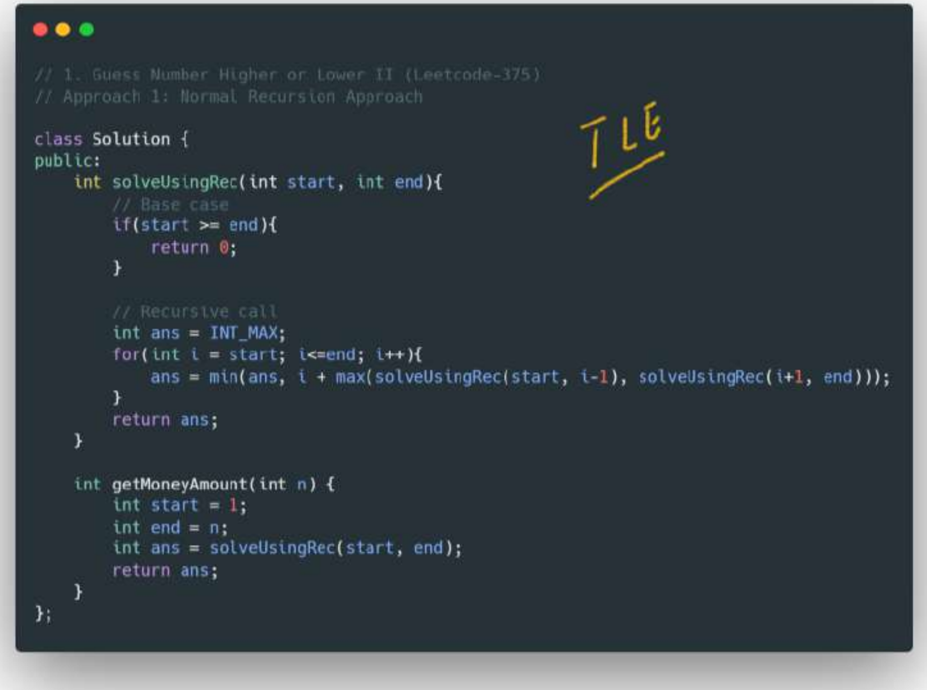
RECURSIVE RELATION FOR EACH NUMBER b/w Range

$$Ans = min(Ans, i + max(f(start, i-1), f(i+1, END)));$$

Base cases

1. Start > END
   - Out of Range Guessing
   - panality = 0
2. Start == END
   - Sirf EK hi Number Hai
   - panality = 0

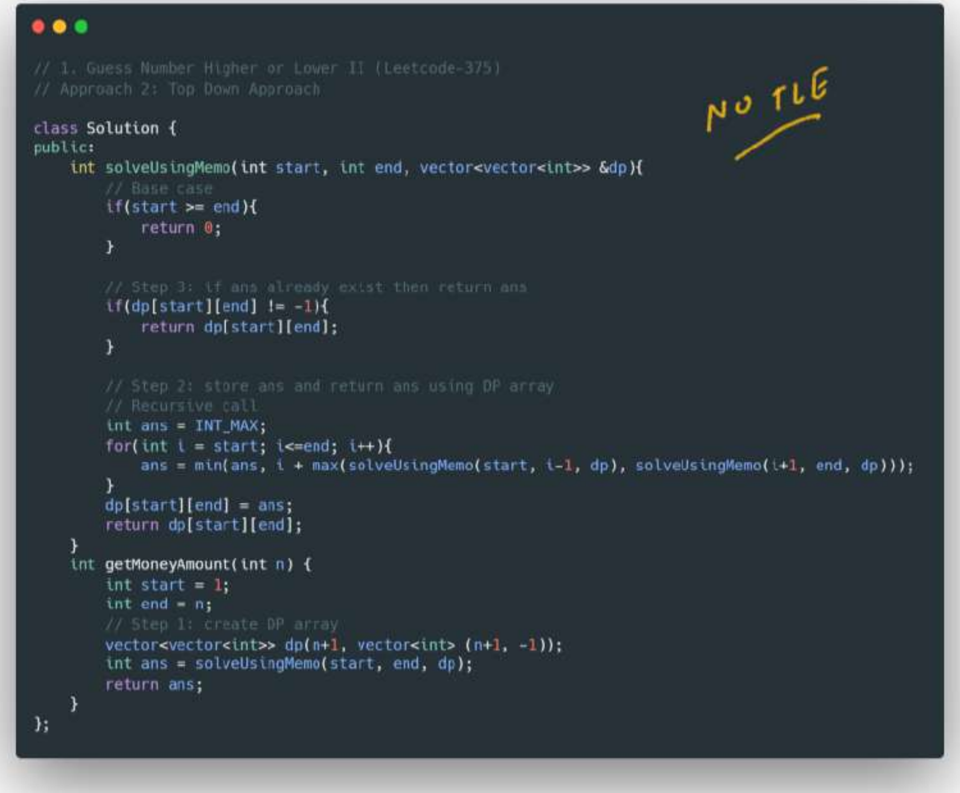
Approach 1: Recursion



```
// 1. Guess Number Higher or Lower II (Leetcode-375)
// Approach 1: Normal Recursion Approach

class Solution {
public:
    int solveUsingRec(int start, int end){
        // Base case
        if(start >= end){
            return 0;
        }

        // Recursive call
        int ans = INT_MAX;
        for(int i = start; i<=end; i++){
            ans = min(ans, i + max(solveUsingRec(start, i-1), solveUsingRec(i+1, end)));
        }
        return ans;
    }

    int getMoneyAmount(int n) {
        int start = 1;
        int end = n;
        int ans = solveUsingRec(start, end);
        return ans;
    }
};
```

TLE

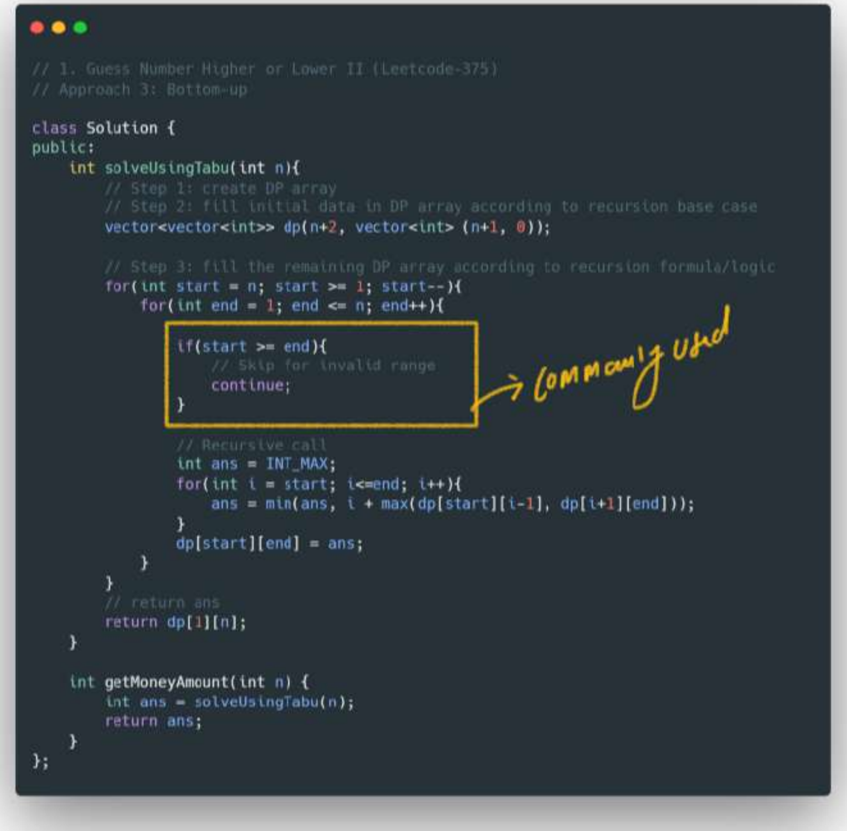
Approach 2: Top Down



```
// 1. Guess Number Higher or Lower II (Leetcode-375)
// Approach 2: Top Down Approach

class Solution {
public:
    int solveUsingMemo(int start, int end, vector<vector<int>> &dp){
        // Base case
        if(start >= end){
            return 0;
        }

        // Step 3: if ans already exist then return ans
        if(dp[start][end] != -1){
            return dp[start][end];
        }

        // Step 2: store ans and return ans using DP array
        // Recursive call
        int ans = INT_MAX;
        for(int i = start; i<=end; i++){
            ans = min(ans, i + max(solveUsingMemo(start, i-1, dp), solveUsingMemo(i+1, end, dp)));
        }
        dp[start][end] = ans;
        return dp[start][end];
    }
    int getMoneyAmount(int n) {
        int start = 1;
        int end = n;
        // Step 1: create DP array
        vector<vector<int>> dp(n+1, vector<int> (n+1, -1));
        int ans = solveUsingMemo(start, end, dp);
        return ans;
    }
};
```

NO TLE

Approach 3: Bottom Up



```cpp
// 1. Guess Number Higher or Lower II (Leetcode-375)
// Approach 3: Bottom-up

class Solution {
public:
    int solveUsingTabu(int n){
        // Step 1: create DP array
        // Step 2: fill initial data in DP array according to recursion base case
        vector<vector<int>> dp(n+2, vector<int> (n+1, 0));

        // Step 3: fill the remaining DP array according to recursion formula/logic
        for(int start = n; start >= 1; start--){
            for(int end = 1; end <= n; end++){

                if(start >= end){
                    // Skip for invalid range
                    continue;
                }

                // Recursive call
                int ans = INT_MAX;
                for(int i = start; i<=end; i++){
                    ans = min(ans, i + max(dp[start][i-1], dp[i+1][end]));
                }
                dp[start][end] = ans;
            }
        }
        // return ans
        return dp[1][n];
    }

    int getMoneyAmount(int n) {
        int ans = solveUsingTabu(n);
        return ans;
    }
};
```

→ commonly used

Let N=3

Row = 5 ↓ Col = 4 ↓

DP(N+2, vector<int> (n+1, 0))

columns

Rows

	0	1	2	3
0	0	0	0	0
1	0	0	1	2
2	0	0	0	2
3	0	0	0	0
4	0	0	0	0

ST = 3 EN=1 skip
EN=2 skip
EN=3 skip

ST = 2 EN=1 skip
EN=2 skip
EN=3 ✓

ST=1 EN=1 skip
EN=2 ✓
EN=3 ✓

📂 2. Minimum Cost Tree From Leaf Values (Leetcode-1130)

Merge Interval/Partitioning Pattern

Problem statement:
*Given an array **ARR** of positive integers, consider all binary trees such that:*

1. *Each node has either 0 or 2 children;*
2. *The values of ARR correspond to the values of each **leaf** in an in-order traversal of the tree.*
3. *The value of each **non-leaf** node is equal to the product of the largest leaf value in its left and right subtree, respectively.*

Among all possible binary trees considered, return the smallest possible sum of the values of each non-leaf node.
*It is guaranteed this sum fits into a **32-bit** integer.*

Note: *A node is a leaf if and only if it has zero children.*

CON1 we have a B.T. → child Nodes = 0
→ child Nodes = 2

CON2 Get leaf values using Inorder Traversal (LNR) from input arr

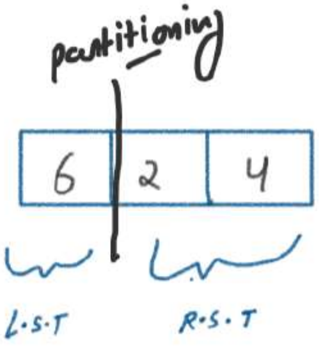
CON3 Build Non-leaf Node → max val of L.S.T × max val of R.S.T

Find kya karna hai? → [Ans1 ANS2 AN3] → min sum print karna hai

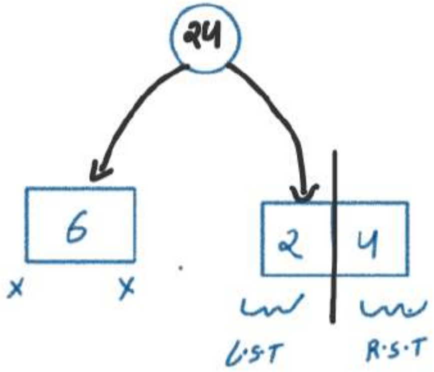
Explanation

Case 1

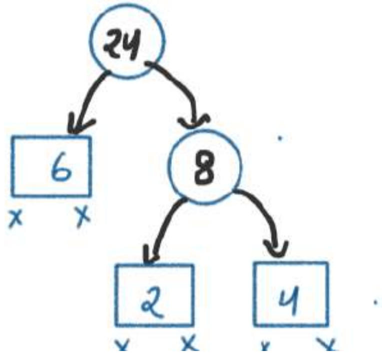
Example 1:
Input: ARR = [6,2,4]
Output: 32



Max value of L.S.T. = 6
Max value of R.S.T = 4
NON-Leaf Node ⇒ 6 × 4 = 24

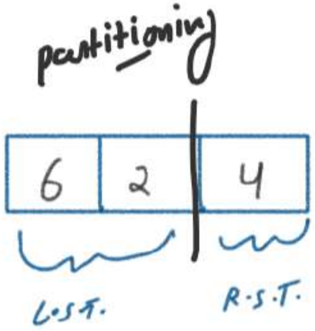


Max value of L.S.T = 2
Max value of R.S.T = 4
NON-Leaf Node ⇒ 2 × 4 = 8



[Sum of NON-Leaf Node = 24 + 8
= 32]

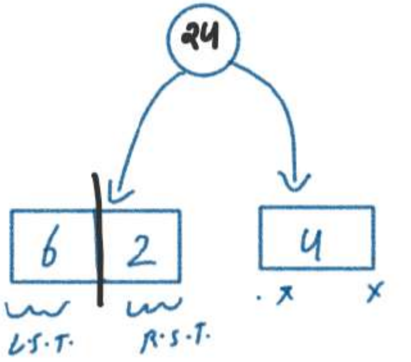
Case2



max value of L.S.T. = 6

max value of R.S.T = 4

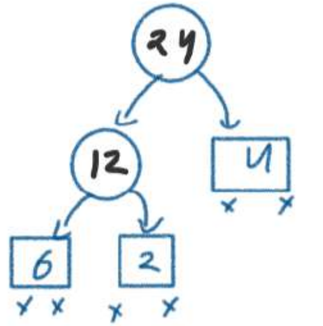
NON-Leaf Node ⇒ 6 × 4 = 24



max value of L.S.T. = 6

max value of R.S.T = 2

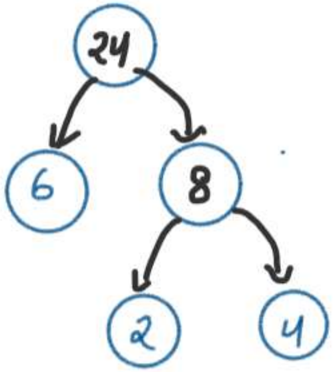
NON-Leaf Node ⇒ 6 × 2 = 12



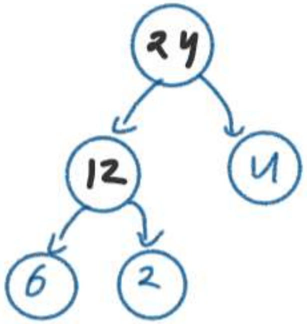
[Sum of NON-Leaf Node = 24 + 12
= 36]

There are two possible binary trees shown

B.T.1



B.T.2



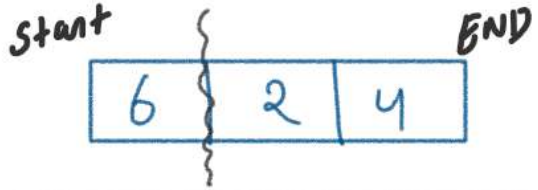
ANS1 = 24 + 8
= 32

ANS2 = 24 + 12
= 36

} min Ans = 32

↳ Final output

REC. call



sum = (maxL.s.T x MaxR.s.T) + solve(Left) + solve (Right)

Ans = min(Ans, sum)

Pre-computation

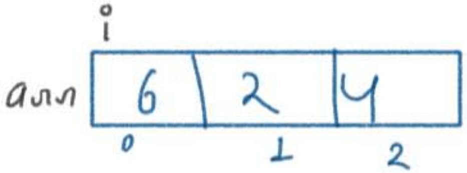
→ why

Diff.^{n} – 2 partition jab karenge To NON leaf Node Build karne ke liye Home Diff.^{n}-2 partition ki max val ki need Hogi

$$\text{Map} < \underbrace{\text{pair} < \text{int}, \text{int} >}_{\text{KEY}}, \underbrace{\text{int}}_{\text{VALUE}} > \text{maxi};$$

KEY	VALUE
{0,0}	6
{0,1}	6
{0,2}	6
{1,1}	2
{1,2}	4
{2,2}	4

← maxVal



All possible Partitions of Input arr →

{0,0} → [6] → max = 6

{0,1} → [6 | 2] → max = 6

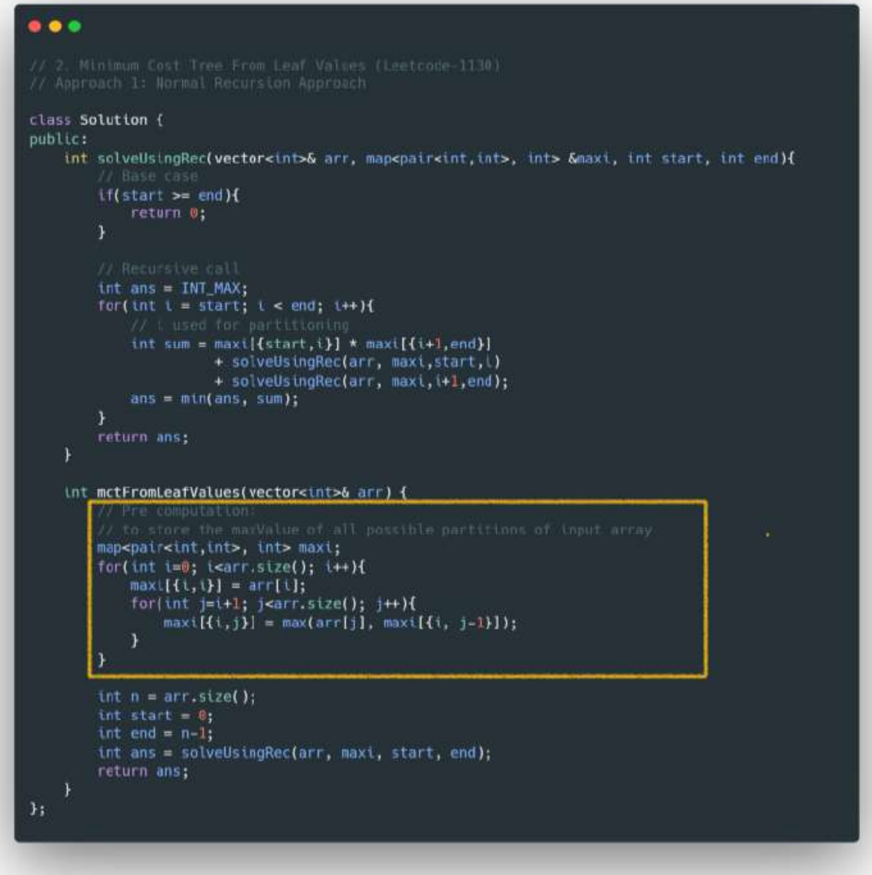
{0,2} → [6 | 2 | 4] → max = 6

{1,1} [2] → max = 2

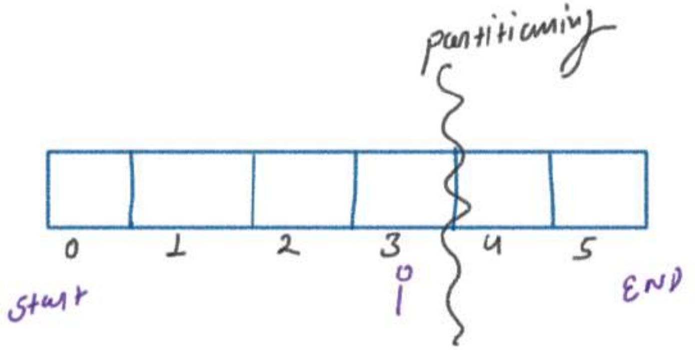
{1,2} [2 | 4] → max = 4

{2,2} [4] → max = 4

Approach 1: Recursion

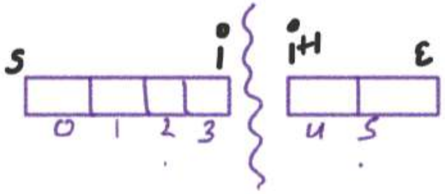


```cpp
// 2. Minimum Cost Tree From Leaf Values (Leetcode-1130)
// Approach 1: Normal Recursion Approach

class Solution {
public:
    int solveUsingRec(vector<int>& arr, map<pair<int,int>, int> &maxi, int start, int end){
        // Base case
        if(start >= end){
            return 0;
        }

        // Recursive call
        int ans = INT_MAX;
        for(int i = start; i < end; i++){
            // i used for partitioning
            int sum = maxi[{start,i}] * maxi[{i+1,end}]
                    + solveUsingRec(arr, maxi,start,i)
                    + solveUsingRec(arr, maxi,i+1,end);
            ans = min(ans, sum);
        }
        return ans;
    }

    int mctFromLeafValues(vector<int>& arr) {
        // Pre computation:
        // to store the maxValue of all possible partitions of input array
        map<pair<int,int>, int> maxi;
        for(int i=0; i<arr.size(); i++){
            maxi[{i,i}] = arr[i];
            for(int j=i+1; j<arr.size(); j++){
                maxi[{i,j}] = max(arr[j], maxi[{i, j-1}]);
            }
        }

        int n = arr.size();
        int start = 0;
        int end = n-1;
        int ans = solveUsingRec(arr, maxi, start, end);
        return ans;
    }
};
```

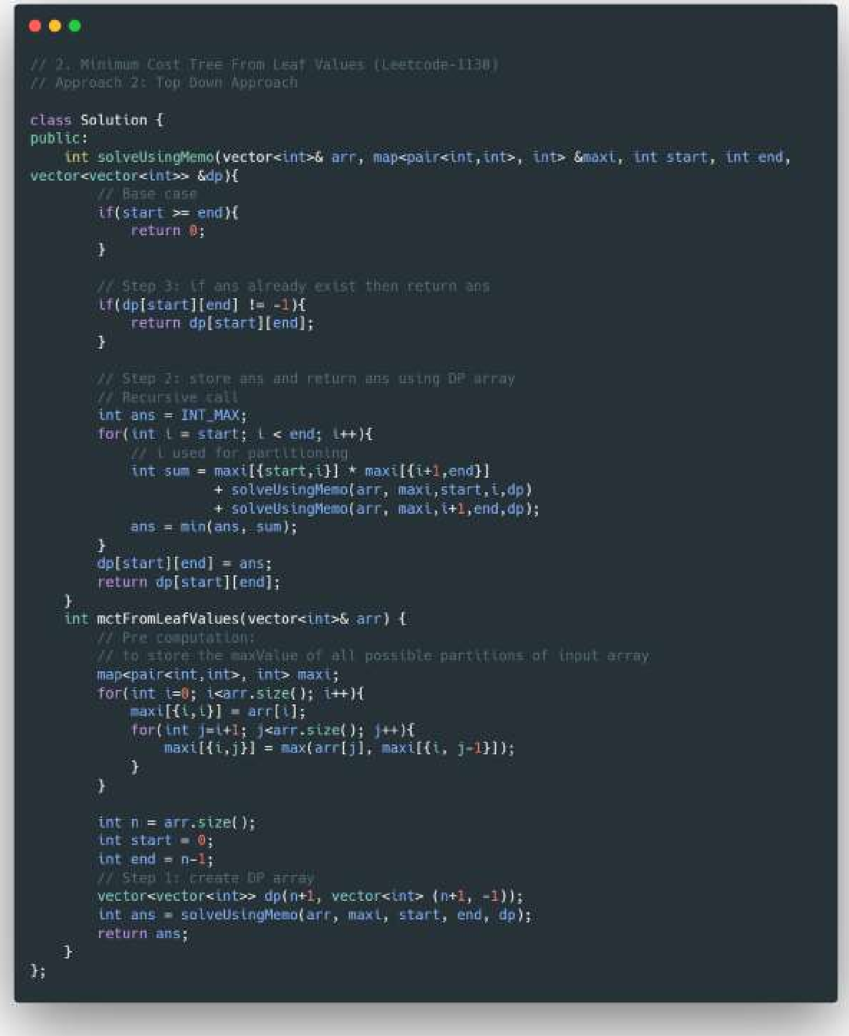


way1 solve(start, i) + solve(i+1, End)

Lets i = 3
s = 0
e = 5

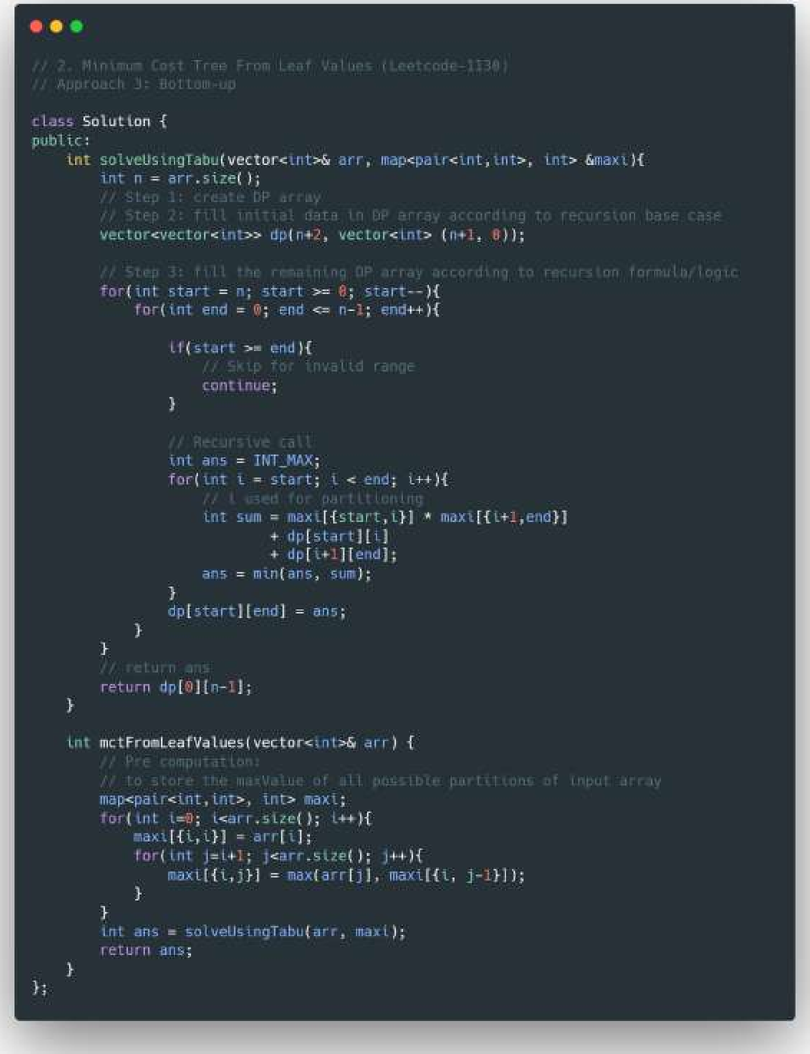


Approach 2: Top Down



```cpp
// 2. Minimum Cost Tree From Leaf Values (Leetcode-1130)
// Approach 2: Top Down Approach

class Solution {
public:
    int solveUsingMemo(vector<int>& arr, map<pair<int,int>, int> &maxi, int start, int end,
vector<vector<int>> &dp){
        // Base case
        if(start >= end){
            return 0;
        }

        // Step 3: if ans already exist then return ans
        if(dp[start][end] != -1){
            return dp[start][end];
        }

        // Step 2: store ans and return ans using DP array
        // Recursive call
        int ans = INT_MAX;
        for(int i = start; i < end; i++){
            // i used for partitioning
            int sum = maxi[{start,i}] * maxi[{i+1,end}]
                      + solveUsingMemo(arr, maxi,start,i,dp)
                      + solveUsingMemo(arr, maxi,i+1,end,dp);
            ans = min(ans, sum);
        }
        dp[start][end] = ans;
        return dp[start][end];
    }
    int mctFromLeafValues(vector<int>& arr) {
        // Pre computation:
        // to store the maxValue of all possible partitions of input array
        map<pair<int,int>, int> maxi;
        for(int i=0; i<arr.size(); i++){
            maxi[{i,i}] = arr[i];
            for(int j=i+1; j<arr.size(); j++){
                maxi[{i,j}] = max(arr[j], maxi[{i, j-1}]);
            }
        }

        int n = arr.size();
        int start = 0;
        int end = n-1;
        // Step 1: create DP array
        vector<vector<int>> dp(n+1, vector<int> (n+1, -1));
        int ans = solveUsingMemo(arr, maxi, start, end, dp);
        return ans;
    }
};
```

Approach 3: Bottom Up



```cpp
// 2. Minimum Cost Tree From Leaf Values (Leetcode-1130)
// Approach 3: Bottom-up

class Solution {
public:
    int solveUsingTabu(vector<int>& arr, map<pair<int,int>, int> &maxi){
        int n = arr.size();
        // Step 1: create DP array
        // Step 2: fill initial data in DP array according to recursion base case
        vector<vector<int>> dp(n+2, vector<int> (n+1, 0));

        // Step 3: fill the remaining DP array according to recursion formula/logic
        for(int start = n; start >= 0; start--){
            for(int end = 0; end <= n-1; end++){

                if(start >= end){
                    // Skip for invalid range
                    continue;
                }

                // Recursive call
                int ans = INT_MAX;
                for(int i = start; i < end; i++){
                    // i used for partitioning
                    int sum = maxi[{start,i}] * maxi[{i+1,end}]
                            + dp[start][i]
                            + dp[i+1][end];
                    ans = min(ans, sum);
                }
                dp[start][end] = ans;
            }
        }
        // return ans
        return dp[0][n-1];
    }

    int mctFromLeafValues(vector<int>& arr) {
        // Pre computation:
        // to store the maxValue of all possible partitions of input array
        map<pair<int,int>, int> maxi;
        for(int i=0; i<arr.size(); i++){
            maxi[{i,i}] = arr[i];
            for(int j=i+1; j<arr.size(); j++){
                maxi[{i,j}] = max(arr[j], maxi[{i, j-1}]);
            }
        }
        int ans = solveUsingTabu(arr, maxi);
        return ans;
    }
};
```