### **Recursion:**

## **✅ Fix: Use the “Choice-Result-Return” Template**

This 3-step mindset works in **all recursive problems**, especially max/min/ways.

### **🔁 Step-by-step Recursion Mindset (CRR):**

1. **Choice**: At this step, what options do I have?
2. **Result**: What will I get by picking each option?
3. **Return**: From those results, what do I return (min/max/sum/etc.)?

### **🧠 Let's Take a Real Example: House Robber**

**Problem**: Given nums = [2,7,9,3,1], return the **maximum amount** you can rob **without robbing two adjacent houses**.

### **1. Choices:**

At index i, I can either:

* Rob house i → skip i+1, go to i+2
* Don’t rob house i → go to i+1

### **2. Result:**

If I rob → I get nums[i] + solve(i + 2)  
 If I skip → I get solve(i + 1)

### **3. Return:**

### return max(rob, skip);

### **So Final Recursion:**

int solve(int i) {  
 if (i >= nums.size()) return 0;  
  
 int rob = nums[i] + solve(i + 2);  
 int skip = solve(i + 1);  
  
 return max(rob, skip);  
}

Just plug in the CRR mindset.

### **👊 Now Apply CRR to Any Problem:**

* Climbing stairs → Choice: 1 step or 2 steps
* Coin change → Choice: pick coin or skip
* Min path sum → Choice: go down or right
* Max profit → Choice: buy/sell/skip

Same structure every time.

## **🧩 Problem**

Given a grid[m][n] where you can only move **right** or **down**, find the **minimum path sum** from the top-left (0,0) to bottom-right (m-1,n-1).

## **✅ CRR Breakdown**

### **✅ C → Choices**

From cell (i, j), you have two valid moves:

* Move **Down** to (i + 1, j)
* Move **Right** to (i, j + 1)

### **✅ R → Result of those choices**

You want the **minimum** path sum among these two paths. So:

int down = solve(i + 1, j);  
int right = solve(i, j + 1);

### **✅ R → Return**

You return the value of the current cell grid[i][j] plus the **minimum** of the two choices:

return grid[i][j] + min(down, right);

## **✅ Base Cases (very important for recursion)**

1. **Reached destination**:

if (i == m - 1 && j == n - 1) return grid[i][j];

1. **Out of bounds**:

if (i >= m || j >= n) return INT\_MAX;

## **🔁 Full CRR Template Recap**

int solve(int i, int j) {  
 if (i >= m || j >= n) return INT\_MAX;  
 if (i == m - 1 && j == n - 1) return grid[i][j];  
  
 int down = solve(i + 1, j);  
 int right = solve(i, j + 1);  
  
 return grid[i][j] + min(down, right);  
}

### **🧠 Problem Recap:**

You are given an m x n grid filled with non-negative numbers. Starting at the top-left, you can only move **right** or **down**. Your goal is to **minimize the sum** of all numbers along the path to the bottom-right.

## **✅ CRMA Breakdown for Minimum Path Sum**

### **✅ C – Choices**

From cell (i, j), you have **two choices**:

* Move **Down** → (i+1, j)
* Move **Right** → (i, j+1)

### **✅ R – Result of a choice**

Each choice gives you a **subproblem**:

* down = solve(i + 1, j) → min path sum from the cell below
* right = solve(i, j + 1) → min path sum from the cell to the right

Then combine it with the **current cell value**:

return grid[i][j] + min(down, right);

### **✅ M – Memoization**

Since (i, j) states repeat, memoize with a 2D dp table:

if (dp[i][j] != -1) return dp[i][j];

Use:

vector<vector<int>> dp(m, vector<int>(n, -1));

### **✅ A – Answer**

Call the function from the **starting cell**:

solve(0, 0)

### **Final Recursive Code with CRMA**

int solve(int i, int j, vector<vector<int>>& grid, vector<vector<int>>& dp) {  
 int m = grid.size(), n = grid[0].size();  
   
 if (i >= m || j >= n) return INT\_MAX; // out of bounds  
 if (i == m - 1 && j == n - 1) return grid[i][j]; // reached destination  
   
 if (dp[i][j] != -1) return dp[i][j];  
   
 int down = solve(i + 1, j, grid, dp);  
 int right = solve(i, j + 1, grid, dp);  
   
 return dp[i][j] = grid[i][j] + min(down, right);  
}

### **🟢 Intuition behind INT\_MAX**

We return INT\_MAX for out-of-bound cells because:

* We're using min(down, right)
* If you fall outside the grid, that path shouldn't be considered
* INT\_MAX ensures it never gets picked