**✅ When to Use 2 for Loops Before Calling DFS**

Use **2 nested for loops** when:

**🔹 1. You want to explore *all starting points* in a 2D matrix**

🎯 **Example Scenarios**:

* Count number of islands (DFS from every unvisited 1)
* Maximum area of an island
* Get maximum gold from any cell
* Find all connected components
* Search for multiple disconnected paths

cpp

CopyEdit

for (int i = 0; i < m; i++) {

for (int j = 0; j < n; j++) {

if (grid[i][j] == some\_condition) {

dfs(i, j, grid); // DFS from every valid start

}

}

}

**PATTERN 1**

**✅ SAMPLE PROBLEM: Count total paths from top-left (0,0) to bottom-right (n-1,n-1) in a maze**

**📌 Problem:**

Given a n x n grid maze (0 = blocked, 1 = open), count how many unique paths exist from (0,0) to (n-1,n-1) using **only 4 directions**: up, down, left, right.

**✅ CODE with INLINE EXPLANATION**

#include <iostream>

#include <vector>

using namespace std;

int countPaths(int row, int col, int n, vector<vector<int>>& maze) {

// ✅ 1. BASE CASE: Out of bounds or blocked → invalid path

if (row < 0 || col < 0 || row >= n || col >= n || maze[row][col] == 0)

return 0;

// ✅ 2. REACHED GOAL

if (row == n - 1 && col == n - 1)

return 1;

// ✅ 3. MARK CURRENT CELL AS VISITED

maze[row][col] = 0;

// ✅ 4. EXPLORE 4 DIRECTIONS (DFS)

int totalPaths = 0;

totalPaths += countPaths(row + 1, col, n, maze); // down

totalPaths += countPaths(row - 1, col, n, maze); // up

totalPaths += countPaths(row, col + 1, n, maze); // right

totalPaths += countPaths(row, col - 1, n, maze); // left

// ✅ 5. UNMARK CURRENT CELL (Backtrack)

maze[row][col] = 1;

// ✅ 6. RETURN TOTAL PATHS FOUND

return totalPaths;

}

int main() {

vector<vector<int>> maze = {

{1, 0, 0, 0, 0},

{1, 1, 1, 1, 1},

{1, 1, 1, 0, 1},

{0, 0, 0, 0, 1},

{0, 0, 0, 0, 1}

};

int n = maze.size();

int total = countPaths(0, 0, n, maze);

cout << "Total Paths from (0,0) to (" << n-1 << "," << n-1 << "): " << total << endl;

return 0;

}

**✅ So your current DFS logic already means:**

**“Do not revisit the same cell in the same path, but allow it in other paths.”**

This is the **correct behavior** for path-counting DFS.

**❗ But if your scenario changes like:**

“Once visited in **any** path, a cell should never be used again.” (i.e., **global visited**)

Then modify your logic to **not backtrack**:

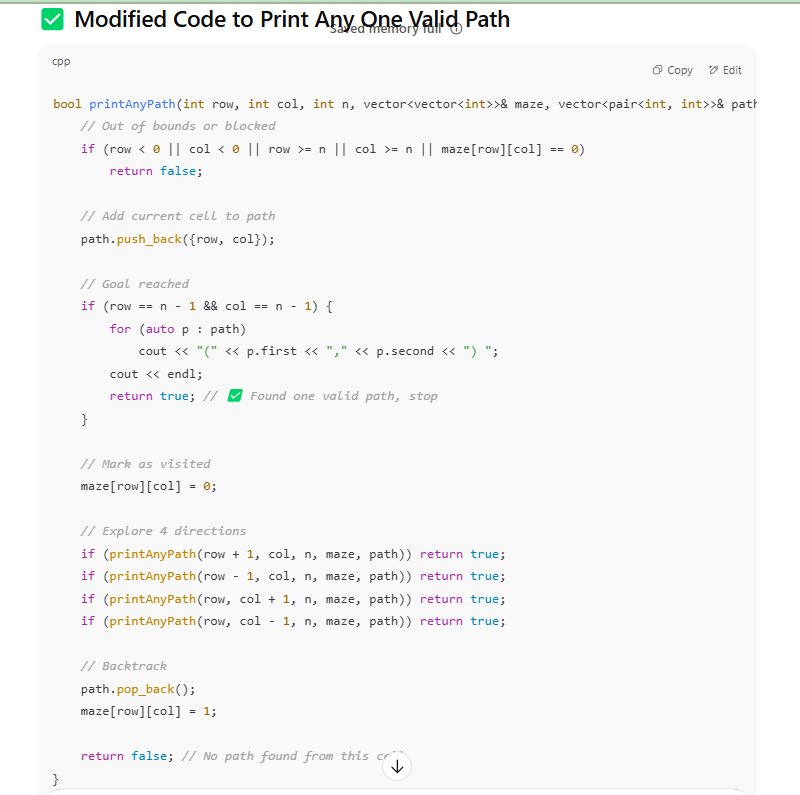
cpp

CopyEdit

maze[row][col] = 0; // Mark visited

...

// ❌ No backtracking (don't reset to 1)



**PATTERN 2**

**✅ 2. Finding Maximum / Minimum (gold collection, increasing path)**

➡️ *"What’s the max sum/value/length I can collect?"*  
💡 **Return max value as int**

* **Base Case**: Invalid cell → return 0
* **Recursive Step**: Try all directions, take max
* **Backtrack**: Yes (restore grid[i][j] after recursion)
* **Return**: current\_value + max(next moves)

int maxVal = 0;

maxVal = max(maxVal, dfs(...));

return grid[i][j] + maxVal;

class Solution {

public:

    int help(int r,int c,vector<vector<int>>&grid,int m,int n){

           if(r>=m || r<0 || c>=n || c<0 || grid[r][c]==0) return 0;

           int total=0;

           int temp=grid[r][c];

           grid[r][c]=0;

         total = max(total, help(r + 1, c, grid,m, n)); // down

         total = max(total, help(r - 1, c, grid,m, n)); // up

         total = max(total, help(r, c + 1, grid,m, n)); // right

         total = max(total, help(r, c - 1, grid,m, n)); // left

           grid[r][c]=temp;

           return temp+total;

    }

    int getMaximumGold(vector<vector<int>>& grid) {

        int maxi=0,m=grid.size(),n=grid[0].size();

        for(int i=0;i<m;i++){

            for(int j=0;j<n;j++){

                if(grid[i][j]!=0){

                int total=0;

                total=help(i,j,grid,m,n);

                maxi=max(maxi,total);

                }

            }

        }

        return maxi;

    }

};

**Finding maximum path**

**class Solution {**

**public:**

**int maxGold = 0;**

**vector<pair<int, int>> maxPath;**

**void help(int r, int c, vector<vector<int>>& grid, int m, int n, int currSum,**

**vector<pair<int, int>>& currPath) {**

**if (r < 0 || r >= m || c < 0 || c >= n || grid[r][c] == 0)**

**return;**

**// Collect gold**

**int gold = grid[r][c];**

**currSum += gold;**

**currPath.push\_back({r, c});**

**grid[r][c] = 0; // Mark visited**

**// Update maxGold and maxPath if this is the best so far**

**if (currSum > maxGold) {**

**maxGold = currSum;**

**maxPath = currPath;**

**}**

**// Explore 4 directions**

**help(r + 1, c, grid, m, n, currSum, currPath);**

**help(r - 1, c, grid, m, n, currSum, currPath);**

**help(r, c + 1, grid, m, n, currSum, currPath);**

**help(r, c - 1, grid, m, n, currSum, currPath);**

**// Backtrack**

**grid[r][c] = gold;**

**currPath.pop\_back();**

**}**

**int getMaximumGold(vector<vector<int>>& grid) {**

**int m = grid.size(), n = grid[0].size();**

**for (int i = 0; i < m; i++) {**

**for (int j = 0; j < n; j++) {**

**if (grid[i][j] != 0) {**

**vector<pair<int, int>> currPath;**

**help(i, j, grid, m, n, 0, currPath);**

**}**

**}**

**}**

**// Print the max path**

**cout << "Maximum Gold Path: ";**

**for (auto [x, y] : maxPath)**

**cout << "(" << x << "," << y << ") ";**

**cout << endl;**

**return maxGold;**

**}**

**};**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

class Solution {

public:

int m, n;

vector<vector<int>> dp;

// Original DFS logic to compute longest increasing path from (i,j)

int dfs(int i, int j, vector<vector<int>>& matrix) {

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

int maxLen = 0;

if (i > 0 && matrix[i - 1][j] > matrix[i][j])

maxLen = max(maxLen, dfs(i - 1, j, matrix));

if (i < m - 1 && matrix[i + 1][j] > matrix[i][j])

maxLen = max(maxLen, dfs(i + 1, j, matrix));

if (j > 0 && matrix[i][j - 1] > matrix[i][j])

maxLen = max(maxLen, dfs(i, j - 1, matrix));

if (j < n - 1 && matrix[i][j + 1] > matrix[i][j])

maxLen = max(maxLen, dfs(i, j + 1, matrix));

return dp[i][j] = 1 + maxLen;

}

int longestIncreasingPath(vector<vector<int>>& matrix) {

m = matrix.size();

n = matrix[0].size();

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

int maxPath = 0;

int startRow = 0, startCol = 0;

// Step 1: Compute DP and find the starting cell with max path

for (int i = 0; i < m; i++) {

for (int j = 0; j < n; j++) {

int len = dfs(i, j, matrix);

if (len > maxPath) {

maxPath = len;

startRow = i;

startCol = j;

}

}

}

// Step 2: Print the actual path using dp values

cout << "Longest Increasing Path Elements: ";

int i = startRow, j = startCol;

cout << matrix[i][j] << " ";

while (true) {

bool moved = false;

// Check all directions for the next cell in the path

if (i > 0 && matrix[i - 1][j] > matrix[i][j] && dp[i - 1][j] == dp[i][j] - 1) {

i = i - 1;

moved = true;

} else if (i < m - 1 && matrix[i + 1][j] > matrix[i][j] && dp[i + 1][j] == dp[i][j] - 1) {

i = i + 1;

moved = true;

} else if (j > 0 && matrix[i][j - 1] > matrix[i][j] && dp[i][j - 1] == dp[i][j] - 1) {

j = j - 1;

moved = true;

} else if (j < n - 1 && matrix[i][j + 1] > matrix[i][j] && dp[i][j + 1] == dp[i][j] - 1) {

j = j + 1;

moved = true;

}

if (!moved) break;

cout << matrix[i][j] << " ";

}

cout << endl;

return maxPath;

}

};

int main() {

Solution sol;

vector<vector<int>> matrix = {

{9, 9, 4},

{6, 6, 8},

{2, 1, 1}

};

int length = sol.longestIncreasingPath(matrix);

cout << "Length of Longest Increasing Path: " << length << endl;

return 0;

}

**PATTERN 3**

✅ 3. Finding Largest Group / Area / Component (like maxAreaOfIsland) ➡️ "How big is this connected group?" 💡 Return count of connected 1s (area) Base Case: Out of bounds or cell != 1 → return 0 Mark visited: grid[i][j] = 0 Add 1 (self) and explore all 4 directions Return total area

**We should not do area=max(area,dfs()) because we are not calculating maximum. We calculate count**

class Solution {

public:

    int dfs(int r, int c, vector<vector<int>>& grid, int m, int n) {

        if (r < 0 || r >= m || c < 0 || c >= n || grid[r][c] == 0)

            return 0;

        grid[r][c] = 0; // Mark visited

        int area = 0;

        area += dfs(r + 1, c, grid, m, n); // down

        area += dfs(r - 1, c, grid, m, n); // up

        area += dfs(r, c + 1, grid, m, n); // right

        area += dfs(r, c - 1, grid, m, n); // left

        return 1+area;

    }

    int maxAreaOfIsland(vector<vector<int>>& grid) {

        int m = grid.size(), n = grid[0].size();

        int maxi = 0;

        for (int i = 0; i < m; i++) {

            for (int j = 0; j < n; j++) {

                if (grid[i][j] == 1) {

                    int area = dfs(i, j, grid, m, n);

                    maxi = max(maxi, area);

                }

            }

        }

        return maxi;

    }

};

**PATH**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

class Solution {

public:

int m, n;

vector<pair<int, int>> currentPath; // To store current island path

vector<pair<int, int>> maxIslandPath; // To store max area island path

int dfs(int r, int c, vector<vector<int>>& grid) {

if (r < 0 || r >= m || c < 0 || c >= n || grid[r][c] == 0)

return 0;

grid[r][c] = 0; // mark visited

currentPath.push\_back({r, c}); // store current cell in path

int area = 1;

area += dfs(r + 1, c, grid);

area += dfs(r - 1, c, grid);

area += dfs(r, c + 1, grid);

area += dfs(r, c - 1, grid);

return area;

}

int maxAreaOfIsland(vector<vector<int>>& grid) {

m = grid.size(), n = grid[0].size();

int maxi = 0;

for (int i = 0; i < m; i++) {

for (int j = 0; j < n; j++) {

if (grid[i][j] == 1) {

currentPath.clear();

int area = dfs(i, j, grid);

if (area > maxi) {

maxi = area;

maxIslandPath = currentPath;

}

}

}

}

// Print the coordinates of the largest island path

cout << "Path of largest island (coordinates): ";

for (auto& cell : maxIslandPath) {

cout << "(" << cell.first << "," << cell.second << ") ";

}

cout << endl;

return maxi;

}

};

int main() {

vector<vector<int>> grid = {

{0,0,1,0,0},

{0,1,1,0,0},

{0,0,0,0,0},

{1,1,0,1,1},

{1,0,0,1,1}

};

Solution sol;

int maxArea = sol.maxAreaOfIsland(grid);

cout << "Maximum Area: " << maxArea << endl;

return 0;

}

**Area=1 potu return la summa area potukalam (or) area=0 potu return la 1+area potukalam**

**PATTERN 4**

✅ Closed Islands / Region Filling ➡️ "Count the number of completely enclosed areas" 💡 DFS must check all 4 sides, return true/false

Base Case: If out of bounds → return false

If water or visited → return true

Explore all directions, result = AND of all

bool top = dfs(i-1, j); bool down = dfs(i+1, j); ...

return top && down && left && right;

#include <iostream>

#include <vector>

using namespace std;

class Solution {

public:

vector<pair<int,int>> curpath;

vector<pair<int,int>> maxpath;

bool dfs(vector<vector<int>>& g, int i, int j) {

if (i < 0 || j < 0 || i >= g.size() || j >= g[0].size()) {

return false; // touches boundary → not closed

}

if (g[i][j] == 1) {

return true; // already water or visited

}

g[i][j] = 1; // mark visited

curpath.push\_back({i, j});

bool d1 = dfs(g, i + 1, j);

bool d2 = dfs(g, i, j + 1);

bool d3 = dfs(g, i - 1, j);

bool d4 = dfs(g, i, j - 1);

return d1 && d2 && d3 && d4; // all must be enclosed

}

int closedIsland(vector<vector<int>>& grid) {

int res = 0;

for (int i = 0; i < grid.size(); i++) {

for (int j = 0; j < grid[0].size(); j++) {

if (grid[i][j] == 0) {

curpath.clear();

if (dfs(grid, i, j)) {

res++;

if (curpath.size() > maxpath.size()) {

maxpath = curpath;

}

}

}

}

}

cout << "Coordinates of largest closed island:\n";

for (auto e : maxpath) {

cout << "(" << e.first << ", " << e.second << ")\n";

}

return res;

}

};

int main() {

vector<vector<int>> grid = {

{1, 1, 1, 1, 1, 1, 1, 0},

{1, 0, 0, 0, 0, 1, 1, 0},

{1, 0, 1, 0, 1, 1, 1, 0},

{1, 0, 0, 0, 0, 1, 0, 1},

{1, 1, 1, 1, 1, 1, 1, 0}

};

Solution sol;

int result = sol.closedIsland(grid);

cout << "Total closed islands: " << result << endl;

return 0;

}

**PATTERN 5**

**✅ WHEN DO YOU NEED A for LOOP INSIDE DFS?**

**👉 You use a for loop inside DFS when:**

You need to **try multiple choices/options** at the current recursive level.

That is:

* Try every **element** or **index** from current position
* Try **partitions** of a string
* Try **combinations**, **subsets**, or **arrangements** of choices

**✅ Key Rule to Identify:**

🧠 If the problem asks you to:  
“**Return all possible combinations/subsets/ways**”  
→ You almost always need a **for loop inside DFS** to **explore each choice**.

**✅ When Order Matters (Permutations)**

You care about different arrangements of the same elements.

**Example: Permutations of [1, 2]**

* Output: [1,2], [2,1] (both are different)

cpp

CopyEdit

for (int i = 0; i < nums.size(); i++) {

if (!used[i]) {

used[i] = true;

temp.push\_back(nums[i]);

dfs(nums, temp, used, res); // Try everything from 0 again

temp.pop\_back();

used[i] = false;

}

}

✔️ **Use i = 0 to n** every time  
✔️ Need a used[] array to avoid reuse in the same permutation

**✅ When Order Doesn't Matter (Subsets, Combinations)**

You care about **what** is included, not the order.

**Example: Subsets of [1, 2]**

* Output: [1,2] and [2,1] are considered **same** → only keep one

cpp

CopyEdit

for (int i = index; i < nums.size(); i++) {

temp.push\_back(nums[i]);

dfs(i + 1, nums, temp, res); // move forward only

temp.pop\_back();

}

✔️ **Use i = index to n** to prevent reusing earlier numbers  
✔️ Automatically avoids duplicate combinations

**🎯 WHEN TO USE dfs(i + 1)?**

**🔹 Use it in subset / combination / partition problems**

**➡️ You want to go forward from current index, without reusing elements**

**✅ Rule:**

**cpp**

**CopyEdit**

**for (int i = index; i < nums.size(); i++) {**

**temp.push\_back(nums[i]);**

**dfs(i + 1, ...); // move forward**

**temp.pop\_back();**

**}**

**✅ Used in:**

* **Subsets**
* **Combination Sum (without repetition)**
* **Palindrome Partitioning**

**🔹 Why i + 1?**

* **To avoid picking same element again**
* **We’re picking combinations, not permutations**

**🎯 WHEN TO USE dfs(i)?**

**🔹 Use it in combination sum (repeats allowed) or coin change**

**➡️ You can reuse the same element in the next recursive call**

**cpp**

**CopyEdit**

**for (int i = index; i < nums.size(); i++) {**

**if (nums[i] <= target) {**

**temp.push\_back(nums[i]);**

**dfs(i, ...); // same index again**

**temp.pop\_back();**

**}**

**}**

**🎯 WHEN TO USE dfs(l + 1)?**

**🔹 Use it in permutation generation using swapping**

**➡️ l is the position being fixed**

**cpp**

**CopyEdit**

**for (int i = l; i < nums.size(); i++) {**

**swap(nums[i], nums[l]);**

**dfs(l + 1, ...); // move to next fixable position**

**swap(nums[i], nums[l]); // backtrack**

**}**

**✅ Used in:**

* **In-place permutation using swapping**
* **Heap's Algorithm**
* **Sudoku (row/col advancing)**

**✅ Real Examples**

**1. 🔸 Subsets Problem**

cpp

CopyEdit

void dfs(int index, vector<int>& nums, vector<int>& temp, vector<vector<int>>& res) {

res.push\_back(temp); // Every path is a valid subset

for (int i = index; i < nums.size(); i++) {

temp.push\_back(nums[i]); // Choose

dfs(i + 1, nums, temp, res); // Explore

temp.pop\_back(); // Backtrack

}

}

* ✅ **Why loop needed?**  
  → You try every number at and after index.

**2. 🔸 Palindrome Partitioning**

class Solution {

public:

    bool ispalindrome(string s,int l,int r){

        while(l<=r){

            if(s[l]!=s[r]) return false;

            l++;

            r--;

        }

        return true;

    }

void dfs(int start, string s, vector<string>& temp, vector<vector<string>>& res) {

if (start == s.size()) {

res.push\_back(temp);

return;

}

for (int end = start; end < s.size(); end++) {

if (isPalindrome(s, start, end)) {

temp.push\_back(s.substr(start, end - start + 1));

dfs(end + 1, s, temp, res);

temp.pop\_back();

}

}

}

    vector<vector<string>> partition(string s) {

        vector<vector<string>>res;

        vector<string>temp;

        int n=s.length();

        help(res,temp,0,n,s);

        return res;

    }

* ✅ **Why loop needed?**  
  → You try **all possible partitions** of the string starting at index start.

**3. 🔸 Combination Sum**

void dfs(int index, int target, vector<int>& nums, vector<int>& temp, vector<vector<int>>& res) {

if (target == 0) {

res.push\_back(temp);

return;

}

for (int i = index; i < nums.size(); i++) {

if (nums[i] <= target) {

temp.push\_back(nums[i]);

dfs(i, target - nums[i], nums, temp, res); // Not i+1 because reuse allowed

temp.pop\_back();

}

}

}

PERMUTATION 1

class Solution {

public:

    void help(vector<vector<int>>& res, vector<int>& nums, int l) {

        if (l == nums.size()) {  // Base case: If all elements are fixed

            res.push\_back(nums); // Store the permutation

            return;

        }

        for (int i = l; i < nums.size(); i++) {

            swap(nums[l], nums[i]);    // Swap to fix one element at `l`

            help(res, nums, l + 1);    // Recur for the next position

            swap(nums[l], nums[i]);    // Backtrack (undo the swap)

        }

    }

    vector<vector<int>> permute(vector<int>& nums) {

        vector<vector<int>> res;

        help(res, nums, 0); // Start recursion from index 0

        return res;

    }

};

**🔁 2. Permutations (order matters, every element only once)**

**🧠 Key Idea:**

You can pick **any element not already used**.

**✅ Example: Permutations of [1, 2, 3]**

class Solution {

public:

  void help(vector<vector<int>>& res, vector<int>& nums, int l) {

    if (l == nums.size()) {

        res.push\_back(nums);

        return;

    }

    unordered\_set<int> used\_this\_level;  // 👈 to track used elements at this level

    for (int i = l; i < nums.size(); ++i) {

        if (used\_this\_level.count(nums[i])) continue;

        used\_this\_level.insert(nums[i]);

        swap(nums[i], nums[l]);

        help(res, nums, l + 1);

        swap(nums[i], nums[l]);

    }

}

    vector<vector<int>> permuteUnique(vector<int>& nums) {

        vector<vector<int>>res;

        sort(nums.begin(),nums.end());

        help(res,nums,0);

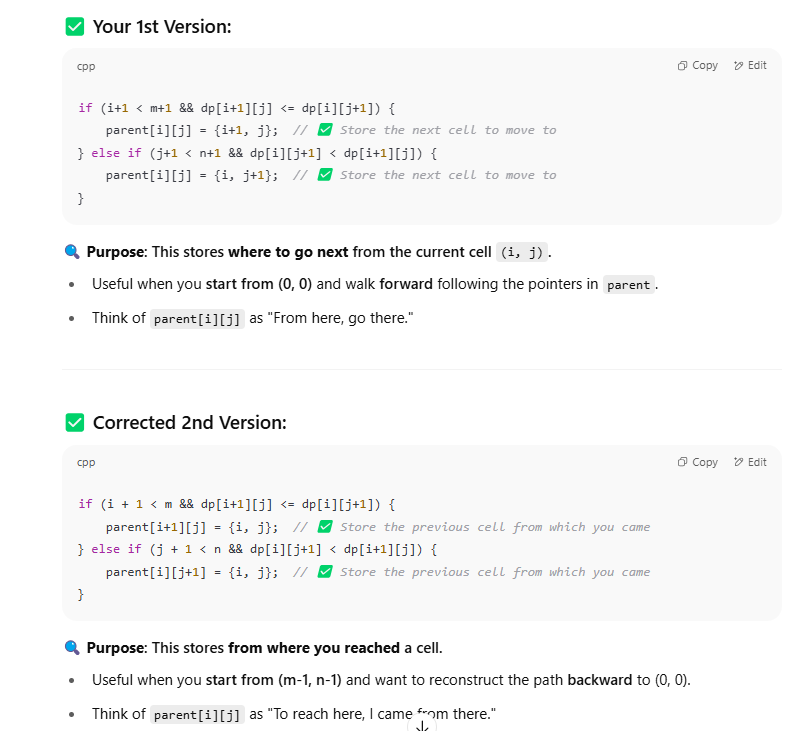
        return res;

    }

};

| **Task** | **Return Type** | **How DFS accumulates** | **Backtracking?** | **Time Complexity** |
| --- | --- | --- | --- | --- |
| ✅ Count total paths | int | total += dfs(...) | ✅ Yes | **O(4ⁿ)** in worst-case (no obstacles), where n = rows × cols |
| ✅ Find max/min path/value | int | max = max(...) | ✅ Yes | **O(m × n)** if memoized, else exponential |
| ✅ Find connected region | int | area++ or area += dfs(...) | ✅ Yes | **O(m × n)** — each cell visited once |
| ✅ Generate combinations | void | push\_back(), pop\_back() | ✅ Yes | **O(2ⁿ × n)** — for subsets, n! for permutations |
| ✅ Partition / Subsets | void | substr + recursion | ✅ Yes | **O(2ⁿ × n)** — for string length n |
| ✅ Region checking (closed islands, etc.) | bool or int | AND/OR of directions | ✅ Yes | **O(m × n)** — each cell visited once |

**DUNGEON GAME PATH**



class Solution {

public:

    int calculateMinimumHP(vector<vector<int>>& dungeon) {

        int m=dungeon.size(),n=dungeon[0].size();

        vector<vector<int>>dp(m+1,vector<int>(n+1,INT\_MAX));

        vector<vector<pair<int,int>>>parent(m,vector<pair<int,int>>(n,{-1,-1}));

        dp[m][n-1]=1;

        dp[m-1][n]=1;

        for(int i=m-1;i>=0;i--){

            for(int j=n-1;j>=0;j--){

                int minpower=min(dp[i+1][j],dp[i][j+1])-(dungeon[i][j]);

                if(minpower<=0)

                dp[i][j]=1;

                else

                dp[i][j]=minpower;

                if(i+1<m && dp[i+1][j]<=dp[i][j+1]){

                    parent[i][j]={i+1,j};

                }

                else if(j+1<n && dp[i][j+1]<dp[i+1][j])

                parent[i][j]={i,j+1};

            }

        }

       vector<pair<int, int>> path;

int i = 0, j = 0;

while (i != -1 && j != -1) {

    path.push\_back({i, j});

    pair<int, int> next = parent[i][j];

    if (next.first == -1 && next.second == -1) break;

    i = next.first;

    j = next.second;

}

        for(auto e:path) cout<<e.first<<" "<<e.second<<endl;

        return dp[0][0];

    }

};

TRAVERSING FROM END OF MATRIX AND PARENT[i][j] MEANS FOR PRINTING GO FROM i=0,j=0

TRAVERSING FROM START OF MATRIX AND PARENT[i][j] MEANS FOR PRINTING GO FROM i=m-1,j=n-1 and reverse

EXAMPLE

// Function to get the minimum cost path

vector<pair<int,int>> getMinPath(vector<vector<int>>& grid) {

int m = grid.size(), n = grid[0].size();

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

vector<vector<pair<int, int>>> parent(m, vector<pair<int, int>>(n, {-1, -1}));

dp[0][0] = grid[0][0];

// Fill DP and parent table

for (int i = 1; i < m; i++) {

dp[i][0] = dp[i - 1][0] + grid[i][0];

parent[i][0] = {i - 1, 0};

}

for (int j = 1; j < n; j++) {

dp[0][j] = dp[0][j - 1] + grid[0][j];

parent[0][j] = {0, j - 1};

}

for (int i = 1; i < m; i++) {

for (int j = 1; j < n; j++) {

if (dp[i - 1][j] < dp[i][j - 1]) {

dp[i][j] = dp[i - 1][j] + grid[i][j];

parent[i][j] = {i - 1, j};

} else {

dp[i][j] = dp[i][j - 1] + grid[i][j];

parent[i][j] = {i, j - 1};

}

}

}

// Reconstruct path from parent table

vector<pair<int, int>> path;

int x = m - 1, y = n - 1;

while (x != -1 && y != -1) {

path.push\_back({x, y});

tie(x, y) = parent[x][y];

}

reverse(path.begin(), path.end()); // from start to end

return path;

}

**✅ 3. Coin Change (Fewest Coins)**

cpp

CopyEdit

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

vector<int> dp(amount + 1, INT\_MAX);

vector<int> parent(amount + 1, -1);

dp[0] = 0;

for (int c : coins) {

for (int j = c; j <= amount; j++) {

if (dp[j - c] != INT\_MAX && dp[j - c] + 1 < dp[j]) {

dp[j] = dp[j - c] + 1;

parent[j] = j - c;

}

}

}

// Reconstruct path

vector<int> path;

if (dp[amount] == INT\_MAX) return {}; // No solution

int curr = amount;

while (curr > 0) {

path.push\_back(curr - parent[curr]);

curr = parent[curr];

}

reverse(path.begin(), path.end());

return path;

}