DFS算法刷题路线

```cpp
#include <iostream>
using namespace std;

// ================= 全局变量定义 =================
int n, m;                // 网格的行数n，列数m（题目通常从1开始计数）
int map[50][50];         // 存储网格状态（0表示可通行，1表示障碍）
bool vis[50][50];        // 访问标记数组，防止重复访问同一位置
int ans = 0;             // 结果计数器，存储从起点到终点的总路径数

// 方向数组：右(0,1)、下(1,0)、左(0,-1)、上(-1,0)
int dx[4] = {0, 1, 0, -1};
int dy[4] = {1, 0, -1, 0};

// ================= DFS深度优先搜索 =================
/**
 * 递归搜索所有从(x,y)到(n,m)的路径
 *
 * @param x 当前位置行坐标（从1开始）
 * @param y 当前位置列坐标（从1开始）
 */
void dfs(int x, int y) {

    // 1. 终点判断：到达右下角(n,m)时，找到一条完整路径
    if (x == n && y == m) {
        ans++;           // 路径计数加1
        return;          // 结束当前路径的搜索
    }

    // 2. 遍历四个方向进行探索
    for (int i = 0; i < 4; i++) {
        int nx = x + dx[i];    // 计算下一个位置的行坐标
        int ny = y + dy[i];    // 计算下一个位置的列坐标

        // 3. 可行性检查：
        //    a) 不越界：nx∈[1,n]，ny∈[1,m]
        //    b) 可通行：map[nx][ny] == 0
        //    c) 未访问：!vis[nx][ny]
        if (nx >= 1 && nx <= n && ny >= 1 && ny <= m &&
            map[nx][ny] == 0 && !vis[nx][ny]) {

            vis[nx][ny] = true;    // 标记为已访问（防止循环）
            dfs(nx, ny);           // 递归深入搜索下一位置
            vis[nx][ny] = false;   // 回溯：恢复未访问状态（允许其他路径探索）
        }
    }
}

// ================= 主程序入口 =================
int main() {
    // 输入网格规格
    cin >> n >> m;

    // 输入网格数据（假设题目中0表示空地，1表示障碍）
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            cin >> map[i][j];
        }
    }
```

```
    // 初始化起点(1,1)为已访问状态
    vis[1][1] = true;

    // 开始DFS搜索
    dfs(1, 1);

    // 输出从起点到终点的总路径数
    cout << ans << endl;

    return 0;
}
```

递归思想

```
#include <iostream>      // 包含输入输出流头文件，用于使用cout和endl
using namespace std;     // 使用标准命名空间，避免每次都要写std::

// 递归函数：倒计时
// 参数n：倒计时的起始数字
void countDown(int n) {
    // 递归终止条件：当n等于0时
    if (n == 0) {
        cout << "发射！" << endl;  // 输出发射信息，endl表示换行
        return;                    // 结束递归调用，返回上一层
    }
    // 输出当前倒计时数字
    cout << n << "..." << endl;    // 显示当前数字并换行
    // 递归调用：减少n的值，进入下一层倒计时
    countDown(n - 1);              // n-1作为新参数调用自身
}
// 主函数：程序入口点
int main() {
    // 调用倒计时函数，从3开始倒计时
    countDown(3);    // 函数调用：将控制权交给countDown函数
    // 返回0表示程序正常结束
    return 0;        // 返回给操作系统，0通常表示成功
}
```

力扣

二叉树的前序遍历

题目描述：

给你二叉树的根节点 root ，返回它节点值的 前序 遍历。

示例 1：

（根节点1，右孩子是2，2的左孩子是3）

- 输入： root = [1,null,2,3]
- 输出： [1,2,3]

示例 2：

- 输入： root = [1,2,3,4,5,null,8,null,null,6,7,9]
- 输出： [1,2,4,5,6,7,3,8,9]

示例 3：

- 输入： root = []
- 输出： []

示例 4：

- 输入： root = [1]
- 输出： [1]

提示：

- 树中节点数目在范围 [0, 100] 内
- -100 <= Node.val <= 100

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    void dfs(TreeNode* node,vector<int>& result){
        if(node == nullptr){return;}

        result.push_back(node->val);
        dfs(node->left,result);
        dfs(node->right,result);
    }
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> result;
        dfs(root,result);
        return result;
    }
};
```

> 前序: 根 -> 左 -> 右

二叉树的中序遍历

题目描述：

给定一个二叉树的根节点 root ，返回 它的 中序 遍历。

示例 1：

（根节点1，右孩子是2，2的左孩子是3）

- 输入： root = [1,null,2,3]
- 输出： [1,3,2]

示例 2：

- 输入： root = []

- 输出： []

示例 3：

- 输入： root = [1]
- 输出： [1]

提示：

- 树中节点数目在范围 [0, 100] 内
- -100 <= Node.val <= 100

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    // 深度优先搜索（DFS）函数，用于中序遍历二叉树
    // 中序遍历顺序：左子树 -> 根节点 -> 右子树
    void dfs(TreeNode* node, vector<int>& result){
        // 递归终止条件：当前节点为空
        if(node == nullptr){
            return;  // 直接返回，不进行任何操作
        }

        // 递归遍历左子树
        dfs(node->left, result);

        // 访问当前节点（根节点），将节点值加入结果数组
        // 中序遍历的核心：在左右子树递归之间处理当前节点
        result.push_back(node->val);

        // 递归遍历右子树
        dfs(node->right, result);
    }

    // 主函数，二叉树的中序遍历
    vector<int> inorderTraversal(TreeNode* root) {
        // 创建存储遍历结果的数组
        vector<int> result;

        // 从根节点开始深度优先搜索
        dfs(root, result);

        // 返回遍历结果
        return result;
    }
};
```

> 中序: 左 -> 根 -> 右

> 对于任何一个节点，必须先去它的左子树里转一圈，把左边的事情做完了，才回来记录这个节点的值，最后再去右子树转一圈。

二叉树的后序遍历

题目描述：

给你一棵二叉树的根节点 root ，返回其节点值的 后序遍历 。

示例 1：

（根节点1，右孩子是2，2的左孩子是3）

- 输入： root = [1,null,2,3]
- 输出： [3,2,1]

示例 2：

- 输入： root = [1,2,3,4,5,null,8,null,null,6,7,9]
- 输出： [4,6,7,5,2,9,8,3,1]

示例 3：

- 输入： root = []
- 输出： []

示例 4：

- 输入： root = [1]
- 输出： [1]

提示：

- 树中节点数目在范围 [0, 100] 内
- -100 <= Node.val <= 100

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    void dfs(TreeNode* node,vector<int>& result){
        if(node == nullptr){return;}
        dfs(node->left,result);
        dfs(node->right,result);
        result.push_back(node->val);
    }
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int> result;
        dfs(root,result);
        return result;
    }
};
```

> 后序：左 -> 右 -> 根

语法重点：

- `vector<int>`：创建一个整数动态数组；
- `push_back()`：动态数组的一个功能函数，往数组后面塞一个数据。

二叉树的最大深度

题目描述：

给定一个二叉树 root ，返回其最大深度。

二叉树的 最大深度 是指从根节点到最远叶子节点的最长路径上的节点数。

示例 1：

- 输入： root = [3,9,20,null,null,15,7]
- 输出： 3

示例 2：

- 输入： root = [1,null,2]
- 输出： 2

提示：

- 树中节点的数量在 [0, 10^4] 区间内。
- -100 <= Node.val <= 100

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if(root == nullptr){return 0;}

        int leftNum = maxDepth(root->left);
        int rightNum = maxDepth(root->right);

        return max(leftNum,rightNum)+1;
    }
};
```

路径总和

题目描述：

给你二叉树的根节点 root 和一个表示目标和的整数 targetSum 。判断该树中是否存在 根节点到叶子节点 的路径，这条路径上所有节点值相加等于目

标和 targetSum 。如果存在，返回 true ；否则，返回 false 。

叶子节点 是指没有子节点的节点。

示例 1：

- 输入： root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22
- 输出： true

示例 2：

- 输入： root = [1,2,3], targetSum = 5
- 输出： false

示例 3：

- 输入： root = [], targetSum = 0
- 输出： false

提示：

- 树中节点的数目在范围 [0, 5000] 内
- -1000 <= Node.val <= 1000
- -1000 <= targetSum <= 1000

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    bool hasPathSum(TreeNode* root, int targetSum) {
        if(root == nullptr){return false;}

        if(root->left == nullptr && root->right == nullptr){
            if(root->val == targetSum){
                return true;
            }else{return false;}
        }

        return hasPathSum(root->left,targetSum - root->val) || hasPathSum(root->right,targetSum-root->val);
    }
};
```

减法递归

图像渲染

题目描述：

有一幅以 `m x n` 的二维整数数组表示的图画 `image` ，其中 `image[i][j]` 表示该图画的像素值大小。你也被给予三个整数 `sr` , `sc` 和 `color` 。你应

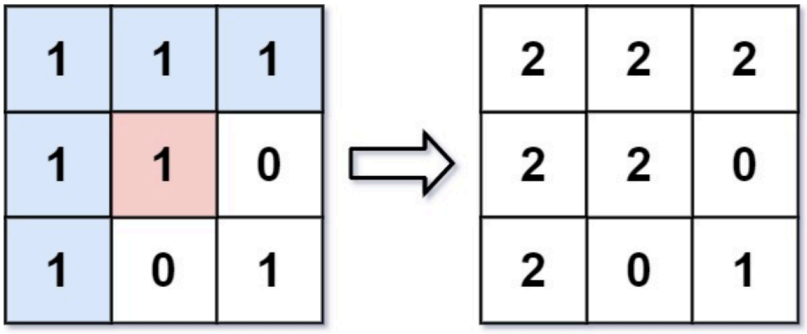
该从像素 `image[sr][sc]` 开始对图像进行上色**填充**。

为了完成**上色工作**：

1. 从初始像素开始，将其颜色改为`color`。
2. 对初始坐标的**上下左右四个方向上**相邻且与初始像素的原始颜色同色的像素点执行相同操作。
3. 通过检查与初始像素的原始颜色相同的相邻像素并修改其颜色来继续**重复**此过程。
4. 当**没有**其它原始颜色的相邻像素时**停止**操作。

最后返回经过上色渲染**修改**后的图像。

示例 1：



- 输入： image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, color = 2
- 输出： [[2,2,2],[2,2,0],[2,0,1]]

> 解释：在图像的正中间，坐标 (sr,sc)=(1,1) （即红色像素）,在路径上所有符合条件的像素点的颜色都被更改成相同的新颜色（即蓝色像素）。
> 注意，右下角的像素 没有 更改为2，因为它不是在上下左右四个方向上与初始点相连的像素点。

```
[1 1 1]  下标0
[1 1 0]  下标1
[1 0 1]  下标2


sr=1 sc=1 -> image[1][1];就是中间的1，对他进行填色，变成2；上和左相同，变成一样；而下和右为0，不同，就不填；刚刚的上（[0][1]）左右也是1，填色，直到不同。
```

示例 2：

- 输入： image = [[0,0,0],[0,0,0]], sr = 0, sc = 0, color = 0
- 输出： [[0,0,0],[0,0,0]]

提示：

- m == image.length
- n == image[i].length
- 1 <= m, n <= 50
- 0 <= image[i][j], color < 2^16
- 0 <= sr < m
- 0 <= sc < n

```
class Solution {
public:
    int dx[4] = {0, 1, 0, -1};
    int dy[4] = {1, 0, -1, 0};
```

```
    void dfs(vector<vector<int>>& image, int x, int y, int oldColor, int newColor) {
        image[x][y] = newColor;

        for(int i = 0; i < 4; i++) {
            int nx = x + dx[i];
            int ny = y + dy[i];

            if(nx >= 0 && nx < image.size() && ny >= 0 && ny < image[0].size()) {
                if(image[nx][ny] == oldColor) {
                    dfs(image, nx, ny, oldColor, newColor);
                }
            }
        }
    }

    vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int color) {
        int oldColor = image[sr][sc];

        if(oldColor == color) {
            return image;
        }

        dfs(image, sr, sc, oldColor, color);
        return image;
    }
};
```

> 油漆桶原理

> 二维数组

岛屿数量

给你一个由 `'1'` （陆地）和 `'0'` （水）组成的的二维网格，请你计算网格中岛屿的数量。

岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。

此外，你可以假设该网格的四条边均被水包围。

示例 1：

- 输入：

```
grid = [
  ["1","1","1","1","0"],
  ["1","1","0","1","0"],
  ["1","1","0","0","0"],
  ["0","0","0","0","0"]
]
```

- 输出： `1`
- 解释： 所有的 '1' 都在一起，连成了一整块大陆。

示例 2：

- 输入：

```
grid = [
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]
```

- 输出：`3`
- 解释：
  - 左上角有一块（4个1连着）。
  - 中间有一块（单独一个1）。
  - 右下角有一块（2个1连着）。
  - 总共 3 座岛。

提示：

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 300`
- `grid[i][j]` 的值为 `'0'` 或 `'1'`

```cpp
class Solution {
public:
    int dx[4] = {0,0,1,-1};
    int dy[4] = {1,-1,0,0};

    void dfs(vector<vector<char>>& grid, int x, int y){
        grid[x][y] = '0';

        for(int i=0; i<4; i++){
            int nx = x + dx[i];
            int ny = y + dy[i];


            if(nx >= 0 && nx < grid.size() && ny >= 0 && ny < grid[0].size() && grid[nx][ny] == '1'){
                dfs(grid, nx, ny);
            }
        }
    }

    int numIslands(vector<vector<char>>& grid) {
        int sum = 0;
        int n = grid.size();
        int m = grid[0].size();

        for(int i=0; i<n; i++){
            for(int j=0; j<m; j++){
                if(grid[i][j] == '1'){
                    sum++;
                    dfs(grid, i, j);
                }
            }
        }

        return sum;
    }
};
```

> 沉岛思想

> 把 '1' 变成 '0'

岛屿的最大面积

题目描述：

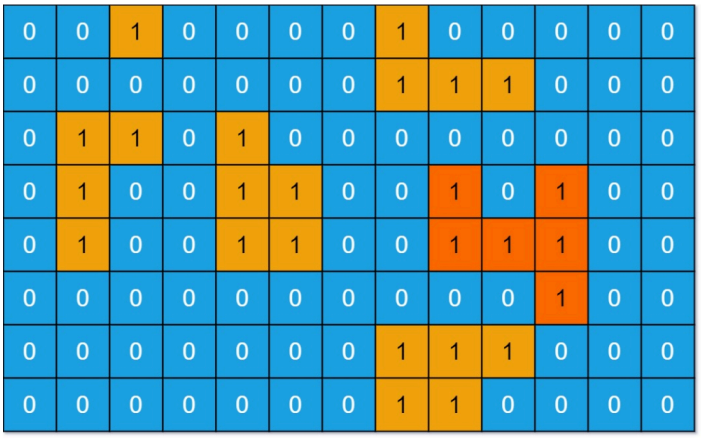
给你一个大小为 `m x n` 的二进制矩阵 `grid` 。

岛屿是由一些相邻的 `1` (代表土地) 构成的组合，这里的「相邻」要求两个 `1` 必须在**水平或者竖直的四个方向上**相邻。你可以假设 `grid` 的四个边缘都被 `0`（代表水）包围着。

岛屿的面积是岛上值为 `1` 的单元格的数目。

计算并返回 `grid` 中最大的岛屿面积。如果没有岛屿，则返回面积为 `0` 。

示例 1：



- 输入：

```
grid = [[0,0,1,0,0,0,0,1,0,0,0,0,0],
[0,0,0,0,0,0,0,1,1,1,0,0,0],[0,1,1,0,1,0,0,0,0,0,0,0,0],[0,1,0,0,1,1,0,0,1,0,1,0,0],[0,1,0,0,1,1,0,0,1,1,1,0,0],
[0,0,0,0,0,0,0,0,0,0,1,0,0],[0,0,0,0,0,0,0,1,1,1,0,0,0],[0,0,0,0,0,0,0,1,1,0,0,0,0]]
```

- 输出： `6`
- 解释： 答案不应该是 `11` ，因为岛屿只能包含水平或垂直这四个方向上的 `1` 。（斜着的不算）

示例 2：

- 输入： `grid = [[0,0,0,0,0,0,0,0]]`
- 输出： `0`

提示：

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 50`
- `grid[i][j]` 为 `0` 或 `1`

```
class Solution {
public:
```

```
    int dx[4] = {0,0,1,-1};
    int dy[4] = {1,-1,0,0};

    int dfs(vector<vector<int>>& grid,int x,int y){
        grid[x][y] = 0;

        int area = 1;

        for(int i=0;i<4;i++){
            int nx = x+dx[i];
            int ny = y+dy[i];

            if(nx>=0 && nx<grid.size() && ny>=0 && ny<grid[0].size() && grid[nx][ny] == 1){
                area += dfs(grid, nx, ny);
            }
        }

        return area;
    }
    int maxAreaOfIsland(vector<vector<int>>& grid) {
        int n = grid.size();
        int m = grid[0].size();
        int maxAns = 0;

        for(int i=0;i<n;i++){
            for(int j=0;j<m;j++){
                if(grid[i][j] == 1){
                    int currentIslandArea = dfs(grid, i, j);//取最大值
                    maxAns = max(maxAns, currentIslandArea);//比较
                }
            }
        }

        return maxAns;
    }
};
```