BFS算法刷题路线

力扣

```cpp
#include <iostream>
#include <queue>  // 使用STL队列实现BFS
using namespace std;

// ================= 数据结构定义 =================
/**
 * 节点结构体：表示网格中的一个位置
 * @param x     行坐标（从1开始）
 * @param y     列坐标（从1开始）
 * @param step 从起点到当前节点的步数
 */
struct Node {
    int x, y;
    int step;
};

// ================= 全局变量定义 =================
int n, m;                // 网格的行数n，列数m
int map[50][50];         // 网格地图（0可通行，1障碍物）
bool vis[50][50];        // 访问标记数组，记录是否已访问

// 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};

// ================= BFS广度优先搜索 =================
/**
 * 使用BFS搜索从起点到终点的最短路径
 *
 * @param start_x 起点行坐标
 * @param start_y 起点列坐标
 * @param end_x   终点行坐标
 * @param end_y   终点列坐标
 * @return 最短路径步数（无法到达时返回-1）
 */
int bfs(int start_x, int start_y, int end_x, int end_y) {
    // 1. 初始化队列
    queue<Node> q;

    // 2. 创建起点节点并入队
    Node start = {start_x, start_y, 0};
    q.push(start);
    vis[start_x][start_y] = true;  // 标记起点已访问

    // 3. 队列非空时循环搜索
    while (!q.empty()) {
        // 3.1 取出队首节点（当前探索的节点）
        Node now = q.front();
        q.pop();

        // 3.2 终点判断：到达终点时返回步数
        if (now.x == end_x && now.y == end_y) {
            return now.step;
        }
```

```
        // 3.3 遍历四个方向
        for (int i = 0; i < 4; i++) {
            int nx = now.x + dx[i];  // 计算下一位置行坐标
            int ny = now.y + dy[i];  // 计算下一位置列坐标

            // 3.4 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（防止重复入队）

                // 创建下一节点并入队（步数+1）
                Node next = {nx, ny, now.step + 1};
                q.push(next);
            }
        }
    }

    // 4. 队列空仍未找到终点，说明不可达
    return -1;
}

// ================= 主程序入口 =================
int main() {
    // 1. 输入网格规格
    cin >> n >> m;

    // 2. 输入网格数据（这里假设map已在全局初始化）
    // 实际使用时需要先读入map数据，例如：
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            cin >> map[i][j];
        }
    }

    // 3. 调用BFS计算从(1,1)到(n,m)的最短步数
    cout << bfs(1, 1, n, m) << endl;

    return 0;
}
```

```
#include <iostream>       // 引入输入输出流库，用于标准输入输出
#include <queue>          // 引入队列容器，用于BFS中的节点存储
#include <unordered_set>  // 引入无序集合容器，用于记录已访问节点

using namespace std;      // 使用标准命名空间，简化代码

// 定义广度优先搜索（BFS）函数，参数为起始节点指针
void bfs(Node* start) {
    queue<Node*> q;               // 创建队列q，用于存储待访问的节点
    q.push(start);                // 将起始节点加入队列
    unordered_set<Node*> visited; // 创建无序集合visited，用于存储已访问节点
    visited.insert(start);        // 将起始节点标记为已访问

    // 主循环：当队列不为空时，继续搜索
    while (!q.empty()) {
        int size = q.size();      // 获取当前层的节点数量

        // 遍历当前层的所有节点
        for (int i = 0; i < size; i++) {
```

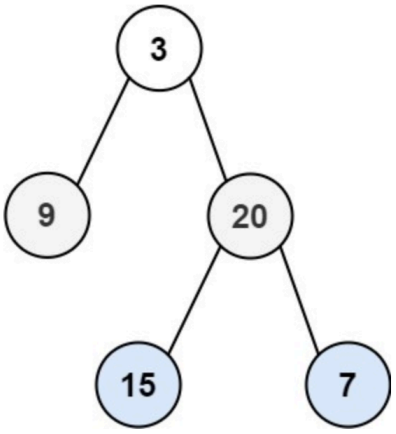
```
            Node* cur = q.front(); // 从队列中取出队首节点作为当前节点
            q.pop();               // 将队首节点弹出队列

            // 检查当前节点是否是目标节点，若是则返回步数（此处为伪代码，需根据实际实现调整）
            if (cur is target) return step;

            // 遍历当前节点的所有邻居节点
            for (Node* neighbor : cur->neighbors) {
                // 如果邻居节点未被访问过
                if (visited.find(neighbor) == visited.end()) {
                    q.push(neighbor);         // 将邻居节点加入队列
                    visited.insert(neighbor); // 标记邻居节点为已访问
                }
            }
        }
    }
}
```

二叉树的层序遍历

题目描述：

给你二叉树的根节点 `root` ，返回其节点值的 **层序遍历** 。 （即逐层地，从左到右访问所有节点）。

示例 1：



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

示例 2：

- 输入： `root = [1]`
- 输出： `[[1]]`

示例 3：

- 输入： `root = []`
- 输出： `[]`

提示：

- 树中节点数目在范围 `[0, 2000]` 内
- `-1000 <= Node.val <= 1000`

```
/**
```

```
 * 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:
    vector<vector<int>> levelOrder(TreeNode* root) {
      vector<vector<int>> res;//定义动态二维数组
      //空树
      if(root == nullptr){
        return res;
      }
      //定义队列-为q
      queue<TreeNode*> q;
      q.push(root);//讲根节点推入队列
      //队列不为空循环
      while(!q.empty()){
        int n = q.size();
        vector<int> res_n;//定义二维数组里面的一维数组

        for(int i=0;i<n;i++){
            TreeNode* node = q.front();// 取出队头
            q.pop();                   // 弹出队头

            res_n.push_back(node->val);// 记录数值

           //将下一层的左右孩子加入队列
            if(node->left != nullptr) q.push(node->left);
            if(node->right != nullptr) q.push(node->right);
        }

        res.push_back(res_n);
      }
      return res;
    }
};
```

二维数组

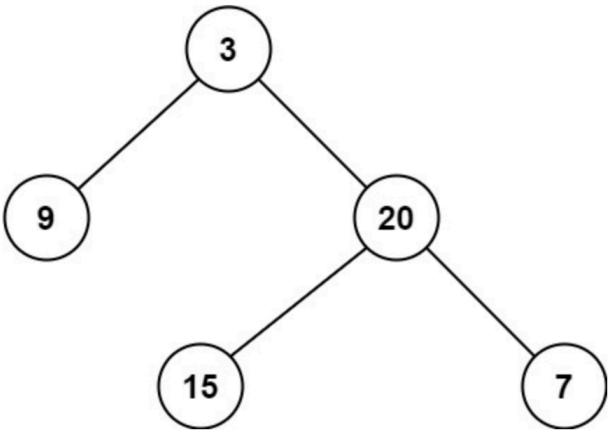
二叉树的最小深度

给定一个二叉树，找出其最小深度。

最小深度是从根节点到最近叶子节点的最短路径上的节点数量。

说明： 叶子节点是指没有子节点的节点。

示例 1：



- 输入： `root = [3,9,20,null,null,15,7]`
- 输出： `2`
- 解释： 最小深度是从根节点 3 到叶子节点 9 的路径，节点数为 2。

示例 2：

- 输入： `root = [2,null,3,null,4,null,5,null,6]`
- 输出： `5`

提示：

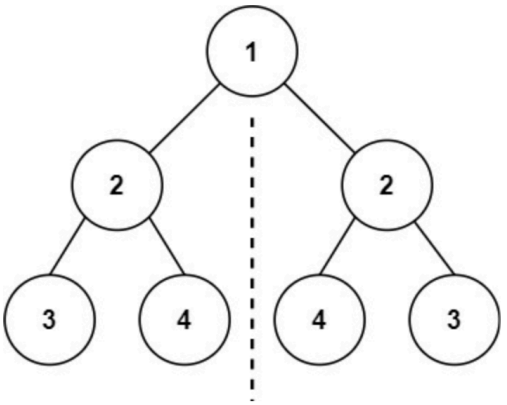
- 树中节点数的范围在 `[0, 10^5]` 内
- `-1000 <= Node.val <= 1000`

```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:
    int minDepth(TreeNode* root) {
        if(root == nullptr){
            return 0;
        }

        queue<TreeNode*> q;
        q.push(root);
        int depth = 1;

        while(!q.empty()){
            int n = q.size();
            for(int i =0;i<n;i++){
                TreeNode* node = q.front();
                q.pop();
                if(node->left == nullptr && node->right == nullptr){
                    return depth;
                }

                if(node->left != nullptr) q.push(node->left);
```

```
                if(node->right != nullptr) q.push(node->right);
            }
            depth++;
        }

        return depth;
    }
};
```

对称二叉树

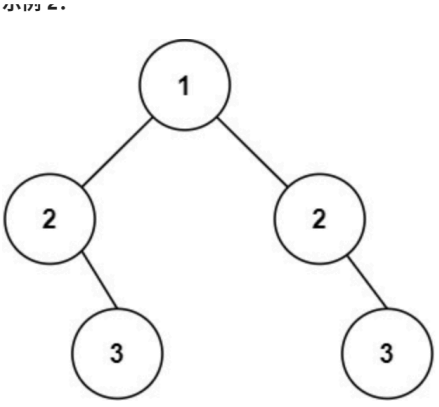
给你一个二叉树的根节点 `root` ， 检查它是否轴对称。



示例 1：

- 输入： `root = [1,2,2,3,4,4,3]`
- 输出： `true`

示例 2：



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

提示：

- 树中节点数目在范围 `[1, 1000]` 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:
    bool isSymmetric(TreeNode* root) {
        if(root == nullptr){return true;}

        queue<TreeNode*> q;
        q.push(root->left);
        q.push(root->right);

        while(!q.empty()){
            TreeNode* leftNode = q.front();
            q.pop();
            TreeNode* rightNode = q.front();
            q.pop();

            if(leftNode == nullptr && rightNode == nullptr){continue;}
            if(leftNode == nullptr || rightNode == nullptr || (leftNode->val != rightNode->val)){
                return false;
            }

            q.push(leftNode->left);
            q.push(rightNode->right);

            q.push(leftNode->right);
            q.push(rightNode->left);
        }
        return true;
    }
};
```

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'`

```
class Solution {
public:
    void bfs(vector<vector<char>>& grid, int x, int y) {
        queue<pair<int, int>> q;
        q.push({x, y});
        grid[x][y] = '0';

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

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

        while (!q.empty()) {
            auto [n_x, n_y] = q.front();
            q.pop();

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

                if (nx >= 0 && nx < n && ny >= 0 && ny < m && grid[nx][ny] == '1') {
                    q.push({nx, ny});
                    grid[nx][ny] = '0';
                }
            }
        }
    }

    int numIslands(vector<vector<char>>& grid) {
        if (grid.empty()) return 0;

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

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

pair<int, int>：容器

腐烂的橘子

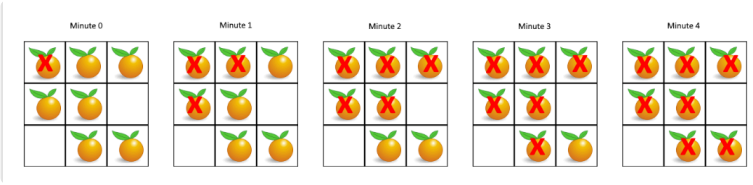
在给定的 `m x n` 网格 `grid` 中，每个单元格可以有以下三个值之一：

- 值 `0` 代表空单元格；
- 值 `1` 代表新鲜橘子；
- 值 `2` 代表腐烂的橘子。

每分钟，腐烂的橘子 周围 4 个方向上相邻 的新鲜橘子都会腐烂。

返回直到单元格中没有新鲜橘子为止所必须经过的最小分钟数。如果不可能，返回 `-1` 。

示例 1：



- 输入： `grid = [[2,1,1],[1,1,0],[0,1,1]]`
- 输出： `4`

示例 2：

- 输入： `grid = [[2,1,1],[0,1,1],[1,0,1]]`
- 输出： `-1`
- 解释： 左下角的橘子（第 2 行，第 0 列）永远不会腐烂，因为腐烂只会发生在 4 个方向上。

示例 3：

- 输入： `grid = [[0,2]]`
- 输出： `0`
- 解释： 因为 0 分钟时已经没有新鲜橘子了，所以答案就是 0 。

提示：

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 10`
- `grid[i][j]` 仅为 `0` 、 `1` 或 `2`

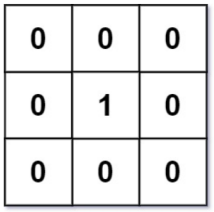
```cpp
class Solution {
public:
    int orangesRotting(vector<vector<int>>& grid) {
        int n = grid.size();
        int m = grid[0].size();

        queue<pair<int,int>> q;
        int n_X = 0;

        for(int i = 0;i<n;i++){
            for(int j =0;j<m;j++){
                if(grid[i][j] == 2){
                    q.push({i,j});
                }else if(grid[i][j] == 1){
                    n_X++;
                }
            }
        }

        if(n_X == 0){return 0;}

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

        while(!q.empty() && n_X>0){
            minutes++;
            int size = q.size();

            for(int i=0;i<size;i++){
                auto [n_x,n_y] = q.front();
                q.pop();

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

                    if(nx>=0 && nx<n && ny>=0 && ny<m && grid[nx][ny] == 1){
                        grid[nx][ny] = 2;
                        q.push({nx,ny});
                        n_X--;
                    }
                }
            }
        }

        if(n_X >0){
            return -1;
        }else{return minutes;}
    }
};
```

01 矩阵

给定一个由 `0` 和 `1` 组成的矩阵 `mat` ，请输出一个大小相同的矩阵，其中每一个格子是 `mat` 中对应位置元素到最近的 `0` 的距离。

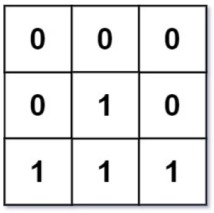
两个相邻元素间的距离为 `1` 。

示例 1：



- 输入： `mat = [[0,0,0],[0,1,0],[0,0,0]]`
- 输出： `[[0,0,0],[0,1,0],[0,0,0]]`

示例 2：



- 输入： `mat = [[0,0,0],[0,1,0],[1,1,1]]`
- 输出： `[[0,0,0],[0,1,0],[1,2,1]]`

提示：

- `m == mat.length`
- `n == mat[i].length`
- `1 <= m, n <= 10^4`
- `1 <= m * n <= 10^4`
- `mat[i][j]` 恰好是 `0` 或 `1`
- `mat` 中至少有一个 `0`

```cpp
class Solution {
public:
    vector<vector<int>> updateMatrix(vector<vector<int>>& mat) {
        int n = mat.size();
        int m = mat[0].size();

        queue<pair<int,int>> q;
        vector<vector<int>> dist(n, vector<int>(m, -1));//申请二维数组，初始值为-1，n行m列。

        for(int i=0;i<n;i++){
            for(int j=0;j<m;j++){
                if(mat[i][j] == 0){
                    dist[i][j] = 0;
                    q.push({i,j});
                }
            }
        }

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

        while(!q.empty()){
            auto [n_x,n_y] = q.front();
            q.pop();

            for(int i =0;i<4;i++){
                int nx = n_x+dx[i];
                int ny = n_y+dy[i];
```

```
                if(nx>=0 && nx<n && ny>=0 && ny<m && dist[nx][ny] == -1){
                    dist[nx][ny] = dist[n_x][n_y] + 1;
                    q.push({nx,ny});
                }
            }
        }
        return dist;
    }
};
```