

A. 回溯问题

一、子集问题

1. 子集-78
2. 子集II-90

二、组合问题

1. 组合-77
2. 组合总和-39
3. 组合总和II-40
4. 组合总和III-216

三、排列问题

1. 全排列-46
2. 全排列II-47

四、分割问题

1. 分割回文串-131

五、二叉树问题

1. 路径总和-112
2. 路径总和II-113

六、总结

B. 广度优先算法-BFS

一、岛屿问题

1. **被围绕的区域-130**
2. 统计封闭岛屿的数目-1254
3. 岛屿数量-200
3. 不同的岛屿数量-694
4. 岛屿的最大面积-695
5. 统计子岛屿-1905
6. 飞地的数量-1020

二、二叉树

1. 路径总和-III-43

A. 回溯问题

参考链接: <https://programmercarl.com/>

- 回溯问题一般可以分为以下几种问题
 - **子集问题** : 一个N个数的集合里有多少条符合条件的子集
 - **组合问题** : N个数里面按照一定规律找出k个数的集合
 - **排列问题** : N个数按照一定规则全排列, 有几种排列方式
 - **切割问题** : 一个字符串按一定规则切割有几种切割方式
 - **棋盘问题** : N皇后, 解数独等等
- 组合无序, 排列有序
- 子集问题: 在回溯函数中先将路径添加到结果中, `res.push_back(path)`
- 排列问题: 不需要 `startIndex`

一、子集问题

1. 子集-78

- 给定整数数组 `nums`
- 数组内元素互不相同
- 返回数组的所有子集

- 注意子集问题的回溯第一步是将路径添加到结果数组 `res.push_back(path)`
- 由于不能重复，因此回溯的索引是 `i+1`
- 结束条件: `startIndex>nums.size()`
- 初始化的时候 `startIndex=0`

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;

    void backtrack(vector<int>& nums, int startIndex){
        // 子集问题首先应该将路线添加到结果中
        res.push_back(path);
        if(startIndex==nums.size()) return;
        for(int i = startIndex; i<nums.size();i++){
            path.push_back(nums[i]);
            backtrack(nums,i+1);
            path.pop_back();
        }
    }

    vector<vector<int>> subsets(vector<int>& nums) {
        backtrack(nums, 0);
        return res;
    }
};
```

2. 子集II-90

- 给定整数数组 `nums`
- 数组内存在重复元素
- 返回数组的不重复子集

- 注意子集问题的回溯第一步是将路径添加到结果数组 `res.push_back(path)`
- 由于不能重复，因此回溯的索引是 `i+1`
- 结束条件: `startIndex>nums.size()`
- 利用 `used` 数组实现去重，具体来说分为两步
 - `if(i>0&&nums[i]==nums[i-1]&&used[i-1]==true) continue`，来跳过同层重复的可能
 - 回溯前 `used[i]=true`，回溯后 `used[i]=false`

- 初始化的时候 `startIndex=0`
- 由于是去重复，因此还需要进行排序 `sort`

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;

    void backtrack(vector<int>& nums, int startIndex, vector<bool>& used){
        res.push_back(path);
        for(int i = startIndex; i<nums.size(); i++){
            // 跳过同一树层中使用过的元素
            if(i>0&&nums[i]==nums[i-1]&&used[i-1]==false) continue;
            path.push_back(nums[i]);
            used[i] = true;
            backtrack(nums, i+1, used);
            used[i] = false;
            path.pop_back();
        }
    }

    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        vector<bool> used(nums.size(), false);
        backtrack(nums, 0, used);
        return res;
    }
};
```

二、组合问题

1. 组合-77

- 给定两个整数 `n` 和 `k`
- 返回范围 `[1,n]` 内所有可能的 `k` 个数的组合

- 注意组合的问题是当满足了返回的条件时候，才将路径存入结果数组
- 由于不能重复，因此回溯的索引是 `i+1`
- 结束条件: `path.size()==k`
- 初始化的时候 `startIndex=1`，数字的范围是 `[1,n]`，因此起始值是 `1`

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;

    void backtrack(int startIndex, int n, int k){
        if(path.size()==k) {
            res.push_back(path);
            return;
        }
    }
};
```

```

        for(int i=startIndex; i<=n; i++){
            path.push_back(i);
            backtrack(i+1, n, k);
            path.pop_back();
        }

    }

    vector<vector<int>> combine(int n, int k) {
        backtrack(1, n, k);
        return res;
    }

};

```

2. 组合总和-39

- 给定 **无重复** 元素的整数数组 **candidates**
- 目标整数 **target**
- 返回和为 **target** 的所有不同组合
- 注意：同一个数字可以被无限制地重复选取

- 注意组合的问题是当满足了返回的条件时候，才将路径存入结果数组
- 由于数字可以重复，因此回溯的索引是 **i**
- 结束条件：**sum>target** 和 **sum==target**
- 初始化的时候 **startIndex=0**

```

class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;

    void backtrack(vector<int>& candidates, int target, int sum, int startIndex)
    {
        if(sum>target){
            return;
        }
        if(sum==target){
            res.push_back(path);
            return;
        }
        for(int i = startIndex; i<candidates.size(); i++){
            sum+=candidates[i];
            path.push_back(candidates[i]);
            backtrack(candidates, target, sum, i);
            sum-=candidates[i];
            path.pop_back();
        }
    }

    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        backtrack(candidates, target, 0, 0);
        return res;
    }

};

```

3. 组合总和II-40

- 给定 有重复 元素的整数数组 `candidates`
- 目标整数 `target`
- 返回和为 `target` 的所有不同组合
- 注意：同一个数字在每个组合里只能使用一次

- 注意组合的问题是当满足了返回的条件时候，才将路径存入结果数组
- 由于数字不能重复，因此回溯的索引是 `i+1`
- 结束条件： `sum>target` 和 `sum==target`
- 初始化的时候 `startIndex=0`
- 利用 `used` 数组实现去重，具体来说分为两步
 - `if(i>0&&nums[i]==nums[i-1]&&used[i-1]==false) continue`，来跳过同层重复的可能
 - 回溯前 `used[i]=true`，回溯后 `used[i]=false`
- 由于是去重复，因此还需要进行排序 `sort`

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;

    void backtrack(vector<int>& candidates, int target, int sum, int startIndex,
vector<bool>& used){
        if(sum>target) return;
        if(sum==target){
            res.push_back(path);
            return;
        }
        for(int i = startIndex; i < candidates.size(); i++) {
            if(i>0&&candidates[i]==candidates[i-1]&&used[i-1]==false){
                continue;
            }
            used[i] = true;
            sum+=candidates[i];
            path.push_back(candidates[i]);
            backtrack(candidates, target, sum, i+1, used);
            path.pop_back();
            sum-=candidates[i];
            used[i] = false;
        }
    }

    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        sort(candidates.begin(), candidates.end());
        vector<bool> used(candidates.size(), false);
        backtrack(candidates, target, 0, 0, used);
        return res;
    }
};
```

4. 组合总和III-216

- 给定两个整数 n 和 k
- 返回相加和为 n 的 k 个数的组合
- 只使用数字 $1\sim 9$
- 每个数字最多使用一次

- 注意组合的问题是当满足了返回的条件时候，才将路径存入结果数组
- 由于不能重复，因此回溯的索引是 $i+1$
- 结束条件: $path.size()==k$ 和 $sum>n$
- 初始化的时候 $startIndex=1$ ，数字的范围是 $[1,9]$ ，因此起始值是 1

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
    // 和为n，数量为k，由于数字的范围是1到9，因此startIndex肯定是从1开始计算的
    void backtrack(int k, int n, int sum, int startIndex){
        // 什么时候结束呢？当pth和k大小一样的时候
        if(path.size()==k){
            if(sum==n){
                res.push_back(path);
            }
        }
        if(sum>n){
            return;
        }
        for(int i = startIndex;i<=9;i++){
            path.push_back(i);
            sum+=i;
            backtrack(k, n, sum, i+1);
            sum-=i;
            path.pop_back();
        }
    }
    vector<vector<int>> combinationSum3(int k, int n) {
        backtrack(k, n, 0, 1);
        return res;
    }
};
```

三、排列问题

1. 全排列-46

- 给定 不含重复 数字的数组 $nums$
- 返回所有可能的全排列

- 排列问题是需要去重的， $used$ 数组来记录 $path$ 里哪些元素已经使用过

- 排列问题不需要 `startIndex`
- `if(used[i]==false){}` , 如果没有使用过, 继续进行判断。

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;

    void backtrack(vector<int>& nums, vector<bool> used){
        if(path.size()==nums.size()){
            res.push_back(path);
            return;
        }

        for(int i=0;i<nums.size();i++){
            if(used[i]==false){
                used[i] = true;
                path.push_back(nums[i]);
                backtrack(nums, used);
                path.pop_back();
                used[i] = false;
            }
        }
    }

    vector<vector<int>> permute(vector<int>& nums) {
        vector<bool> used(nums.size(), false);
        backtrack(nums, used);
        return res;
    }
};
```

2. 全排列 II-47

- 给定 含重复 数字的数组 `nums`
- 返回所有可能的 不重复 的全排列

- 排列问题是需要去重的, `used` 数组来记录 `path` 里哪些元素已经使用过
- 排列问题不需要 `startIndex`
- 利用 `used` 数组实现去重, 具体来说分为两步
 - `if(i>0&&nums[i]==nums[i-s]&&nums[i-1]==false) continue` , 来跳过同层重复的可能
 - 回溯前 `used[i]=true` , 回溯后 `used[i]=false`
- 由于是去重复, 因此还需要进行排序 `sort`

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
```

```

void backtrack(vector<int>& nums, vector<bool>& used){
    if(path.size()==nums.size()){
        res.push_back(path);
        return;
    }
    for(int i=0;i<nums.size();i++){
        // if(i>0&&nums[i]==nums[i-1]&&used[i-1]==true) continue;
        // 这种写法也是可以的
        if(i>0&&nums[i]==nums[i-1]&&used[i-1]==false) continue;
        if(used[i]==false){
            used[i] = true;
            path.push_back(nums[i]);
            backtrack(nums, used);
            path.pop_back();
            used[i] = false;
        }
    }
}

vector<vector<int>>> permuteUnique(vector<int>& nums) {
    vector<bool> used(nums.size(), false);
    sort(nums.begin(), nums.end());
    // 注意，是不是对结果去重的时候都需要排序
    backtrack(nums, used);
    return res;
}
};

```

四、分割问题

1. 分割回文串-131

- 给定一个字符串 *s*，将字符串分割成子串，是的每个子串都是回文子串
- 返回 *s* 所有可能的分割方案

```

class Solution {
public:
    vector<vector<string>>> res;
    vector<string> path;

    void backtrack(const string& s, int startIndex){
        if(startIndex>=s.size()){
            res.push_back(path);
            return;
        }
        for(int i = startIndex; i<s.size();i++){
            if(isPalindrome(s, startIndex, i)){
                // [startIndex, i]
                string str = s.substr(startIndex, i - startIndex + 1);
                path.push_back(str);
            }
        }
    }
};

```



```

    }
    else{
        continue;
    }
    backtrack(s, i+1);
    path.pop_back();
}
}

bool isPalindrome(const string& s, int left, int right){
    while(left<right){
        if(s[left]==s[right]){
            left++;
            right--;
        }
        else return false;
    }
    return true;
}

vector<vector<string>> partition(string s) {
    backtrack(s, 0);
    return res;
}

};

```

五、二叉树问题

- 递归和回溯的关系密不可分，而对于一些路径问题更适合用回溯来进行遍历。

1. 路径总和-112

- 给定了二叉树的根节点 `root` 和整数目标 `targetSum`
- 判断是否存在从根结点到叶子点的路径，使得路径和为 `targetSum`

```

class Solution {
public:
    bool backtrack(TreeNode* node, int targetSum, int sum){
        // 先添加根节点
        cout<<"current node:"<<node->val<<" , sum="<<sum<<endl;
        if (node->left == NULL && node->right == NULL) {
            if (sum==targetSum) {
                return true;
            }
        }

        if (node->left != NULL) {
            sum += node->left->val;
            // 如果满足要求，直接跳出
            if(backtrack(node->left, targetSum, sum)) return true;
            sum -= node->left->val;
        }
    }
}

```

```

        if (node->right != NULL) {
            sum += node->right->val;
            if(backtrack(node->right, targetSum, sum)) return true;
            sum -= node->right->val;
        }
        return false;
    }

    bool hasPathSum(TreeNode* root, int sum) {
        if (root == NULL) return false;
        return backtrack(root, sum, root->val);
    }
};

```

2. 路径总和II-113

- 给定了二叉树的根节点 `root` 和整数目标 `targetSum`
- 找到所有从根结点到叶子点的路径，使得路径和为 `targetSum`

- 初始值需要将根节点的值 `root->val` 存入到 `path` 中
- 同时将 `sum` 的初值更新为 `root->val`

```

/**
 * 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>>> res;
    vector<int> path;
    void backtrack(TreeNode* node, int targetSum, int sum){
        // 先添加根节点
        cout<<"current node:"<<node->val<<" , sum="<<sum<<endl;
        if (node->left == NULL && node->right == NULL) {
            if (sum==targetSum) {
                res.push_back(path);
            }
        }
        // 负数的情况下 -2>-5 不能进行剪枝
        // if (sum > targetSum) return;
        if (node->left != NULL) {
            path.push_back(node->left->val);
            sum += node->left->val;
            backtrack(node->left, targetSum, sum);
            sum -= node->left->val;
        }
        if (node->right != NULL) {
            path.push_back(node->right->val);
            sum += node->right->val;
            backtrack(node->right, targetSum, sum);
            sum -= node->right->val;
        }
    }
};

```

```

        path.pop_back();
    }
    if (node->right != NULL) {
        path.push_back(node->right->val);
        sum += node->right->val;
        backtrack(node->right, targetSum, sum);
        sum -= node->right->val;
        path.pop_back();
    }
}

vector<vector<int>> pathSum(TreeNode* root, int targetSum) {
    // return null vector
    if(root==NULL) return {};
    // 先将根节点的值添加到路径中
    path.push_back(root->val);
    // 同时sum的值也更新为根节点的值，作为初值
    backtrack(root, targetSum, root->val);
    return res;
}
};

```

六、总结

子集问题	给定数组，返回有多少个符合条件的子集	
组合问题	给定数组，找目标和为target的不同组合，或者其他规律的组合	组合无序
排列问题	给定数组，找全部的排列方式	排列有序
二叉树问题	给定二叉树，找到满足条件的所有路径	

回溯问题	数组元素	抽取元素	回溯索引	startIndex	used	是否排序
子集	不重复	不重复	i+1	0		
子集II	存在重复	不重复	i+1	0	使用	排序
组合	不重复	不重复	i+1	1, 范围是 [1,n]		

回溯问题	数组元素	抽取元素	回溯索引	startIndex	used	是否排序
组合总和	不重复	可重复	<code>i</code>	0		
组合总和 II	存在重复	不重复	<code>i+1</code>	0	使用	排序
组合总和 III	不重复	不重复	<code>i+1</code>	<code>1</code> , 范围是 <code>[1,9]</code>		
全排列	不重复	不重复	无	无	用	
全排列 II	存在重复	不重复	无	无	使用	排序

- 对于数组元素存在重复的情况下，应该使用 `used` 数组，`跳过` 同层使用过的情况。
 - `if(i>0&&nums[i]==nums[i-1]&&used[i-1]==false) continue;`
- 对于排列问题，应该使用 `used` 数组来记录数字是否使用过，如果没有用过，那么接下来继续讨论。
 - `if(used[i]==false) {};`
- 如果抽取元素 `可重复`，回溯的索引是 `i`，否则为 `i+1`

B. 广度优先算法-BFS

一、岛屿问题

- 注意输入的是数字还是字符串

1. 被围绕的区域-130

- 将 `陆地` 填为 `海水`
 - 陆地: `"0"`
 - 海水: `"X"`
- 边界的陆地不会被填充
- 解法有问题，后期再修改。

```
class Solution {
public:
    void dfs(vector<vector<char>>& board, int i, int j) {
        int m = board.size();
        int n = board[0].size();
        if (i < 0 || i >= m || j < 0 || j >= n) return;
        // 这个逻辑写的有问题，不能直接返回这个，边上的不能直接覆盖。
```

```

        if (board[i][j] == 'X') return;
        board[i][j] = 'X';
        dfs(board, i - 1, j);
        dfs(board, i + 1, j);
        dfs(board, i, j - 1);
        dfs(board, i, j + 1);
    }

    void solve(vector<vector<char>>& board) {
        int m = board.size();
        int n = board[0].size();

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

```

2. 统计封闭岛屿的数目-1254

- 将 陆地 填为 海水
 - 陆地: 0
 - 海水: 1

```

class Solution {
public:
    // 还需要排除边界的情况， 可以先将四周的海水淹没
    void dfs(vector<vector<int>>& grid, int i, int j) {
        int m = grid.size();
        int n = grid[0].size();
        if (i < 0 || i >= m || j < 0 || j >= n) return;
        if (grid[i][j] == 1) return;
        grid[i][j] = 1;
        dfs(grid, i - 1, j);
        dfs(grid, i + 1, j);
        dfs(grid, i, j - 1);
        dfs(grid, i, j + 1);
    }

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

```

```

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

for (int j = 0; j < n; j++) {
    dfs(grid, 0, j);
    dfs(grid, m - 1, j);
}

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

```

3. 岛屿数量-200

将 陆地 填为 海水

- 陆地: '1'
- 海水: '0'

```

class Solution {
public:
    // 1--> island
    void dfs(vector<vector<char>>& grid, int i, int j) {
        int m = grid.size();
        int n = grid[0].size();
        if (i < 0 || i >= m || j < 0 || j >= n) return;
        if (grid[i][j] == '0') return;
        grid[i][j] = '0';
        dfs(grid, i - 1, j);
        dfs(grid, i + 1, j);
        dfs(grid, i, j - 1);
        dfs(grid, i, j + 1);
    }

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

```

```

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

```

3. 不同的岛屿数量-694

将 陆地 填为 海水

- 陆地: 1
- 海水: 0

- 回溯后添加字符保证走过的路线不相同

```

// int dir[4][2] = {{1, 0}, {-1, 0}, {0, -1}, {0, 1}};
class Solution {
public:
    void dfs(vector<vector<int>>& grid, int i, int j, string& temp, char c){
        int m = grid.size();
        int n = grid[0].size();
        if (i < 0 || i >= m || j < 0 || j >= n) return;
        if (grid[i][j] == 0) return;
        grid[i][j] = 0;

        temp += c;

        dfs(grid, i + 1, j, temp, 'r');
        dfs(grid, i - 1, j, temp, 'l');
        dfs(grid, i, j + 1, temp, 'd');
        dfs(grid, i, j - 1, temp, 'u');

        temp += '6666666666';
    }

    int numDistinctIslands(vector<vector<int>>& grid) {
        int m = grid.size();
        int n = grid[0].size();
        int res = 0;
        set<string> visited;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == 1) {
                    string aux = "";
                    dfs(grid, i, j, aux, 's');

```

```

        if (visited.count(aux) == 0) {
            res++;
            visited.insert(aux);
        }
    }
}
return res;
}
};

```

4. 岛屿的最大面积-695

将 陆地 填为 海水

- 陆地: 1
- 海水: 0

```

class Solution {
public:
    int dfs(vector<vector<int>>& grid, int i, int j) {
        int m = grid.size();
        int n = grid[0].size();
        if (i < 0 || i >= m || j < 0 || j >= n) return 0;
        if (grid[i][j] == 0) return 0;
        grid[i][j] = 0;

        int area = 1 + dfs(grid, i - 1, j) + dfs(grid, i + 1, j) + dfs(grid, i,
j - 1) + dfs(grid, i, j + 1);
        return area;
    }
    int maxAreaOfIsland(vector<vector<int>>& grid) {
        int m = grid.size();
        int n = grid[0].size();
        int res = 0;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == 1) {
                    int area = dfs(grid, i, j);
                    res = max(res, area);
                }
            }
        }
        return res;
    }
};

```


5. 统计子岛屿-1905

将 陆地 填为 海水

- 陆地: 1
- 海水: 0

- 将第一个为海水，第二个为陆地的情况时，将第二个陆地淹没。
- 那么再次遍历，由于排除了不合法的情况，那么第二个为陆地的时候，进行计数。

```
class Solution {
public:
    void dfs(vector<vector<int>>& grid, int i, int j) {
        int m = grid.size();
        int n = grid[0].size();
        if (i < 0 || i >= m || j < 0 || j >= n) return;
        if (grid[i][j] == 0) return;
        grid[i][j] = 0;
        dfs(grid, i - 1, j);
        dfs(grid, i + 1, j);
        dfs(grid, i, j - 1);
        dfs(grid, i, j + 1);
    }

    int countSubIslands(vector<vector<int>>& grid1, vector<vector<int>>& grid2)
    {
        // 判断两个岛屿是否被包含呢？
        // 先排除一些不合理的情况...
        // 如果左侧的位置是0但是右侧的位置是1，说明一定是不被包含的情况；
        int m = grid1.size();
        int n = grid1[0].size();
        int res = 0;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid1[i][j] == 0 && grid2[i][j] == 1) {
                    // 将不合理的情况，直接淹没!!!
                    dfs(grid2, i, j);
                }
            }
        }

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid2[i][j] == 1) {
                    // 将不合理的情况，直接淹没!!! 那么剩下的情况，肯定是被包含的了。
                    res++;
                    dfs(grid2, i, j);
                }
            }
        }
        return res;
    }
}
```

```
};
```

6. 飞地的数量-1020

将 陆地 填为 海水

- 陆地: 1
- 海水: 0

- 相当于计算封闭岛屿的陆地总数
- 统计 陆地 的面积

```
class Solution {
public:
    // 还需要排除边界的情况， 可以先将四周的海水淹没
    void dfs(vector<vector<int>>& grid, int i, int j) {
        int m = grid.size();
        int n = grid[0].size();
        if (i < 0 || i >= m || j < 0 || j >= n) return;
        if (grid[i][j] == 0) return;
        grid[i][j] = 0;
        dfs(grid, i - 1, j);
        dfs(grid, i + 1, j);
        dfs(grid, i, j - 1);
        dfs(grid, i, j + 1);
    }

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

        for (int j = 0; j < n; j++) {
            dfs(grid, 0, j);
            dfs(grid, m - 1, j);
        }

        for (int i = 0; i < m; i++) {
            dfs(grid, i, 0);
            dfs(grid, i, n - 1);
        }

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

        return res;
    }
};
```

二、二叉树

1. 路径总和-III-43

- 给定了二叉树的根节点 `root` 和整数目标 `targetSum`
- 找到所有从 父节点 到 子结点 的路径，使得路径和为 `targetSum`