算法设计之深度优先搜索

(1)104. 二叉树的最大深度

Description

给定一个二叉树，找出其最大深度。二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

Solution

很简单，直接递归求深度

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

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

struct TreeNode *CreatBinaryTree(vector<int>& nums){
    int a = nums.front(); nums.erase(nums.begin());
```

```
    if(a == 0) return NULL;//愚蠢的做法，多少叶子就需要输入2倍个 0 才能结束创建进
程

    struct TreeNode *newnode;
    newnode = (struct TreeNode*)malloc( sizeof(struct TreeNode) );
    newnode->val = a;
    newnode->left  = CreatBinaryTree(nums);  //递归创建左子树
    newnode->right = CreatBinaryTree(nums);  //递归创建右子树
    return newnode;
}

class Solution {
public:
    int maxDepth(TreeNode* root) {
        if(root == NULL) return 0;
        return max(maxDepth(root->left), maxDepth(root->right)) + 1;
    }
};

int main(){
    Solution solu;
    struct TreeNode* root;
    vector<int> nums = {3,9,0,0,20,15,0,0,7,0,0} ;
    root = CreatBinaryTree(nums);
    cout << solu.maxDepth(root);
    return 0;
}
```

---

(2)101. 对称二叉树

Description

给定一个二叉树，检查它是否是镜像对称的。

Solution

对 root 底下的左右子树进行
左子树先序遍历
右子树后序遍历
一旦遇到数不同，则不对称，结束算法

结合到这道题上
return isTreesSymmetric(root1->left, root2->right) && isTreesSymmetric(root1->right,

root2->left);

这句用的很为巧妙

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

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

struct TreeNode *CreatBinaryTree(){
    int a;
    cin >> a;   //愚蠢的做法，多少叶子就需要输入2倍个 0 才能结束创建进程
    if(a == 0) return NULL;

    struct TreeNode *newnode;
    newnode = (struct TreeNode*)malloc( sizeof(struct TreeNode) );
    newnode->val = a;
    newnode->left  = CreatBinaryTree();  //递归创建左子树
    newnode->right = CreatBinaryTree();  //递归创建右子树
    return newnode;
}

class Solution {
private:
    bool isTreesSymmetric(TreeNode* root1, TreeNode* root2){
        if(root1==NULL && root2==NULL) return true;

        if(root1 && root2 && root1->val == root2->val){
            return isTreesSymmetric(root1->left, root2->right) && isTrees
Symmetric(root1->right, root2->left);
        }
        return false;
    }
public:
    bool isSymmetric(TreeNode* root) {
        if(root==NULL)  return true;
        return isTreesSymmetric(root->left, root->right);
    }
};

int main(){
    Solution solu;
    struct TreeNode* root;
    root = CreatBinaryTree();

    cout << boolalpha << solu.isSymmetric(root);
```

```
    return 0;
}
```

---

(3)733. 图像渲染

Description

给你一个坐标 (sr, sc) 表示起点和一个新的颜色值 newColor，让你重新上色这幅图像。 意思是将所有有与起点相连的，且数值与起点数值一样的这块区域数值 改为 newColor。说明：相连是指上下左右四个方向上相连

Solution

使用递归，从起点开始，朝四个方向改过去，不断更新迭代

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

class Solution {
private:
    void dfs(vector<vector<int>>& image, int sr, int sc, int newColor,int
 oldColor){
        int r = image.size();
        int c = image[0].size();
        int dir[][2] = {{-1,0},{0,-1},{1,0},{0,1}}; //4个方向
        image[sr][sc]=newColor;
        for(int i = 0; i < 4; ++i){
            int x = sr+dir[i][0];
            int y = sc+dir[i][1];
            if(x>=r||x<0||y>=c||y<0||image[x][y]!=oldColor) continue;
            dfs(image, x, y, newColor,oldColor);
        }
    }

public:
    vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int
 sc, int newColor) {
        int oldColor = image[sr][sc];
        if(oldColor!=newColor)
        dfs(image,sr,sc,newColor,oldColor);
        return image;
    }
```

```
};

void output(vector<vector<int>> image) {
    for(auto line:image){
        for(auto e:line) cout<<e<<' ';
        cout<<endl;
    }
}

int main(){
    Solution solu;
    vector<vector<int>> image = {{1,1,1},{1,1,0},{1,0,1}};
    output(solu.floodFill(image, 1, 1, 2)) ;
    return 0;
}
```

---

(4)111. 二叉树的最小深度

Description

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。

Solution

方法: 和求最大深度相反
但不是拿那个代码简单改个大于小于号就行
使用递归求深度
因为不是遍历到最深，所以需考虑节点是否为叶子 。说明: 叶子节点是指没有子节点的节点。

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

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

struct TreeNode *CreatBinaryTree(){
    int a;
```

```
    cin >> a;    //愚蠢的做法，多少叶子就需要输入2倍个 0 才能结束创建进程
    if(a == 0) return NULL;

    struct TreeNode *newnode;
    newnode = (struct TreeNode*)malloc( sizeof(struct TreeNode) );
    newnode->val = a;
    newnode->left  = CreatBinaryTree();  //递归创建左子树
    newnode->right = CreatBinaryTree();  //递归创建右子树
    return newnode;
}

class Solution {
public:
    int minDepth(TreeNode* root) {
        if(root == NULL) return 0;
        if(root->left  == NULL) return minDepth(root->right) + 1;
        if(root->right == NULL) return minDepth(root->left)  + 1;
        return min(minDepth(root->left), minDepth(root->right)) + 1;
    }
};

int main(){
    Solution solu;
    struct TreeNode* root;
    root = CreatBinaryTree();

    cout << solu.minDepth(root);
    return 0;
}

/*
输入例子
3 9 0 0 20 15 0 0 7 0 0

1 0 2 0 0

1 2 0 0 0
*/
```

---

(5)690. 员工的重要性

Description

给定一个保存员工信息的数据结构，它包含了员工唯一的id，重要度 和 直系下属的id。现在输入一个公司的所有员工信息，以及单个员工id，返回这个员工和他所有下属的重要度之和。

Solution

思路简单

直接递归即可

难点是

类的构造和指针的使用

vector<Employee*> 容器内放的是这个类的地址，用&符号引用地址

更正：

测试用例中下属id不是按数组下标的来

改进

用一个for来找id，当数据很大时会花费很多时间。使用map来记录id，就能很快找到了

```
#include <vector>
#include <iostream>
#include <unordered_map>
using namespace std;

class Employee {
public:
    int id;
    int importance;
    vector<int> subordinates;

    Employee() { };

    void set(int a, int b, vector<int> c) {
        id = a;
        importance = b;
        subordinates = c;
    }
};

class Solution {
public:
    unordered_map<int,Employee*> mm; //无序容器

    int dp(int id){
        int sum = mm[id]->importance; //自身重要度
        for(auto c:mm[id]->subordinates)  sum += dp(c); //下属重要度
        return sum;
    }
    int getImportance(vector<Employee*> employees, int id) {
        for (auto* p:employees) mm[p->id]=p; //vector 改用 map容器，方便查找

        return dp(id);
    }
};

int main() {
```

```
    Solution solu;
    vector<Employee*> ve;

    vector<int> c1,c2;
    c1.push_back(2);
    c1.push_back(3);
    Employee empl[3];
    empl[0].set(1,5,c1);
    empl[1].set(2,3,c2);
    empl[2].set(3,3,c2);
    ve.push_back(&empl[0]);
    ve.push_back(&empl[1]);
    ve.push_back(&empl[2]);

    cout << solu.getImportance(ve,3);
    return 0;
}
```

---

(6)841. 钥匙和房间

Description

有 N 个房间，每个房间 i 都放着一堆钥匙列表 rooms[i]，对于每个钥匙由 [0,1，…，N-1] 中的一个整数表示

rooms[i][j] 表示在 i 房间里有打开房间j的钥匙

最初，除 0 号房间外的其余所有房间都被锁住。

请判断你能否进入每个房间，如果能返回 true，否则返回 false。

Solution

思路

使用 set 容器

用 set 存储未被打开的房间

打开了房间就从 set 中删去

一旦 set 为空，说明能进入所有房间

```
#include <set>
#include <vector>
#include <iostream>
using namespace std;

class Solution {
```

```
public:
    set<int> lockrooms;
    void openrooms(vector<vector<int> >& rooms,int i){ //打开房间i
        for(auto e : rooms[i]){
            if(lockrooms.count(e)==0) continue; //房间e已打开，这个钥匙e没用

            lockrooms.erase(e); //删除，即打开
            openrooms(rooms,e); //开房
        }
    }
    bool canVisitAllRooms(vector<vector<int> >& rooms) {
        for(int i=1; i<rooms.size(); i++) lockrooms.insert(i); //0号开着，
取余锁着的房间放入set容器
        openrooms(rooms,0);
        return lockrooms.empty();
    }
};

int main() {
    Solution solu;
    vector<vector<int> > rooms = {{1},{2},{3},{}};
    cout << boolalpha << solu.canVisitAllRooms(rooms);
    return 0;
}
```

---

(7)113. 路径总和 II

Description

给定一个二叉树和一个目标和，找到所有从根节点到叶子节点路径总和等于给定目标和的路径。

Solution

思路：

粗暴的，不进行剪枝

先求出所有根节点到子节点的和，并且生成路径

然后for循环，挑出和与目标相等的路径，返回

说明：

本例代码主要目的是生成二叉树所有路径与路径和

```
#include <vector>
```

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

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

struct TreeNode *CreatBinaryTree(vector<int> &nums){
    int a = nums.front(); nums.erase(nums.begin());
    if(a == 0) return NULL;

    struct TreeNode *newnode;
    newnode = (struct TreeNode*)malloc( sizeof(struct TreeNode) );
    newnode->val = a;
    newnode->left  = CreatBinaryTree(nums);  //递归创建左子树
    newnode->right = CreatBinaryTree(nums);  //递归创建右子树
    return newnode;
}

void output_2V(vector<vector<int> > matrix){
    for(auto nums:matrix){
        for(auto e:nums) cout<<e;
        cout<<endl;
    }
    cout<<endl;
}

class Solution {
public:
    vector< pair<int,vector<int> > > treAllPaths;
    void dfs(TreeNode* node, int sum,vector<int> path){
        path.push_back(node->val);
        sum += node->val;

        //遍历到子节点
        if(node->left==NULL && node->right==NULL) treAllPaths.push_back(
 make_pair(sum, path) );

        if(node->left!=NULL)  dfs(node->left, sum,path);
        if(node->right!=NULL) dfs(node->right,sum,path);
    }
    vector<vector<int>> pathSum(TreeNode* root, int sum) {
        vector<vector<int>> result;

        if(root!=NULL){
            vector<int> path;
            dfs(root,0,path);
            for(auto e:treAllPaths){
```

```
                if(e.first == sum) result.push_back(e.second);
            }
        }
        return result;
    }
};

int main(){
    Solution solu;
    vector<int> nums = {5,4,11,7,0,0,2,0,0,0,8,13,0,0,4,5,0,0,1,0,0};
    struct TreeNode* root;
    root = CreatBinaryTree(nums);
    output_2V(solu.pathSum(root,22));
    return 0;
}
```

加入剪枝

就是对sum值判断，如果sum值大于目标值了，这条路径就剪去

更正：

剪枝方法行不通，因为测试用例中val值有负数

改进

只存需要的路径，就不用最后for循环查找。就是判断子节点处sum等于目标值时，才把该路径存入；

```
class Solution {
public:
    vector<vector<int>> result;
    vector<int> path;
    int target;
    int cursum;

    void dfs(TreeNode* root){
        if(root==NULL) return;

        path.push_back(root->val);
        cursum += root->val;

        //遍历到子节点
        if(root->left==NULL && root->right==NULL && cursum==target) resul
t.push_back( path );

        dfs(root->left);
        dfs(root->right);

        path.pop_back(); //这个就是回退
        cursum -= root->val;
    }
    vector<vector<int>> pathSum(TreeNode* root, int sum) {
        target = sum;
        cursum = 0;
```

```
        dfs(root);
        return result;
    }
};
```

(8)130. 被围绕的区域

Description

给定一个二维的矩阵，包含 'X' 和 'O'（字母 O）。找到所有被 'X' 围绕的区域，并将这些区域里所有的 'O' 用 'X' 填充。

被围绕的区间不会存在于边界上，任何边界上的 O 都不会被填充为 X，任何不在边界上，或不与边界上的 O 相连的 O 最终都会被填充为 X

Solution

思路：

遍历整个数组，遇到O字符时

通过递归找到与这个O相连的这块区域，再判断这块区域是否是被围绕的

改进

逆向思考，从边界的点出发

直接找四边的O字符，直接改为N字符作为标记，递归找到相连的O字符，也直接改为N字符

如此，当四边的O字符都这样处理完后，剩下的O字符都是被围绕的了

最后遍历整个数组，O字符改为X，不要忘记把N字符改回O

```
#include <queue>
#include <iostream>
#include <algorithm>
using namespace std;

class Solution {
public:
    void dfs(vector<vector<char>>& board, vector<pair<int, int>> &dir, pa
ir<int,int> p) {
        board[p.first][p.second] = 'N';

        for (auto e:dir) {
            int x = p.first + e.first;
            int y = p.second + e.second;
            if (x<0 || board.size()<=x || y<0 || board[0].size()<=y) cont
inue;

            if (board[x][y] == 'O')  dfs(board, dir, make_pair(x,y));
        }
```

```
    }
    void solve(vector<vector<char>>& board) {
        if ( board.empty() ) return;

        vector<pair<int, int>> dir({{0, 1}, {0, -1}, {1, 0}, {-1, 0}});
        int n = board.size(), m = board[0].size();
        for (int j = 0; j < m; j++) {
            if(board[0][j]=='O')    dfs(board, dir, make_pair(0,j));
            if(board[n-1][j]=='O')  dfs(board, dir, make_pair(n-1,j));
        }
        for (int i = 0; i < n; i++) {
            if(board[i][0]=='O')    dfs(board, dir, make_pair(i,0));
            if(board[i][m-1]=='O')  dfs(board, dir, make_pair(i,m-1));
        }


        for (int i = 0; i < n; i++)
            for (int j = 0; j < m; j++)
                board[i][j] = (board[i][j] == 'N' ? 'O' : 'X');
    }
};

void output_2V(vector<vector<char> > matrix){
    for(auto nums:matrix){
        for(auto e:nums) cout<<e;
        cout<<endl;
    }
}

int main(){
    Solution solu;
    vector<vector<char>> board = {{'X','X','X','X'},{'X','O','O','X'},{
'X','X','O','X'},{'X','O','X','X'}};
    solu.solve(board);
    output_2V(board);
    return 0;
}
```

(9)417. 太平洋大西洋水流问题

Description

给定一个 m x n 的非负整数矩阵来表示一片大陆上各个单元格的高度。“太平洋”处于大陆的左边界和上边界，而“大西洋”处于大陆的右边界和下边界。

规定水流只能按照上、下、左、右四个方向流动，且只能从高到低或者在同等高度上流动。

请找出那些水流既可以流动到“太平洋”，又能流动到“大西洋”的陆地单元的坐标。

Solution

思路：

一种比较自然的思路是依次对每个点进行 bfs 看其是否既能到达太平洋又能到达大西洋

这种思路会导致做很多重复的操作以致超时

转换思路

从目标出发去寻找源，可以从与太平洋和大西洋直接相邻的点（也就是矩阵边缘上的点）出发进行 bfs 去遍历矩阵中所有的点

遍历到的点肯定是能够到达某个海洋的点

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

class Solution {
public:
    vector<pair<int, int>> dir = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};
    int n , m;

    void bfs(vector<vector<int>>& matrix, vector<vector<bool>>& visit, pa
ir<int, int> tmp) {
        if(visit[tmp.first][tmp.second]==true) return;
        visit[tmp.first][tmp.second] = true;

        for(auto &v:dir) {
            int x = tmp.first + v.first;
            int y = tmp.second + v.second;
            if (x<0 || n<=x || y<0 || m<=y) continue;

            if (visit[x][y]==false && matrix[x][y]>=matrix[tmp.first][tm
p.second])
                bfs(matrix, visit, make_pair(x,y));
        }
    }

    vector<pair<int, int>> pacificAtlantic(vector<vector<int>>& matrix) {
        vector<pair<int, int>> result;
        if ( matrix.empty() ) return result;

        n = matrix.size();
        m = matrix[0].size();
        vector<vector<bool>> pVisit(n, vector<bool>(m, false)); // Pacifi
c
        vector<vector<bool>> aVisit(n, vector<bool>(m, false)); // Atlant
ic
        for (int j = 0; j < m; j++) {
            bfs(matrix, pVisit, {0,j} );
            bfs(matrix, aVisit, {n-1,j} );
        }
```

```
            for (int i = 0; i < n; i++) {
                bfs(matrix, pVisit, make_pair(i,0));
                bfs(matrix, aVisit, {i,m-1});
            }

            for (int i = 0; i < n; i++)
                for (int j = 0; j < m; j++)
                    if (pVisit[i][j] && aVisit[i][j])
                        result.push_back({i, j});
            return result;
        }
    };

    void output_pair(vector<pair<int, int>> matrix){
        for(auto e:matrix)
            cout<<e.first<<e.second<<endl;
    }

    int main() {
        Solution solu;
        vector<vector<int>> matrix({{1,2,2,3,5},{3,2,3,4,4},{2,4,5,3,1},{6,7,
    1,4,5},{5,1,1,2,4}});
        output_pair(solu.pacificAtlantic(matrix));
        return 0;
    }
```

(10)542. 01 矩阵

Description

给定一个由 0 和 1 组成的矩阵，找出每个元素到最近的 0 的距离。两个相邻元素间的距离为 1 。

Solution

思路：

找到所有的 0 作为递归起点，每个 0 都进行 dfs，更新距离

直到周围没有 1 或者 周围1到最近的0的距离 都不大于到该 0 距离时，停止dfs

使用深度优先 - 递归

做法超时了

改进

不应用深度优先，应用广度优先

因为使用深度优先的话：下一个 0 的dfs 可能会覆盖前一个 0 的dfs，这样就造成时间上的浪费

但是如果用广度优先的话：找到所有与该0距离为1的点，标记，结束；再找与下一个0距离为1的点，标记，结束

。。。
从距离为1的点出发，找到下一个距离0为2的点。。。
所以，一旦点的距离被确定就一定是到最近的0的距离

```
#include <queue>
#include <iostream>
#include <algorithm>
using namespace std;

class Solution {
public:
    vector<vector<int>> updateMatrix(vector<vector<int>>& matrix) {
        if ( matrix.empty() ) return {};

        vector<pair<int,int>> dir{{1,0},{-1,0},{0,1},{0,-1}};
        int n=matrix.size(), m=matrix[0].size();
        vector<vector<int>> result(n, vector<int>(m,0) );

        queue<pair<int, int>> q;
        for(int i=0; i<n; i++)  for(int j=0; j<m; j++){
            if(matrix[i][j]==0) q.push({i,j}); //纳入所有0的位置作为递归起点
            else  result[i][j]=-1; //标记-1的是距离还未确定的
        }

        while( !q.empty() ) { //广度优先
            auto tmp = q.front(); q.pop();

            for(auto v:dir) {
                int x = tmp.first + v.first;
                int y = tmp.second + v.second;
                if (x<0 || n<=x || y<0 || m<=y) continue;

                if (result[x][y]==-1) {
                    result[x][y] = result[tmp.first][tmp.second]+1;
                    q.push({x,y});
                }
            }
        }
        return result;
    }

};


void output_2V(vector<vector<int>> matrix){
    for (auto line:matrix){
        for (auto e:line) cout<<e;
        cout<<endl;
    }
}
```

```
int main() {
    Solution solu;
    vector<vector<int>> matrix({{0,0,0},{0,1,0},{1,1,1}});
    output_2V(solu.updateMatrix(matrix));
    return 0;
}
```

(11)473. 火柴拼正方形

Description

输入每根火柴长度。判断是否能用所有的火柴拼成正方形

Solution

暴力解法

每条火柴分别分配到四条边，穷举所有组合，直到有成功的方案返回

改进：

通过所有火柴长度和可以求出正方形边长，

得到剪枝条件：一旦长度超过边长，该方案不成立，剪枝

简化判断成功条件：只要能组合出三条边，第四条一定能成功

更快达到剪枝条件：火柴按长度从大到小排序

Submit

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

class Solution {
private:
    int circumfer; //正方形周长
    int sideLength; //正方形边长
    vector<int> sides = {0,0,0,0}; //四条边

    bool dfs(const vector<int>& nums, int id) {
        if(id == nums.size())
            return sides[0]==sideLength && sides[1]==sideLength && sides[
2]==sideLength;

        for (int i=0;i<4;i++) {
            if (sides[i]+nums[id] > sideLength) continue; //剪枝
```

```
                sides[i] += nums[id];
                if(dfs(nums,id+1)) return true; //剪枝，有成功的方案，算法停止
                sides[i] -= nums[id];
            }
            return 0;
        }
    public:
        bool makesquare(vector<int>& nums) {
            if (nums.size() < 4) return false;

            circumfer = accumulate(nums.begin(),nums.end(),0); //所有火柴长度和
            if (circumfer%4 != 0) return false;

            sort(nums.begin(),nums.end(),greater<int>()); //降序
            sideLength = circumfer/4;
            if (nums[0] > sideLength) return false;

            return dfs(nums,0);
        }
    };

    int main() {
        Solution solu;
        vector<int> nums({1,1,2,2,2});
        cout << solu.makesquare(nums);
        return 0;
    }
```

(12)773. 滑动谜题

Description

在一个 2 x 3 的板上（board）有 5 块砖瓦，用数字 1~5 来表示, 以及一块空缺用 0 来表示.

0 与一个相邻的数字进行交换.最终当板 board 的结果是 [[1,2,3],[4,5,0]] 谜板被解开。

给出一个谜板的初始状态，返回最少可以通过多少次移动解开谜板，如果不能解开谜板，则返回 -1

Solution

思路：

每个状态都有上下左右四个交换成新状态，用 set 记录状态

如果这个状态已经记录在 set 中，则不在已该状态进行下去

直达达成解开谜题状态，或走完所有状态 结束算法

使用广度优先BFS——队列，

每走一步，获得所有状态，一旦有满足解题状态的，返回步数，结束算法

使用深度优先DFS——递归

需要走遍所有解题方案，获得所有方案所需步数，并找出最小步数方案

亮点：

用string记录状态

二维数组和一维数组相互转化

Submit

```
#include <set>
#include <queue>
#include <iostream>
#include <algorithm>
using namespace std;

class Solution {
public:
    int slidingPuzzle(vector<vector<int>>& board) {
        string now="";
        string des="123450"; //这是解开题的状态
        for(int i=0;i<board.size();i++)
            for(int j=0;j<board[0].size();j++)
                now+=to_string(board[i][j]); //初始状态

        if (now==des) return 0;

        vector<pair<int,int>> dir{{1,0},{-1,0},{0,1},{0,-1}};
        queue<pair<int,string>> q; // q用与广度优先遍历.string记状态，int记该状态是第几步
        set<string> s; //s用于记录所有遍历的状态
        s.insert(now);
        q.push({0,now});
        while (!q.empty()) {

            for(auto &v:dir) {
                auto step = q.front().first;
                auto tmp  = q.front().second;
                int pos = tmp.find('0');
                int x = int(pos/3)+v.first;
                int y = int(pos%3)+v.second;
                if (x<0 || 2<=x || y<0 || 3<=y) continue;

                swap(tmp[pos],tmp[x*3+y]);
                if(tmp==des) return step+1;
                if(s.find(tmp)==s.end()) { //交换后的tmp是新状态
                    s.insert(tmp);
                    q.push({step+1,tmp});
                }
            }
            q.pop();
```

```
            }
            return -1;
        }
    };

    int main() {
        Solution solu;
        vector<vector<int>> matrix({{4,1,2},{5,0,3}});
        cout << solu.slidingPuzzle(matrix);
        return 0;
    }
```