# 并查集及应用

## 一：并查集

是一种数据结构。主要解决元素分组问题，无向图有无环问题。管理一系列不相交的集合，支持2种操作，查找根节点：FindRoot，合并节点：MergeElemet.。

需要抽像的能力将问题抽象成图的动态连通性问题。

# 二：模板伪代码

## 最原始的

查：

int FindRoot(int\* parent, int ele)

{

int i;

if (ele != parent[ele]) {

return FindRoot(parent, parent[ele]);

}

return ele;

}

并：

void Merge(int\* parent, int x, int y)

{

int xRoot, yRoot;

xRoot = FindRoot(parent, x);

yRoot = FindRoot(parent, y);

parent[xRoot] = yRoot;

}

## 优化的

/\*路径压缩：查找根结点的时候节点将同一个集合的点的父节点都置为根节点：递归方法实现\*/其实也可以不用记忆，单纯找根节点就可以了

int FindRootCompress(int\* parent, int ele)

{

    if (ele != parent[ele]) {

        parent[ele] = FindRootCompress(parent, parent[ele]);

        return parent[ele];

    }

    return ele;

}

/\*

按秩合并：即将树层次低的往层次高树合，同高度的合并，父节点的秩要加1

\*/

void MergeEleByRank(int\* parent, int\* rank, int x, int y)

{

    int xRoot, yRoot;

    xRoot = FindRootCompress(parent, x);

    yRoot = FindRootCompress(parent, y);

    // 同属于一个集合无需合并

    if (xRoot == yRoot) {

        return ;

    }

    if (rank[xRoot] > rank[yRoot]) {

        parent[yRoot] = xRoot;

    }

    else if (rank[yRoot] > rank[xRoot]){

        parent[xRoot] = yRoot;

    }

    else {

        parent[xRoot] = yRoot;

        rank[yRoot]++;

    }

}

## 3.小实例

int main()

{

    int parent[Vertices] = {0};

    int rank[Vertices] = {0};

int i;

// 6对亲戚关系，先维护并查集

    int relations[][2] = {

        {0,1},

        {1,2},

        {2,5},

        {1,6},

        {3,6},

        {6,4},

    };

    for (i = 0; i < Vertices; i++) {

        parent[i] = i;

        rank[i] = 1;

    }

    for (i = 0; i < sizeof(relations) / sizeof(relations[0]); i++) {

        MergeEleByRank(parent, rank, relations[i][0], relations[i][1]);

}

// 查看6,9是不是亲戚

int 6Root = FindRootCompress(parent, 6);

int 9Root = FindRootCompress(parent, 9);

if (6Root != 9Root) {

printf(“6 and 9 is not relative\n”);

}

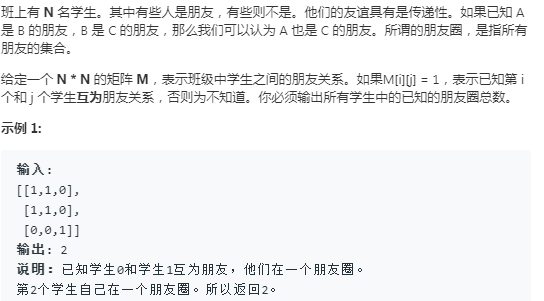
    printf("ele %d father is %d\n", 4, res);

    return 0;

}

# 三：应用及实例

## [547. 朋友圈](https://leetcode-cn.com/problems/friend-circles/)【并查集】



思路：

并查集做法：

初始朋友圈数为Msize（自己和自己算一个圈g\_cycle = MSize）,能merge的圈就少一个(g\_cycke--)，根据输入数组维护并查集，维护好了之后，圈就算出来了。

#include<stdio.h>

#include<stdlib.h>

/\*

并查集做法： 初始朋友圈数为Msize（自己和自己算一个圈g\_cycle = MSize）,能merge的圈就少一个(g\_cycke--)，根据输入数组维护并查集，维护好了之后，圈就算出来了。

对于数组元素为1的才需要去merge，且下表i、j不相同才去merge

\*/

int g\_cycle = 0;

static int box = 0;

int FindRootCompress(int\* parent, int x)

{

    box++;

    printf("x %d box %d\n",x, box);

    if (parent[x] == x) {

        return x;

    }

    parent[x] = FindRootCompress(parent, parent[x]);

    return parent[x];

}

int MergeEleByRank(int\* parent ,int\* rank, int x, int y)

{

    int xRoot, yRoot;

    xRoot = FindRootCompress(parent, x);

    yRoot = FindRootCompress(parent, y);

    if (xRoot == yRoot) {

        return 0;

    }

    g\_cycle--;

    if (rank[xRoot] > rank[yRoot]) {

        parent[yRoot] = xRoot;

    }

    else if(rank[yRoot] < rank[xRoot]) {

        parent[xRoot] = yRoot;

    }

    else {

        parent[xRoot] = yRoot;

        rank[xRoot]++;

    }

    return 1;

}

int findCircleNum(int\*\* M, int MSize, int\* MColSize)

{

    int\* parent = NULL;

    int\* rank = NULL;

    int i, j, iRoot, jRoot;

    int max = (MSize < MColSize[0]) ? MColSize[0] : MSize;

    parent = (int\*)calloc(max, sizeof(int));

    rank = (int\*)calloc(max, sizeof(int));

    g\_cycle = MSize;

    // 初始化，因为和自己都是在同一个朋友圈

    for(i = 0; i < max; i++) {

        parent[i] = i;

        rank[i] = 1;

    }

    for (i = 0; i < MSize; i++) {

        for (j = i + 1; j < MColSize[0]; j++) {

            if (M[i][j] != 1) {

                continue;

            }

            MergeEleByRank(parent, rank, i, j);

        }

    }

    return g\_cycle;

}

int main()

{

    int grid[][3] = {

        {1,1,0},

        {1,1,1},

        {0,1,1},

    };

    int i;

    int res = 0;

    int colSize = sizeof(grid[0]) / sizeof(grid[0][0]);

    int\*\* M = (int\*\*)calloc(sizeof(grid) / sizeof(grid[0]), sizeof(int\*));

    for (i = 0; i < sizeof(grid) / sizeof(grid[0]); i++) {

        M[i] = (int\*)calloc(sizeof(grid[0]) / sizeof(grid[0][0]), sizeof(int));

        memcpy(M[i], grid[i], sizeof(grid[0]) / sizeof(grid[0][0]) \* sizeof(int));

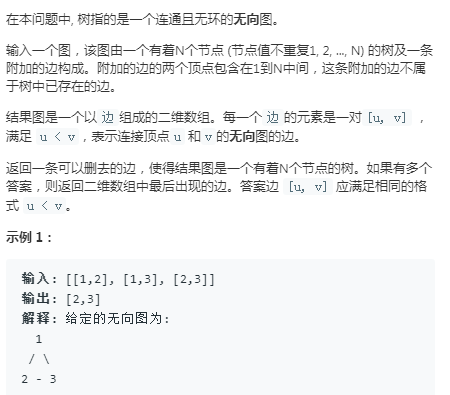
    }

    res = findCircleNum(M, sizeof(grid) / sizeof(grid[0]), &colSize);

    printf("%d\n", res);

}

## 2. [684. 冗余连接](https://leetcode-cn.com/problems/redundant-connection/)【并查集+判定成环】



思路：直接处理输入的边，每条边的顶点合并，当发现不能合并的时候，即当前处理的2个顶点有公共的父节点，则说明当前的边就是答案。

int FindRootCompress(int\* parent, int ele)

{

    if (ele == parent[ele]) {

        return ele;

    }

    parent[ele] = FindRootCompress(parent, parent[ele]);

    return parent[ele];

}

int MergeElementByrank(int\* parent, int\* rank, int x, int y) {

    int xRoot, yRoot;

    xRoot = FindRootCompress(parent, x);

    yRoot = FindRootCompress(parent, y);

    if (xRoot == yRoot) {

        return 0;

    }

    if (rank[xRoot] > rank[yRoot]) {

        parent[yRoot] = xRoot;

    }

    if (rank[xRoot] < rank[yRoot]) {

        parent[xRoot] = yRoot;

    }

    if (rank[xRoot] == rank[yRoot]) {

        parent[yRoot] = xRoot;

        rank[yRoot]++;

    }

    return 1;

}

int\* findRedundantConnection(int\*\* edges, int edgesSize, int\* edgesColSize, int\* returnSize)

{

    int i, j;

    int\* res = (int\*)calloc(2, sizeof(int));

    int\* parent = (int\*)calloc(1001, sizeof(int));

    int\* rank = (int\*)calloc(1001, sizeof(int));

    for (i = 0; i < 1001; i++) {

        parent[i] = i;

        rank[i] = 1;

    }

    for (i = 0; i < edgesSize; i++) {

        // 如果能erge说明当前的2个节点没有共同祖先，即无环

        if (MergeElementByrank(parent, rank, edges[i][0], edges[i][1]) == 1) {

            continue;

        }

        // 否则表示当前的边添加进来会成环，因为这2个点有共同的祖先

        res[0] = edges[i][0];

        res[1] = edges[i][1];

        break;

    }

    if (i < edgesSize) {

        \*returnSize = 2;

        free(parent);

        free(rank);

        return res;

    }

    \*returnSize = 0;

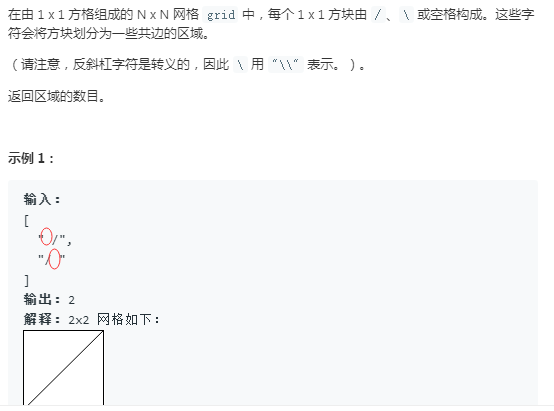
    free(parent);

    free(rank);

    return res;

}

## 3.[959. 由斜杠划分区域](https://leetcode-cn.com/problems/regions-cut-by-slashes/)【并查集+抽像转化】



思路：

并查集思想，将其转化为连通问题。

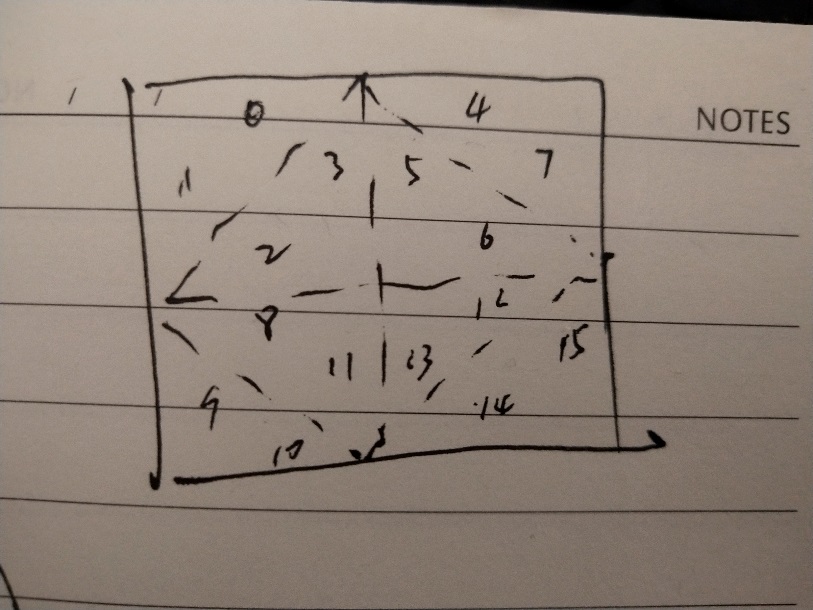
将每个小方格（grid由n\*n的小方格，小方格再转化为4个小三角形）。

从上方的0开始逆时针方向编号。后边的4、5、6、7（其实是0，1，2，3分别加4得到）。（1）对于一个小方格，如果碰到‘/’,需要0和1合并，以及2，3合并。如果碰到‘\’则合并0，3；以及合并1和2；如果碰到空，则合并1，2，3，4。

（2）除了自身合并，还要考虑相领的情形。

不管碰到设么符号，左边方格的3都是和右边方格的1（如图中的5）是连在一块的；上边的方格的2都是和下边方格的0（如图上的8是连载一块的），因此除了合并本方格的小三角还要考虑邻近的格子。

（3）处理完成之后，遍历parent数组，找parent[i] == I 的数量即集合数即可。

。

代码如下：

#include<stdio.h>

#include<stdlib.h>

int FindRootCompress(int\* parent, int ele)

{

    if (ele == parent[ele]) {

        return ele;

    }

    parent[ele] = FindRootCompress(parent, parent[ele]);

    return parent[ele];

}

int MergeElemetByRank(int\* parent, int\* rank, int x, int y)

{

    int xRoot, yRoot;

    xRoot = FindRootCompress(parent, x);

    yRoot = FindRootCompress(parent, y);

    if (xRoot == yRoot) {

        return 0;

    }

    if (rank[xRoot] > rank[yRoot]) {

        parent[yRoot] = xRoot;

    }

    if (rank[xRoot] < rank[yRoot]) {

        parent[xRoot] = yRoot;

    }

    if (rank[xRoot] == rank[yRoot]) {

        parent[xRoot] = yRoot;

        rank[yRoot]++;

    }

    return 1;

}

int MergeElemetByRank(int\* parent, int\* rank, int x, int y)

{

    int xRoot, yRoot;

    xRoot = FindRootCompress(parent, x);

    yRoot = FindRootCompress(parent, y);

    if (xRoot == yRoot) {

        return 0;

    }

    if (rank[xRoot] > rank[yRoot]) {

        parent[yRoot] = xRoot;

    }

    if (rank[xRoot] < rank[yRoot]) {

        parent[xRoot] = yRoot;

    }

    if (rank[xRoot] == rank[yRoot]) {

        parent[xRoot] = yRoot;

        rank[yRoot]++;

    }

    return 1;

}

int regionsBySlashes(char \*\* grid, int gridSize)

{

    // 将每个小格子都用'/'或'\'分为4个小三角形,从上放逆时针数为0，1，2，3

    int\* parent = NULL;

    int\* rank = NULL;

    int i, j = 0, col ,temp;

    int ans = 0;

    parent = (int\*)calloc(gridSize \* gridSize \* 4, sizeof(int));

    rank = (int\*)calloc(gridSize \* gridSize \* 4, sizeof(int));

    for (i = 0; i < gridSize \* gridSize \* 4; i++) {

        parent[i] = i;

        rank[i] = 1;

    }

    // 开始处理网络数组

    for (i = 0; i < gridSize; i++) {

        for (j = 0; j < gridSize; j++) {

            // 处理起始序号

            temp = i \* gridSize \* 4 + 4 \* j;

            // 合并0，1；合并2，3

            if (grid[i][j] == '/') {

                MergeElemetByRank(parent, rank, temp + 0, temp + 1);

                MergeElemetByRank(parent, rank, temp + 2, temp + 3);

            }

            // 合并0，2；合并1，3

            if (grid[i][j] == '\\') {

                MergeElemetByRank(parent, rank, temp + 0, temp + 3);

                MergeElemetByRank(parent, rank, temp + 1, temp + 2);

            }

            // 空格合并0，1，2，4

            if (grid[i][j] == ' ') {

                MergeElemetByRank(parent, rank, temp + 0, temp + 1);

                MergeElemetByRank(parent, rank, temp + 2, temp + 3);

                MergeElemetByRank(parent, rank, temp + 1, temp + 2);

            }

            // 如果不是第一行，向上合并，即本格的0和上边的2和并

            if (i != 0) {

                MergeElemetByRank(parent, rank, temp + 0, temp - 4 \* gridSize + 2);

            }

            // 不是最后一行的话，向下合并，即本格的2和下面的0

            if (i != gridSize - 1) {

                MergeElemetByRank(parent, rank, temp + 2, temp + 4 \* gridSize + 0);

            }

            // 不是第一列的话，向左合并，1和3

            if (j != 0) {

                MergeElemetByRank(parent, rank, temp + 1, temp - 4 + 3);

            }

            // 不是最后一列的话，向右合并

            if (j != gridSize - 1) {

                MergeElemetByRank(parent, rank, temp + 3, temp + 4 + 1);

            }

        }

    }

    for (i = 0; i < gridSize \* gridSize \* 4; i++) {

            if (parent[i] == i) {

                ans++;

            }

    }

    free(parent);

    free(rank);

    return ans;

}

int main()

{

    char\* tmpStr[] = {" /", "/ "};

    int i;

    int ret = 0;

    char\*\* grid = (char\*\*)calloc(sizeof(tmpStr) / sizeof(char\*), sizeof(char\*));

    for (i = 0; i < sizeof(tmpStr) / sizeof(char\*); i++) {

        grid[i] = (char\*)calloc(strlen(tmpStr) + 1, sizeof(char));

        memcpy(grid[i], tmpStr[i], strlen(tmpStr) + 1);

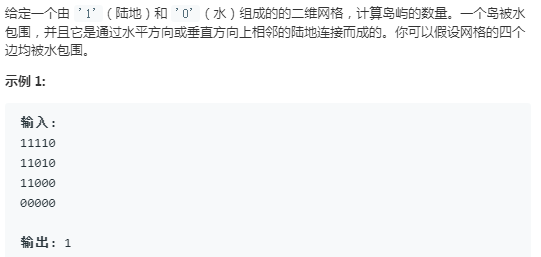
    }

    ret = regionsBySlashes(grid, 2);

    printf("ret %d\n", ret);

}

## 4.[200. 岛屿数量](https://leetcode-cn.com/problems/number-of-islands/)【并查集+判定多少个集合】



思路:

将二维数组展开成一维的，即I \* colSize + j。

遍历二维元素，非0的元素去尝试和4个方向的元素进行和并，能合并说明是连通的。

遍历完成就可以统计parent[i] == I的个数，即为连通的岛屿数

代码：

int dir[4][2] = {

    {0, -1},

    {0, 1},

    {1, 0,},

    {-1, 0},

};

int FindRootCompress(int\* parent, int ele)

{

    if (ele == parent[ele]) {

        return ele;

    }

    parent[ele] = FindRootCompress(parent, parent[ele]);

    return parent[ele];

}

int MergeElementByrank(int\* parent, int\* rank, int x, int y)

{

    int xRoot, yRoot;

    xRoot = FindRootCompress(parent, x);

    yRoot = FindRootCompress(parent, y);

    if (xRoot == yRoot) {

        return 0;

    }

    if (rank[xRoot] > rank[yRoot]) {

        parent[yRoot] = xRoot;

    }

    if (rank[yRoot] > rank[xRoot]) {

        parent[xRoot] = yRoot;

    }

    if (rank[yRoot] == rank[xRoot]) {

        parent[yRoot] = xRoot;

        rank[xRoot]++;

    }

    return 1;

}

int numIslands(char\*\* grid, int gridSize, int\* gridColSize)

{

    int row = gridSize;

    int col = gridColSize[0];

    int i, j, k, root;

    int x, y;

    int ans = 0;

    int\* parent = NULL;

    int\* rank = NULL;

    if (grid == NULL || gridSize == 0 || gridColSize[0] == 0) {

        return 0;

    }

    parent = (int\*)calloc(row \* col, sizeof(int));

    rank = (int\*)calloc(row \* col, sizeof(int));

    for (i = 0 ;i < row \* col; i++) {

        parent[i] = i;

        rank[i] = 1;

    }

    for (i = 0; i < row; i++) {

        for (j = 0; j < col; j++) {

            // 如果是0的话，将并查集对应节点的父节点置为-1

            if (grid[i][j] == '0') {

                parent[i \* col + j] = -1;

                continue;

            }

            // 如果是1的话，需要对周围进行尝试

            for (k = 0; k < 4; k++) {

                x = i + dir[k][0];

                y = j + dir[k][1];

                if (x < 0 || y < 0 || x >= row || y >= col || grid[x][y] == '0') {

                    continue;

                }

                MergeElementByrank(parent, rank, i \* col + j, x \*col + y);

            }

        }

    }

    for (i = 0; i < row \* col; i++) {

        if (parent[i] == i) {

            ans++;

        }

    }

    free(parent);

    free(rank);

    return ans;

}

int main()

{

    char nums[][20] = {

        /\*{'1','1','1','1','0'},

        {'1','1','0','1','0'},

        {'1','1','0','0','0'},

        {'0','0','0','0','0'},\*/

        /\*{'1','1','0','0','0'},

        {'1','1','0','0','0'},

        {'0','0','1','0','0'},

        {'0','0','0','1','1'},\*/

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    {'0','0','0','0','0','0','1','1','1','1','0','1','0','0','0','1','1','0','0','0'},//58

    };

    char\*\* grid = (char\*\*)calloc(sizeof(nums) / sizeof(nums[0]), sizeof(char\*));

    int i, j;

    int ret = 0;

    int colSize = sizeof(nums[i]) / sizeof(nums[i][0]);

    for (i = 0; i < sizeof(nums) / sizeof(nums[0]); i++) {

        grid[i] = (char\*)calloc(grid[i], sizeof(nums[i]) / sizeof(nums[i][0]) \* sizeof(char));

        memcpy(grid[i], nums[i], sizeof(nums[i]) / sizeof(nums[i][0]) \* sizeof(char));

    }

    for (i = 0; i < sizeof(nums) / sizeof(nums[0]); i++) {

        for (j = 0; j < colSize; j++) {

            printf("%c ", nums[i][j]);

        }

        printf("\n");

    }

    ret = numIslands(grid, sizeof(nums) / sizeof(nums[0]), &colSize);

    printf("ret %d\n", ret);

    return 0;

}

## 5. [130. 被围绕的区域](https://leetcode-cn.com/problems/surrounded-regions/)【并查集+设置多余的集合判定属于这个集合】



思路：和第四题题目其实有点相反。本题可以将边界点归为一个集合，可以多设置一个数组元素，遇到边界就和整个多余的点结合。其他的点如果是‘O’,则访问周围的四个点，并和周围四个点钟的‘O’集合。

最后遍历所有元素，如果和多余的点为同一集合的肯定是和边界的‘O’相连或者就是边界的 ‘o’.

int dir[][2] = {

    {0,-1},

    {0,1},

    {1,0},

    {-1,0},

};

int FindRootCompress(int\* parent, int ele)

{

    if (parent[ele] == ele) {

        return ele;

    }

    parent[ele] = FindRootCompress(parent, parent[ele]);

    return parent[ele];

}

int MergeElementByRank(int\* parent, int\* rank, int x, int y)

{

    int xRoot, yRoot;

    xRoot = FindRootCompress(parent, x);

    yRoot = FindRootCompress(parent, y);

    if (xRoot == yRoot) {

        return 0;

    }

    if (rank[xRoot] > rank[yRoot]) {

        parent[yRoot] = xRoot;

    }

    if (rank[yRoot] > rank[xRoot]) {

        parent[xRoot] = yRoot;

    }

    if (rank[yRoot] == rank[xRoot]) {

        rank[xRoot]++;

        parent[yRoot] = xRoot;

    }

    return 1;

}

void solve(char\*\* board, int boardSize, int\* boardColSize){

    int row = boardSize;

    int col = boardColSize[0];

    int i ,j, k;

    int root1, root2, nextX, nextY;

    int\* parent = NULL;

    int\* rank = NULL;

    int max = row \* col + 1;

    if (boardSize == 0 || boardColSize == NULL) {

        return;

    }

    parent = (int\*)calloc(max, sizeof(int));

    rank = (int\*)calloc(max, sizeof(int));

    for (i = 0; i < max; i++) {

        parent[i] = i;

        rank[i] = 1;

    }

    for (i = 0; i < row; i++) {

        for (j = 0; j < col; j++) {

            if (board[i][j] == 'X') {

                continue;

            }

            // 如果是‘O’,先判断是否是边界，边界的话，先归档

            if (i == 0 || i == row - 1 || j == 0 || j == col - 1) {

                MergeElementByRank(parent, rank, max - 1, i \* col + j);

            }

            // 再和周围的'O'结合

           for (k = 0; k < 4; k++) {

            nextX = i + dir[k][0];

            nextY = j + dir[k][1];

            if (nextX < 0 || nextY < 0 || nextX >= row || nextY >= col) {

                continue;

            }

            if (board[nextX][nextY] == 'O') {

                MergeElementByRank(parent, rank, i \* col + j, nextX \* col + nextY);

            }

           }

        }

    }

    root1 = FindRootCompress(parent, max - 1);

    for (i = 0; i < row; i++) {

        for (j = 0; j < col; j++) {

            if (board[i][j] == 'X') {

                continue;

            }

            root2 = FindRootCompress(parent, i \* col + j);

            if (root1 != root2) {

                board[i][j] = 'X';

            }

        }

    }

    free(parent);

    free(rank);

    parent = NULL;

    rank = NULL;

}

int main()

{

    int i ,j;

    char tmp[][6] = {

    {'X','X','X','X','O','X'},

    {'O','X','X','O','O','X'},

    {'X','O','X','O','O','O'},

    {'X','O','O','O','X','O'},

    {'O','O','X','X','O','X'},

    {'X','O','X','O','X','X'},

};

    char\*\* board = NULL;

    int colSize = sizeof(tmp[0]) / sizeof(tmp[0][0]);

    board = (char\*)calloc(sizeof(tmp) / sizeof(tmp[0]), sizeof(char));

    for (i = 0; i < sizeof(tmp) / sizeof(tmp[0]); i++) {

        board[i] = (char\*)calloc(sizeof(tmp[0]) / sizeof(tmp[0][0]) \* sizeof(char), sizeof(char));

        memcpy(board[i], tmp[i], sizeof(tmp[0]) / sizeof(tmp[0][0]) \* sizeof(char));

    }

    solve(board, sizeof(tmp) / sizeof(tmp[0]), &colSize);

    for (i = 0 ; i < sizeof(tmp) / sizeof(tmp[0]); i++) {

        for (j = 0 ; j < sizeof(tmp[0]) / sizeof(tmp[0][0]); j++) {

            printf("%c ",board[i][j]);

        }

        printf("\n");

    }

}

## 6. [947. 移除最多的同行或同列石头](https://leetcode-cn.com/problems/most-stones-removed-with-same-row-or-column/)【并查集，对输入处理判定归入的点很重要】



题意：其实每次移除一个石头的条件是本行或者列需要至少2个石头。比如输入[[0,0],[0,1],[1,0],[1,2],[2,1],[2,2]]。其实对应的矩阵是

1 1 0

1 0 1

0 1 1。

因此是5次。

仔细观察其实不难发新，最多的次数其实就是石头的个数（stoneSize） - 集合个数。

同行同列都加到集合中去可知这个输入其实是一个集合，因此6-1.

思路：将同行同列的石头归入一个集合，归入的“点”就是stone的石头序号。初始集合的总数

为stoneSize.

一开始自己将问题复杂化了写成先对行合并再对列合并，合并的点是i\*colSize+j这样会导致超时

下面是超时代码：

int MyCmp0(const void\* a, const void\* b)

{

int\*\* pa = (int\*\*)a;

int\*\* pb = (int\*\*)b;

return (\*pa)[0] - (\*pb)[0];

}

int MyCmp1(const void\* a, const void\* b)

{

int\*\* pa = (int\*\*)a;

int\*\* pb = (int\*\*)b;

return (\*pa)[1] - (\*pb)[1];

}

int FindRootCompress(int\* parent, int ele)

{

if (parent[ele] == ele) {

return ele;

}

parent[ele] = FindRootCompress(parent, parent[ele]);

return parent[ele];

}

int MergeElementByRank(int\* parent, int\* rank, int x, int y)

{

int xRoot, yRoot;

xRoot = FindRootCompress(parent, x);

yRoot = FindRootCompress(parent, y);

if (xRoot == yRoot) {

return 0;

}

if (rank[xRoot] > rank[yRoot]) {

parent[yRoot] = xRoot;

}

if (rank[xRoot] < rank[yRoot]) {

parent[xRoot] = yRoot;

}

if (rank[xRoot] == rank[yRoot]) {

parent[yRoot] = xRoot;

rank[xRoot]++;

}

return 1;

}

int removeStones(int\*\* stones, int stonesSize, int\* stonesColSize){

int\* parent = NULL;

int\* rank = NULL;

int rowMax = -1;

int colMax = -1;

int size;

int count = 0;

int i, j;

qsort(stones, stonesSize, sizeof(int\*), MyCmp0);

for (i = 0; i < stonesSize; i++) {

colMax = (colMax < stones[i][1]) ? stones[i][1] : colMax;

}

rowMax = stones[stonesSize - 1][0] + 1;

colMax = colMax + 1;

size = (rowMax) \* (colMax);

parent = (int\*)calloc(size, sizeof(int));

rank = (int\*)calloc(size, sizeof(int));

for (i = 0; i < size; i++) {

parent[i] = i;

rank[i] = 1;

}

// 合并同行元素

for (i = 0; i < stonesSize; i = j) {

// 对该个元素合并同行的元素，某行的后续元素则不需要合并同行

for (j = i + 1; j < stonesSize; j++) {

// 第一个元素不同说明不在同一行

if (stones[j][0] != stones[i][0]) {

break;

}

MergeElementByRank(parent, rank, stones[i][0] \* colMax + stones[i][1], stones[j][0] \* colMax + stones[j][1]);

}

}

qsort(stones, stonesSize, sizeof(int\*), MyCmp1);

// 合并同列元素

j = 0;

for (i = 0; i < stonesSize; i = j) {

// 对该列元素合并同列的元素，某列的后续元素则不需要合并

for (j = i + 1; j < stonesSize; j++) {

// 第一个元素不同说明不在同一行

if (stones[j][1] != stones[i][1]) {

break;

}

MergeElementByRank(parent, rank, stones[i][0] \* colMax + stones[i][1], stones[j][0] \* colMax + stones[j][1]);

}

}

for (i = 0; i < size; i++) {

if (parent[i] == i) {

count++;

}

}

free(parent);

free(rank);

// 可移除的次数为 石头数 - 集合数 （因为假设10个石头，集合1中4个， 集合2中1个，其实只能移////走 3个；另一层面

//是10 - 5 -集合1 - 集合2 = 3）

return size - count;

}

下面是AC代码：

int FindRootCompress(int\* parent, int ele)

{

    if (parent[ele] == ele) {

        return ele;

    }

    parent[ele] = FindRootCompress(parent, parent[ele]);

    return parent[ele];

}

int MergeElementByRank(int\* parent, int\* rank, int x, int y)

{

    int xRoot, yRoot;

    xRoot = FindRootCompress(parent, x);

    yRoot = FindRootCompress(parent, y);

    if (xRoot == yRoot) {

        return 0;

    }

    if (rank[xRoot] > rank[yRoot]) {

        parent[yRoot] = xRoot;

    }

    if (rank[xRoot] < rank[yRoot]) {

        parent[xRoot] = yRoot;

    }

    if (rank[xRoot] == rank[yRoot]) {

        parent[yRoot] = xRoot;

        rank[xRoot]++;

    }

    return 1;

}

int removeStones(int\*\* stones, int stonesSize, int\* stonesColSize){

   int\* parent = NULL;

   int\* rank = NULL;

   int count = 0;

   int i, j;

   int xTemp, yTemp;

    parent = (int\*)calloc(stonesSize, sizeof(int));

    rank = (int\*)calloc(stonesSize, sizeof(int));

    for (i = 0; i < stonesSize; i++) {

            parent[i] = i;

            rank[i] = 1;

    }

    // 合并同行元素同列的石头序号

    for (i = 0; i < stonesSize; i++) {

        for (j = i; j < stonesSize; j++) {

            if (stones[i][0] == stones[j][0] || stones[i][1] == stones[j][1]) {

                MergeElementByRank(parent, rank, i, j);

            }

        }

    }

    for (i = 0; i < stonesSize; i++) {

        if (parent[i] == i) {

            count++;

        }

    }

    free(parent);

    free(rank);

    // 可移除的次数为 石头数 - 集合数    （因为假设10个石头，集合1中4个， 集合2中1个，其实只能移////走 3个；另一层面

    //是10 - 5 -集合1 - 集合2 = 3）

    return stonesSize - count;

}