**17. 电话号码的字母组合**

给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。

给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。



输入："23"

输出：["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

|  |
| --- |
| class Solution {  public:  vector<string> letterCombinations(string digits) {  if (digits.empty()) {  return {};  }  recursive(digits, "", 0);  return res\_;  }  void recursive(const string& digits, string str, int pos) {  if (pos == digits.size()) {  res\_.push\_back(str);  return;  }  char digit = digits[pos];  for (char c : hash\_[digit]) {  recursive(digits, str + c, pos +1);  }  }  vector<string> res\_;  map<char, vector<char>> hash\_{  {'2', {'a', 'b', 'c'}},  {'3', {'d', 'e', 'f'}},  {'4', {'g', 'h', 'i'}},  {'5', {'j', 'k', 'l'}},  {'6', {'m', 'n', 'o'}},  {'7', {'p', 'q', 'r', 's'}},  {'8', {'t', 'u', 'v'}},  {'9', {'w', 'x', 'y', 'z'}}  };  };// 暴力搜索所有可能的电话号码组合。 |

**22. 括号生成**

数字 n 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且 有效的 括号组合。

输入：n = 3

输出：[

"((()))",

"(()())",

"(())()",

"()(())",

"()()()"

]

|  |
| --- |
| class Solution {  public:  vector<string> generateParenthesis(int n) {  string path;  n\_ = n;  recursive(0, 0, path);  return res\_;  }  void recursive(int left, int right, string path) {  if (left == n\_ && right == n\_) {  res\_.push\_back(path);  return;  }  if (left < right) {  return;  }  if (left > n\_ || right > n\_) {  return;  }  recursive(left + 1, right, path + "(");  recursive(left, right + 1, path + ")");  }  int n\_ = 0;  vector<string> res\_;  }; |

**37. 解数独**

编写一个程序，通过填充空格来解决数独问题。

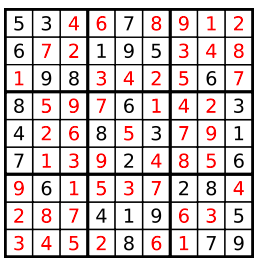
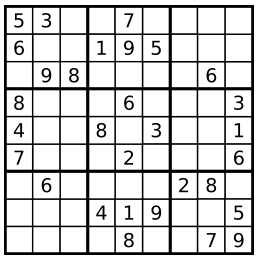
一个数独的解法需遵循如下规则：

数字 1-9 在每一行只能出现一次。

数字 1-9 在每一列只能出现一次。

数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。

空白格用 '.' 表示。



给定的数独序列只包含数字 1-9 和字符 '.' 。

你可以假设给定的数独只有唯一解。

给定数独永远是 9x9 形式的。

|  |
| --- |
| class Solution {  public:  void solveSudoku(vector<vector<char>>& board)  {  Dfs(board, 0, 0);  }  bool Dfs(vector<vector<char>>& board, int x, int y)  {  if (x >= board[0].size()) return Dfs(board, 0, y + 1);//一行找完了，找下一行  if (y >= board.size()) return true; //行列都找完了，结束  if (board[y][x] != '.') return Dfs(board, x + 1, y);//有数字，找下一个格子  for (char num : nums\_) {//依次遍历可放入的元素  if (check(board, x, y, num)) {//如果可选  board[y][x] = num;//填入  if (Dfs(board, x + 1, y)) {//下一个可填，最后一个格子依次返回的都是true  return true;  } else {//回溯  board[y][x] = '.';  }  }  }  return false;  }  bool check(vector<vector<char>>& board, int x, int y, char c)  { //判断当前元素是否可放（x,y）位置  for (int i = 0; i<board.size(); i++) {//横竖  if (board[y][i] == c || board[i][x] == c) return false;  }  for (int i = y / 3 \* 3; i<y / 3 \* 3 + 3; i++) {//找到九宫格左上元素，依次遍历  for (int j = x / 3 \* 3; j<x / 3 \* 3 + 3; j++) {  if (board[i][j] == c) return false;  }  }  return true;  }  vector<char> nums\_{ '1','2','3','4','5','6','7','8','9' };  }; |

**51. N 皇后**

n 皇后问题研究的是如何将 n 个皇后放置在 n×n 的棋盘上，并且使皇后彼此之间不能相互攻击。



上图为 8 皇后问题的一种解法。

给定一个整数 n，返回所有不同的 n 皇后问题的解决方案。

每一种解法包含一个明确的 n 皇后问题的棋子放置方案，该方案中 'Q' 和 '.' 分别代表了皇后和空位。

输入：4

输出：[

[".Q..", // 解法 1

"...Q",

"Q...",

"..Q."],

["..Q.", // 解法 2

"Q...",

"...Q",

".Q.."]

]

解释: 4 皇后问题存在两个不同的解法。

|  |
| --- |
| class Solution {  public:  vector<vector<string>> ans;  void func(vector<string>& res, int row){  if(row == res.size()){  ans.push\_back(res);  return;  }  for(int j = 0; j < res[0].size(); j++){  if(!isValid(res, row, j)) continue;  res[row][j] = 'Q';  func(res, row+1);  res[row][j] = '.';  }  }  bool isValid(vector<string>& res, int row, int col){  for(int i = 0; i < row; i++){  if(res[i][col] == 'Q') return false;  }  for(int i = row-1, j = col-1; j >= 0 && i >= 0;j--, i--){  if(res[i][j] == 'Q') return false;  }  for(int i = row-1, j = col+1; j < res[0].size(), i >= 0; i--, j++){  if(res[i][j] == 'Q') return false;  }  return true;  }  vector<vector<string>> solveNQueens(int n) {  vector<string> res(n, string(n, '.'));  func(res, 0);  return ans;  }  }; |

**52. N皇后 II**

n 皇后问题研究的是如何将 n 个皇后放置在 n×n 的棋盘上，并且使皇后彼此之间不能相互攻击。

给定一个整数 n，返回 n 皇后不同的解决方案的数量。

输入: 4

输出: 2

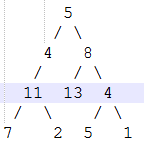
|  |
| --- |
| class Solution {  int ans\_;  public:  int totalNQueens(int n) {  vector<string> res(n, string(n, '.'));  func(res, 0);  return ans\_;  }  void func(vector<string>& res, int row){  if(row == res.size()){  ans\_++;  return;  }  for(int j = 0; j < res[0].size(); j++){  if(!isValid(res, row, j)) continue;  res[row][j] = 'Q';  func(res, row+1);  res[row][j] = '.';  }  }  bool isValid(vector<string>& res, int row, int col){  for(int i = 0; i < row; i++){  if(res[i][col] == 'Q') return false;  }  for(int i = row-1, j = col-1; j >= 0 && i >= 0;j--, i--){  if(res[i][j] == 'Q') return false;  }  for(int i = row-1, j = col+1; j < res[0].size(), i >= 0; i--, j++){  if(res[i][j] == 'Q') return false;  }  return true;  }  }; |

**113. 路径总和 II**

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。

说明: 叶子节点是指没有子节点的节点。

给定如下二叉树，以及目标和 sum = 22，



返回:

[

[5,4,11,2],

[5,8,4,5]

]

|  |
| --- |
| class Solution {  public:  vector<vector<int>> pathSum(TreeNode\* root, int sum) {  if (root == nullptr) {  return {};  }  Dfs(root, sum, vector<int>());  return result\_;  }  void Dfs(TreeNode\* root, int sum, vector<int> path) {  if (root == nullptr) {  return;  }  if (sum == root->val && root->left == nullptr && root->right == nullptr) {  path.push\_back(root->val);  result\_.push\_back(path);  return;  }  path.push\_back(root->val);  Dfs(root->left, sum - root->val, path);  Dfs(root->right, sum - root->val, path);  }  vector<vector<int>> result\_;  }; |

**133. 克隆图**

给你无向 连通 图中一个节点的引用，请你返回该图的 深拷贝（克隆）。

图中的每个节点都包含它的值 val（int） 和其邻居的列表

class Node {

public:

int val;

vector<Node\*> neighbors;

Node() {

val = 0;

neighbors = vector<Node\*>();

}

Node(int \_val) {

val = \_val;

neighbors = vector<Node\*>();

}

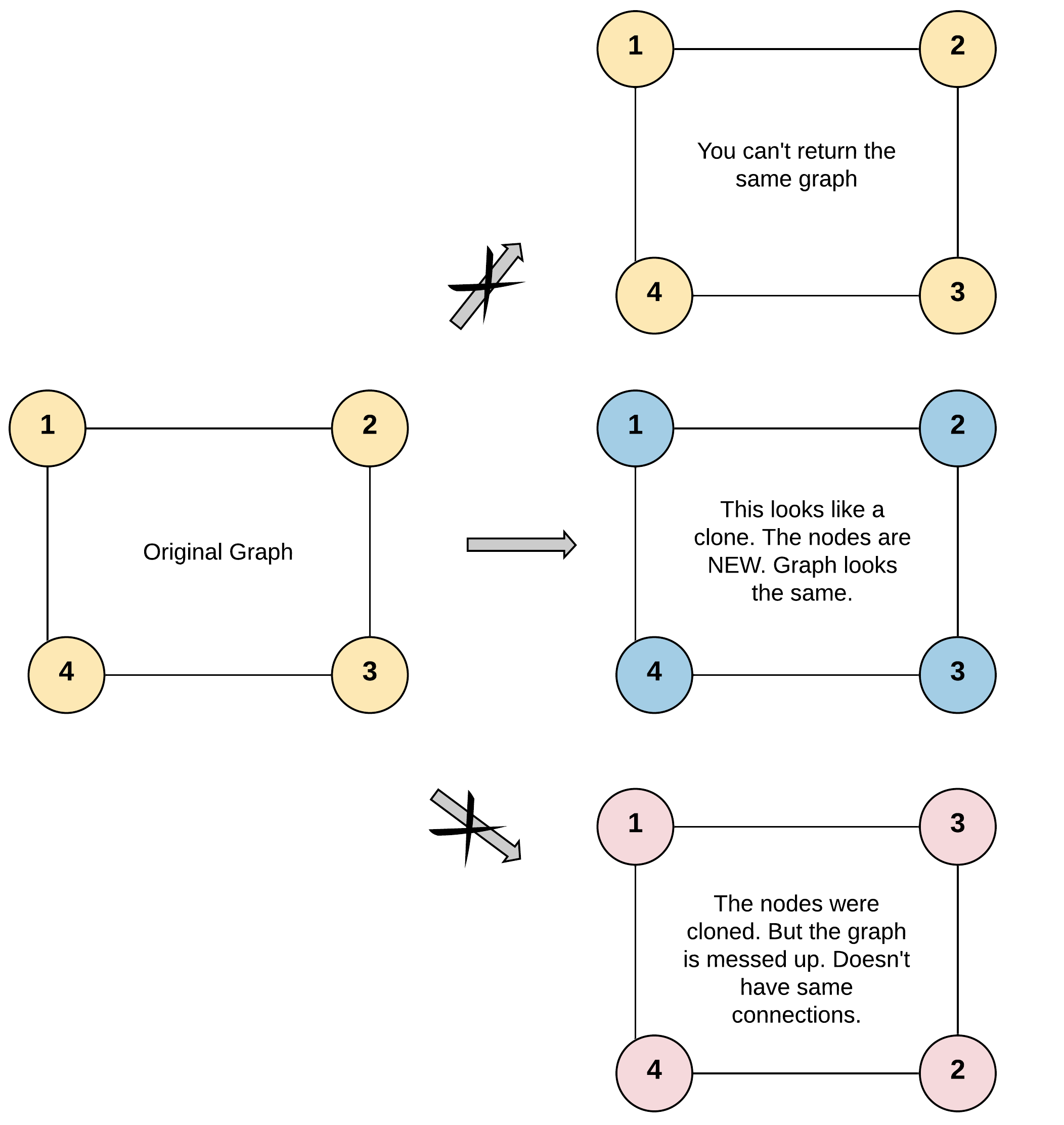
Node(int \_val, vector<Node\*> \_neighbors) {

val = \_val;

neighbors = \_neighbors;

}

};



输入：adjList = [[2,4],[1,3],[2,4],[1,3]]

输出：[[2,4],[1,3],[2,4],[1,3]]

|  |
| --- |
| class Solution {  public:  Node\* cloneGraph(Node\* node) {  if (node == nullptr) {  return nullptr;  }  auto it = visited\_.find(node);  if (it != visited\_.end()) {  return it->second;  }  Node\* newNode = new Node(node->val);  visited\_[node] = newNode;  for (auto item : node->neighbors) {  newNode->neighbors.push\_back(cloneGraph(item));  }  return newNode;  }  map<Node\*, Node\*> visited\_;  };//递归拷贝。如果已经拷贝过就直接设置指针，没有拷贝过就去拷贝。 |

**200. 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。

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

输入：grid = [

["1","1","0","0","0"],

["1","1","0","0","0"],

["0","0","1","0","0"],

["0","0","0","1","1"]

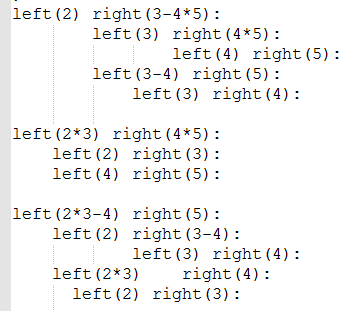
]

输出：3

|  |
| --- |
| class Solution {  public:  int numIslands(vector<vector<char>>& grid) {  int islandNum = 0;  for(int i = 0; i < grid.size(); i++){  for(int j = 0; j < grid[0].size(); j++){  if(grid[i][j] == '1'){  infect(grid, i, j);  islandNum++;  }  }  }  return islandNum;  }  //感染函数  void infect(vector<vector<char>>& grid, int i, int j){  if(i < 0 || i >= grid.size() ||  j < 0 || j >= grid[0].size() || grid[i][j] != '1'){  return;  }  grid[i][j] = '2';  infect(grid, i + 1, j);  infect(grid, i - 1, j);  infect(grid, i, j + 1);  infect(grid, i, j - 1);  }  }; |

**241. 为运算表达式设计优先级**

给定一个含有数字和运算符的字符串，为表达式添加括号，改变其运算优先级以求出不同的结果。你需要给出所有可能的组合的结果。有效的运算符号包含 +, - 以及 \* 。

输入: "2-1-1"

输出: [0, 2]

解释:

((2-1)-1) = 0

(2-(1-1)) = 2

输入: "2\*3-4\*5"

输出: [-34, -14, -10, -10, 10]

解释:

(2\*(3-(4\*5))) = -34

((2\*3)-(4\*5)) = -14

((2\*(3-4))\*5) = -10

(2\*((3-4)\*5)) = -10

(((2\*3)-4)\*5) = 10

|  |
| --- |
| class Solution {  public:  vector<int> diffWaysToCompute(string input)  {  if (input.empty()) {  return {};  }  vector<int> res;  for (size\_t i = 0; i < input.size(); i++) {  char c = input[i];  if (c == '+' || c == '-' || c == '\*') {  auto lefts = diffWaysToCompute(input.substr(0, i));  auto rights = diffWaysToCompute(input.substr(i + 1));  for (auto l : lefts) {  for (auto r : rights) {  if (c == '+') {  res.push\_back(l + r);  } else if (c == '-') {  res.push\_back(l - r);  } else {  res.push\_back(l \* r);  }  }  }  }  }  if (res.empty()) {  return { stoi(input) };  }  return res;  }  }; |

**417. 太平洋大西洋水流问题**

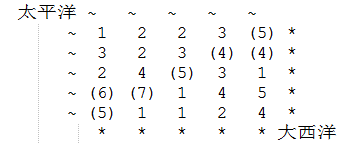
给定一个 m x n 的非负整数矩阵来表示一片大陆上各个单元格的高度。“太平洋”处于大陆的左边界和上边界，而“大西洋”处于大陆的右边界和下边界。

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。

提示：输出坐标的顺序不重要，m 和 n 都小于150

给定下面的 5x5 矩阵:



返回:

[[0, 4], [1, 3], [1, 4], [2, 2], [3, 0], [3, 1], [4, 0]] (上图中带括号的单元).

|  |
| --- |
| class Solution {  public:  vector<vector<int>> pacificAtlantic(vector<vector<int>>& matrix)  {  if (matrix.empty() || matrix[0].empty()) {  return {};  }  n\_ = matrix.size();  m\_ = matrix[0].size();  set<vector<int>> pacificPos;  for (size\_t x = 0; x < m\_; x++) {  Dfs(0, x, matrix, pacificPos, matrix[0][x]);  }  for (size\_t y = 0; y < n\_; y++) {  Dfs(y, 0, matrix, pacificPos, matrix[y][0]);  }  set<vector<int>> atlanticPos;  for (size\_t x = 0; x < m\_; x++) {  Dfs(n\_ - 1, x, matrix, atlanticPos, matrix[n\_ - 1][x]);  }  for (size\_t y = 0; y < n\_; y++) {  Dfs(y, m\_ - 1, matrix, atlanticPos, matrix[y][m\_ - 1]);  }    set<vector<int>> insection;  set\_intersection(pacificPos.begin(), pacificPos.end(),  atlanticPos.begin(), atlanticPos.end(), inserter(insection, insection.begin()));    return vector<vector<int>>(insection.begin(), insection.end());  }  private:  void Dfs(int y, int x, vector<vector<int>>& matrix, set<vector<int>>& visited, int pre)  {  if (!IsValidPos(y, x) || visited.find({y, x}) != visited.end() || matrix[y][x] < pre) {  return;  }  visited.insert({ y, x });  int cur = matrix[y][x];  Dfs(y, x + 1, matrix, visited, cur);  Dfs(y, x - 1, matrix, visited, cur);  Dfs(y + 1, x, matrix, visited, cur);  Dfs(y - 1, x, matrix, visited, cur);  }  bool IsValidPos(int y, int x)  {  return (y >= 0) && (y < n\_) && (x >= 0) && (x < m\_);  }  int n\_{ 0 };  int m\_{ 0 };  };// 逆向考虑，求太平洋和大西洋都水能到达的地方 |

529. 扫雷游戏

给定一个代表游戏板的二维字符矩阵。 'M' 代表一个未挖出的地雷，'E' 代表一个未挖出的空方块，'B' 代表没有相邻（上，下，左，右，和所有4个对角线）地雷的已挖出的空白方块，数字（'1' 到 '8'）表示有多少地雷与这块已挖出的方块相邻，'X' 则表示一个已挖出的地雷。

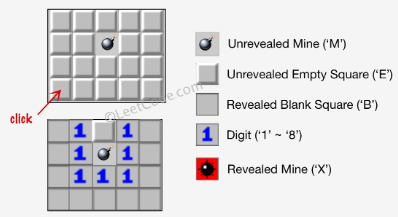
现在给出在所有未挖出的方块中（'M'或者'E'）的下一个点击位置（行和列索引），根据以下规则，返回相应位置被点击后对应的面板：

如果一个地雷（'M'）被挖出，游戏就结束了- 把它改为 'X'。

如果一个没有相邻地雷的空方块（'E'）被挖出，修改它为（'B'），并且所有和其相邻的未挖出方块都应该被递归地揭露。

如果一个至少与一个地雷相邻的空方块（'E'）被挖出，修改它为数字（'1'到'8'），表示相邻地雷的数量。

如果在此次点击中，若无更多方块可被揭露，则返回面板。



输入:

[['E', 'E', 'E', 'E', 'E'],

['E', 'E', 'M', 'E', 'E'],

['E', 'E', 'E', 'E', 'E'],

['E', 'E', 'E', 'E', 'E']]

Click : [3,0]

输出:

[['B', '1', 'E', '1', 'B'],

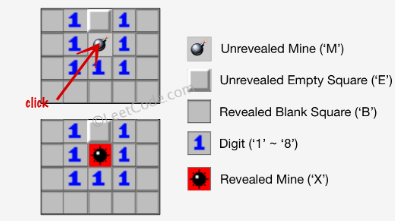
['B', '1', 'M', '1', 'B'],

['B', '1', '1', '1', 'B'],

['B', 'B', 'B', 'B', 'B']]

解释:->

输入:

[['B', '1', 'E', '1', 'B'],

['B', '1', 'M', '1', 'B'],

['B', '1', '1', '1', 'B'],

['B', 'B', 'B', 'B', 'B']]

Click : [1,2]

输出:

[['B', '1', 'E', '1', 'B'],

['B', '1', 'X', '1', 'B'],

['B', '1', '1', '1', 'B'],

['B', 'B', 'B', 'B', 'B']]

注意：

1. 输入矩阵的宽和高的范围为 [1,50]。

2. 点击的位置只能是未被挖出的方块 ('M' 或者 'E')，这也意味着面板至少包含一个可点击的方块。

3. 输入面板不会是游戏结束的状态（即有地雷已被挖出）。

4. 简单起见，未提及的规则在这个问题中可被忽略。例如，当游戏结束时你不需要挖出所有地雷，考虑所有你可能赢得游戏或标记方块的情况。

|  |
| --- |
| class Solution {  public:  vector<vector<char>> updateBoard(vector<vector<char>>& board, vector<int>& click)  {  if (board.empty() || board[0].empty() || click.size() < 2) {  return board;  }  n\_ = board.size();  m\_ = board[0].size();  int startY = click[0];  int startX = click[1];  if (startY >= n\_ || startX >= m\_) {  return board;  }  if (board[startY][startX] == 'M') {  board[startY][startX] = 'X';  return board;  }  TraceBack(startY, startX, board);  return board;  }  void TraceBack(int y, int x, vector<vector<char>>& board)  {  if (!ValidPos(y, x)) {  return;  }  if (board[y][x] != 'E') {  return;  }  int sum = 0;  for (const auto item : orient\_) {  int newY = y + item[0];  int newX = x + item[1];  if (!ValidPos(newY, newX)) {  continue;  }  if (board[newY][newX] == 'M') {  sum++;  }  }  if (sum == 0) {  board[y][x] = 'B';  for (const auto item : orient\_) {  int newY = y + item[0];  int newX = x + item[1];  TraceBack(newY, newX, board);  }  } else {  board[y][x] = '0' + sum;  }  }  bool ValidPos(int y, int x)  {  return y >= 0 && y < n\_ && x >= 0 && x < m\_;  }  int n\_{ 0 };  int m\_{ 0 };  vector<vector<int>> orient\_{ {-1, 0}, {1, 0}, {0, -1}, {0, 1},  {1, 1}, {1, -1}, {-1, -1}, {-1, 1} };  }; |

**698. 划分为k个相等的子集**

给定一个整数数组 nums 和一个正整数 k，找出是否有可能把这个数组分成 k 个非空子集，其总和都相等。

输入： nums = [4, 3, 2, 3, 5, 2, 1], k = 4

输出： True

说明： 有可能将其分成 4 个子集（5），（1,4），（2,3），（2,3）等于总和。

|  |
| --- |
| class Solution {  public:  bool canPartitionKSubsets(vector<int>& nums, int k)  {  int n = nums.size();  if (n < k) {  return false;  }  int maxVal = \*max\_element(nums.begin(), nums.end());  int sum = accumulate(nums.begin(), nums.end(), 0);  int average = sum / k;  if (sum % k != 0 || maxVal > average) {  return false;  }  average\_ = average;  k\_ = k;  n\_ = n;  sort(nums.begin(), nums.end(), greater<int>());  visited\_ = vector<bool>(n\_, false);  return Dfs(0, 0, nums);  }  bool Dfs(int sum, int times, vector<int>& nums) {  if (sum == average\_) {  times++;  sum = 0;  }  if (sum == 0 && times == k\_) {  return true;  }  for (size\_t i = 0; i < n\_; i++) {  if (sum + nums[i] > average\_) {  continue;  }  if (visited\_[i] == true) {  continue;  }  if (i > 0 && nums[i - 1] == nums[i] && visited\_[i - 1] == false) {  continue;  }  visited\_[i] = true;  bool ret = Dfs(sum + nums[i], times, nums);  if (ret) {  return true;  }  visited\_[i] = false;  }  return false;  }  int average\_{ 0 };  vector<bool> visited\_;  int k\_{ 0 };  int n\_{ 0 };  }; |

**785. 判断二分图**

给定一个无向图graph，当这个图为二分图时返回true。

如果我们能将一个图的节点集合分割成两个独立的子集A和B，并使图中的每一条边的两个节点一个来自A集合，一个来自B集合，我们就将这个图称为二分图。

graph将会以邻接表方式给出，graph[i]表示图中与节点i相连的所有节点。每个节点都是一个在0到graph.length-1之间的整数。这图中没有自环和平行边： graph[i] 中不存在i，并且graph[i]中没有重复的值。

输入: [[1,3], [0,2], [1,3], [0,2]]

输出: true

解释:无向图如下: 我们可以将节点分成两组: {0, 2} 和 {1, 3}。

0----1

| |

| |

3----2

输入: [[1,2,3], [0,2], [0,1,3], [0,2]]

输出: false

解释: 无向图如下:我们不能将节点分割成两个独立的子集。

0----1

| \ |

| \ |

3----2

|  |
| --- |
| class Solution {  public:  bool isBipartite(vector<vector<int>>& graph) {  const int n = graph.size();  vector<int> v(n, -1); //-1表示待染色  for (int i = 0; i < n; i++) if (v[i] == -1 && !dfs(graph, i, 0, v)) return false;  return true;  }  private: //返回第i个点染c色能否成功  bool dfs(const vector<vector<int>> &g, int i, int c, vector<int> &v) {  if (v[i] != -1) return v[i] == c; //第i个点已染过  v[i] = c; //对第i个点染上c色  for (int j : g[i]) if (!dfs(g, j, !c, v)) return false; //递归相邻的点，c => !c 换色  return true;  }  };// 二分图判断。给每个节点染色，并且检查它的邻居和它的颜色是否相等，如果存在相等说明不是二分图。 |

**851. 喧闹和富有**

在一组 N 个人（编号为 0, 1, 2, ..., N-1）中，每个人都有不同数目的钱，以及不同程度的安静（quietness）。

为了方便起见，我们将编号为 x 的人简称为 "person x "。

如果能够肯定 person x 比 person y 更有钱的话，我们会说 richer[i] = [x, y] 。注意 richer 可能只是有效观察的一个子集。

另外，如果 person x 的安静程度为 q ，我们会说 quiet[x] = q 。

现在，返回答案 answer ，其中 answer[x] = y 的前提是，在所有拥有的钱不少于 person x 的人中，person y 是最安静的人（也就是安静值 quiet[y] 最小的人）。

输入：richer = [[1,0],[2,1],[3,1],[3,7],[4,3],[5,3],[6,3]], quiet = [3,2,5,4,6,1,7,0]

输出：[5,5,2,5,4,5,6,7]

解释：

answer[0] = 5，

person 5 比 person 3 有更多的钱，person 3 比 person 1 有更多的钱，person 1 比 person 0 有更多的钱。

唯一较为安静（有较低的安静值 quiet[x]）的人是 person 7，

但是目前还不清楚他是否比 person 0 更有钱。

answer[7] = 7，

在所有拥有的钱肯定不少于 person 7 的人中(这可能包括 person 3，4，5，6 以及 7)，

最安静(有较低安静值 quiet[x])的人是 person 7。

其他的答案也可以用类似的推理来解释。

提示：

1. 1 <= quiet.length = N <= 500

2. 0 <= quiet[i] < N，所有 quiet[i] 都不相同。

3. 0 <= richer.length <= N \* (N-1) / 2

4. 0 <= richer[i][j] < N

5. richer[i][0] != richer[i][1]

6. richer[i] 都是不同的。

7. 对 richer 的观察在逻辑上是一致的。

8. 建立有向图，搜索遍历所有比它富有的人，取其中最安静的。

|  |
| --- |
| class Solution {  public int[] loudAndRich(int[][] richer, int[] quiet) {  List<Integer>[] map=new List[quiet.length];  for (int i=0;i<quiet.length;i++)map[i]=new ArrayList<>();  for (int[] r:richer)  {  map[r[1]].add(r[0]);  }  int[] res=new int[quiet.length];  Arrays.fill(res,-1);  for (int i=0;i<quiet.length;i++)  {  dfs(i,map,res,quiet);  }  return res;  }  private int dfs(int i, List<Integer>[] map, int[] res,int[] quiet) {  if(res[i]>=0)return res[i];  res[i]=i;  for (int j:map[i])  {  if(quiet[res[i]]>quiet[dfs(j,map,res,quiet)])  {  res[i]=res[j];  }  }  return res[i];  }  } |

|  |
| --- |
| class Solution {  public:  vector<int> loudAndRich(vector<vector<int>>& richer, vector<int>& quiet)  {  quiet\_ = quiet;  result\_ = vector<int>(quiet\_.size(), 0);  for (const auto& item : richer) {  graph\_[item[1]].push\_back(item[0]);  }  for (size\_t i = 0; i < result\_.size(); i++) {  pair<int, int> mini = { i, quiet[i] };  Dfs(i, mini, graph\_);  result\_[i] = mini.first;  cache\_[i] = mini;  }  return result\_;  }  void Dfs(int person, pair<int, int>& mini, map<int, vector<int>>& graph)  {  auto moreRichers = graph[person];  if (moreRichers.empty()) {  GetMiniQuiet(person, mini);  return;  }  if (cache\_.find(person) != cache\_.end()) {  auto newVal = cache\_[person];  if (newVal.second < mini.second) {  mini = newVal;  }  return;  }  for (const auto& moreRich : moreRichers) {  int newPerson = moreRich;  int newQuiet = quiet\_[newPerson];  if (newQuiet < mini.second) {  mini.first = newPerson;  mini.second = newQuiet;  }  Dfs(newPerson, mini, graph);  }  return;  }  void GetMiniQuiet(int person, pair<int, int>& mini) {  if (quiet\_[person] < mini.second) {  mini.first = person;  mini.second = quiet\_[person];  }  }  map<int, vector<int>> graph\_;  vector<int> quiet\_;  vector<int> result\_;  map<int, pair<int, int>> cache\_;  }; |

**967. 连续差相同的数字**

返回所有长度为 N 且满足其每两个连续位上的数字之间的差的绝对值为 K 的非负整数。

请注意，除了数字 0 本身之外，答案中的每个数字都不能有前导零。例如，01 因为有一个前导零，所以是无效的；但 0 是有效的。

你可以按任何顺序返回答案。

输入：N = 3, K = 7

输出：[181,292,707,818,929]

解释：注意，070 不是一个有效的数字，因为它有前导零。

输入：N = 2, K = 1

输出：[10,12,21,23,32,34,43,45,54,56,65,67,76,78,87,89,98]

|  |
| --- |
| class Solution {  public:  vector<int> numsSameConsecDiff(int N, int K)  {  int startNum = 1;  if (N == 1) {  startNum = 0;  }  if (K == 0) {  orient\_.insert(0);  } else {  orient\_.insert(-1 \* K);  orient\_.insert(K);  }  for (size\_t i = startNum; i < 10; i++) {  Dfs(i, i, N, 1);  }  return vector<int>(result\_.begin(), result\_.end());  }  int AddNum(int baseNum, int addNum)  {  return baseNum \* 10 + addNum;  }  void Dfs(int curt, int last, int N, int len)  {  if (len == N) {  result\_.insert(curt);  return;  }  for (auto& sel : orient\_) {  int newNum = last + sel;  if (newNum < 0 || newNum > 9) {  continue;  }  Dfs(AddNum(curt, newNum), newNum, N, len + 1);  }  }  set<int> orient\_;  set<int> result\_;  int n\_;  }; |

**1034. 边框着色**

给出一个二维整数网格 grid，网格中的每个值表示该位置处的网格块的颜色。

只有当两个网格块的颜色相同，而且在四个方向中任意一个方向上相邻时，它们属于同一连通分量。

连通分量的边界是指连通分量中的所有与不在分量中的正方形相邻（四个方向上）的所有正方形，或者在网格的边界上（第一行/列或最后一行/列）的所有正方形。

给出位于 (r0, c0) 的网格块和颜色 color，使用指定颜色 color 为所给网格块的连通分量的边界进行着色，并返回最终的网格 grid 。

输入：grid = [[1,1],[1,2]], r0 = 0, c0 = 0, color = 3

输出：[[3, 3], [3, 2]]

输入：grid = [[1,2,2],[2,3,2]], r0 = 0, c0 = 1, color = 3

输出：[[1, 3, 3], [2, 3, 3]]

输入：grid = [[1,1,1],[1,1,1],[1,1,1]], r0 = 1, c0 = 1, color = 2

输出：[[2, 2, 2], [2, 1, 2], [2, 2, 2]]

|  |
| --- |
| class Solution {  public int[][] colorBorder(int[][] grid, int r0, int c0, int color) {  if(grid == null || grid.length == 0) return grid;  boolean[][] visited = new boolean[grid.length][grid[0].length];  dfs(grid, r0, c0, color, visited, grid[r0][c0]);  return grid;  }  private int dfs(int[][] grid, int i, int j, int color, boolean[][] visited, int oldColor){  if(i < 0 || i >= grid.length || j < 0 || j >= grid[0].length ||  (!visited[i][j] && grid[i][j] != oldColor)) return 0;  if(visited[i][j]) return 1;  visited[i][j] = true;  int a = dfs(grid, i - 1, j, color, visited, oldColor)  + dfs(grid, i, j - 1, color, visited, oldColor)  + dfs(grid, i + 1, j, color, visited, oldColor)  + dfs(grid, i, j + 1, color, visited, oldColor);  if(a < 4){  grid[i][j] = color;  }  return 1;  }  }// 深度优先搜索。先全涂一个色，然后对边界和非边界分别涂色。 |

把指定位置的连通分量的那一整块边缘染成指定颜色， 只要四个方向有一个方向不是连通分量的颜色（包括边界）就给它染色。 理解题目之后就容易了， 本题采用BFS来遍历该连通分量。

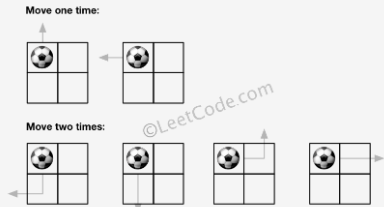
|  |
| --- |
| class Solution {  public int[][] colorBorder(int[][] grid, int r0, int c0, int color) {  Queue<int[]> queue = new LinkedList<>();  boolean[][] visit = new boolean[grid.length][grid[0].length];  int initColor = grid[r0][c0];  visit[r0][c0] = true;  queue.offer(new int[]{r0, c0});  int[][] dirs = new int[][]{{0, 1}, {0, -1}, {1, 0}, {-1, 0}};  while (!queue.isEmpty()) {  int[] point = queue.poll();  int x = point[0], y = point[1];  if (inBorder(grid, x, y))  grid[x][y] = color;  for (int[] dir: dirs) {  int nx = x + dir[0], ny = y + dir[1];  if (nx < 0 || nx >= grid.length || ny < 0 || ny >= grid[0].length || visit[nx][ny])  continue;  if (grid[nx][ny] == initColor) {  queue.offer(new int[]{nx, ny});  visit[nx][ny] = true;  }  else  grid[x][y] = color;  }  }  return grid;  }  public boolean inBorder(int[][] grid, int r, int c) {  return r == 0 || c == 0 || r == grid.length - 1 || c == grid[0].length - 1;  }  } |

**576. 出界的路径数**

给定一个 m × n 的网格和一个球。球的起始坐标为 (i,j) ，你可以将球移到相邻的单元格内，或者往上、下、左、右四个方向上移动使球穿过网格边界。但是，你最多可以移动 N 次。找出可以将球移出边界的路径数量。答案可能非常大，返回 结果 mod 109 + 7 的值。

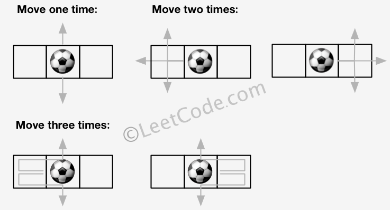
输入: m = 2, n = 2, N = 2, i = 0, j = 0

输出: 6



输入: m = 1, n = 3, N = 3, i = 0, j = 1

输出: 12



说明: 球一旦出界，就不能再被移动回网格内。

网格的长度和高度在 [1,50] 的范围内。

N 在 [0,50] 的范围内。

|  |
| --- |
| class Solution {  public:  int dx[4] = {-1, 0, 0, 1};  int dy[4] = {0, -1, 1, 0};  const int MOD = 1e9+7;  int findPaths(int m, int n, int N, int i, int j) {  return dfs(m, n, N, i, j);  }  private:  unordered\_map<string, int> memo;  int dfs(int m, int n, int N, int x, int y) {  if (N < 0) return 0;  if (x < 0 || x >= m || y < 0 || y >= n) {  return 1;  }  string state = to\_string(N) + "," + to\_string(x) + "," + to\_string(y);  if (memo.count(state)) return memo[state];  int ans = 0;  for (int i = 0; i < 4; i++) {  int nx = x + dx[i];  int ny = y + dy[i];  ans = (ans + dfs(m, n, N - 1, nx, ny)) % MOD;  }  return memo[state] = ans;  }  }; |