**102. 二叉树的层序遍历**

给你一个二叉树，请你返回其按 层序遍历 得到的节点值。 （即逐层地，从左到右访问所有节点）。

二叉树：[3,9,20,null,null,15,7],

3

/ \

9 20

/ \

15 7

返回其层次遍历结果：

[

[3],

[9,20],

[15,7]

]

|  |
| --- |
| class Solution {  public:  vector<vector<int>> levelOrder(TreeNode\* root) {  vector<vector<int>> res;  if (root == nullptr) {  return res;  }  queue<TreeNode\*> q;  q.push(root);  while (!q.empty()) {  int n = q.size();  res.push\_back({});  for (int i = 0; i < n; ++i) {  auto node = q.front();  q.pop();  res.rbegin()->push\_back(node->val);  if (node->left != nullptr) {  q.push(node->left);  }  if (node->right != nullptr) {  q.push(node->right);  }  }  }  return res;  }  }; |

**127. 单词接龙**

给定两个单词（beginWord 和 endWord）和一个字典，找到从 beginWord 到 endWord 的最短转换序列的长度。转换需遵循如下规则：

每次转换只能改变一个字母。

转换过程中的中间单词必须是字典中的单词。

说明:如果不存在这样的转换序列，返回 0。所有单词具有相同的长度。所有单词只由小写字母组成。字典中不存在重复的单词。你可以假设 beginWord 和 endWord 是非空的，且二者不相同。

输入:

beginWord = "hit",

endWord = "cog",

wordList = ["hot","dot","dog","lot","log","cog"]

输出: 5

解释: 一个最短转换序列是 "hit" -> "hot" -> "dot" -> "dog" -> "cog",返回它的长度 5。

输入:

beginWord = "hit"

endWord = "cog"

wordList = ["hot","dot","dog","lot","log"]

输出: 0

解释: endWord "cog" 不在字典中，所以无法进行转换。

|  |
| --- |
| class Solution {  public:  int ladderLength(string beginWord, string endWord, vector<string>& wordList)  {  unordered\_set<string> wordDict(wordList.begin(), wordList.end());  if (wordDict.find(endWord) == wordDict.end()) {  return 0;  }  set<string> visited;  queue<string> q;  q.push(beginWord);  visited.insert(beginWord);  int step = 1;  while (!q.empty()) {  ++step;  int n = q.size();  for (int index = 0; index < n; ++index) {  string cur = q.front();  q.pop();  for (int i = 0; i < cur.size(); ++i) { //修改1个单词，变化出可用的词  string str = cur;  for (char c = 'a'; c <= 'z'; ++c) {  if (c == cur[i]) {  continue;  }  str[i] = c;  if (visited.count(str) > 0 || wordDict.find(str) == wordDict.end()) {  continue;  }  if (str == endWord) {  return step;  }  q.push(str);  visited.insert(str);  }  }  }  }  return 0;  }  }; |

|  |
| --- |
| class Solution {  public:  int ladderLength(string beginWord, string endWord, vector<string>& wordList) {  unordered\_set<string> wordDict(wordList.begin(), wordList.end());  if (wordDict.find(endWord) == wordDict.end()){  return 0;  }  unordered\_set<string> beginSet{beginWord};  unordered\_set<string> endSet{endWord};  int step = 1;  while(!beginSet.empty()){  unordered\_set<string> tempSet;  ++step;  for (auto s : beginSet) {  wordDict.erase(s);  }  for (auto s : beginSet) {  for (int i = 0; i < s.size(); ++i){ //修改1个单词，变化出可用的词  string str = s;  for (char c = 'a'; c <= 'z'; ++c){  str[i] = c;  if (wordDict.find(str) == wordDict.end()){  continue;  }  if (endSet.find(str) != endSet.end()){  return step;  }  tempSet.insert(str);  }  }  }  if (tempSet.size() < endSet.size()){  beginSet = tempSet;  } else {  beginSet = endSet;// **双向BFS 调换顺序**  endSet = tempSet;  }  }  return 0;  }  }; |

**126. 单词接龙 II**

给定两个单词（beginWord 和 endWord）和一个字典 wordList，找出所有从 beginWord 到 endWord 的最短转换序列。转换需遵循如下规则：

每次转换只能改变一个字母。

转换后得到的单词必须是字典中的单词。

说明:

如果不存在这样的转换序列，返回一个空列表。

所有单词具有相同的长度。

所有单词只由小写字母组成。

字典中不存在重复的单词。

你可以假设 beginWord 和 endWord 是非空的，且二者不相同。

输入:

beginWord = "hit",

endWord = "cog",

wordList = ["hot","dot","dog","lot","log","cog"]

输出:

[

["hit","hot","dot","dog","cog"],

["hit","hot","lot","log","cog"]

]

输入:

beginWord = "hit"

endWord = "cog"

wordList = ["hot","dot","dog","lot","log"]

输出: []

解释: endWord "cog" 不在字典中，所以不存在符合要求的转换序列。

|  |
| --- |
| class Solution {  public:  vector<vector<string>> res;  **vector<string> path;**  vector<vector<string>> findLadders(string beginWord, string endWord, vector<string>& wordList) {  unordered\_set<string> s(wordList.begin(), wordList.end());  if (s.count(endWord) == 0) return res;  unordered\_map<string, int> dist;  **dist[beginWord] = 0;**  bfs(beginWord, endWord, s, dist);  // cout << dist[beginWord] << endl;  if (dist.count(endWord) == 0) return res;  path.emplace\_back(endWord);  dfs(endWord, beginWord, dist);  return res;  }  void bfs(string beginWord, string endWord, unordered\_set<string>& s, unordered\_map<string, int>& dist)  {  queue<string> q;  q.emplace(beginWord);  while (!q.empty()) {  auto t = q.front();  q.pop();  auto origin = t;  for (int i = 0; i < t.size(); ++i) {  t = origin;  for (char c = 'a'; c < 'z'; ++c) {  if (t[i] == c) continue;  t[i] = c;  if (s.count(t) > 0 && dist.count(t) == 0) {  **dist[t] = dist[origin] + 1;**  if (t == endWord) return;  q.emplace(t);  }  }  }  }  }  void dfs(string start, string end, unordered\_map<string, int>& dist)  {  if (start == end) {  reverse(path.begin(), path.end());  res.emplace\_back(path);  reverse(path.begin(), path.end());  return;  }  auto origin = start;  for (int i = 0; i < start.size(); ++i) {  start = origin;  for (char c = 'a'; c <= 'z'; ++c) {  if (start[i] == c) continue;  start[i] = c;  if (dist.count(start) > 0 && dist[origin] == dist[start] + 1) {  path.emplace\_back(start);  **dfs(start, end, dist);**  path.pop\_back();  }  }  }  }  }; |

**310. 最小高度树**

对于一个具有树特征的无向图，我们可选择任何一个节点作为根。图因此可以成为树，在所有可能的树中，具有最小高度的树被称为最小高度树。给出这样的一个图，写出一个函数找到所有的最小高度树并返回他们的根节点。

格式：该图包含 n 个节点，标记为 0 到 n - 1。给定数字 n 和一个无向边 edges 列表（每一个边都是一对标签）。

你可以假设没有重复的边会出现在 edges 中。由于所有的边都是无向边， [0, 1]和 [1, 0] 是相同的，因此不会同时出现在 edges 里。

输入: n = 4, edges = [[1, 0], [1, 2], [1, 3]]

0

|

1

/ \

2 3

输出: [1]

输入: n = 6, edges = [[0, 3], [1, 3], [2, 3], [4, 3], [5, 4]]

0 1 2

\ | /

3

|

4

|

5

输出: [3, 4]

|  |
| --- |
| class Solution {  public:  vector<int> findMinHeightTrees(int n, vector<vector<int>>& edges) {  if (n == 0) {  return {};  }  if (n == 1) {  return { 0 };  }  vector<list<int>> graph(n, list<int>());  for (const auto& edge : edges) {  graph[edge[0]].push\_back(edge[1]);  graph[edge[1]].push\_back(edge[0]);  }  list<int> q;  for (size\_t i = 0; i < graph.size(); i++) {  if (graph[i].size() == 1) {  q.push\_back(i);  }  }  if (q.empty()) {  return {};  }  int res = n;  while (res > 2) {  list<int> tmp;  for (auto node : q) {  int to = graph[node].front();  graph[node].clear();  graph[to].remove(node);  res--;  if (graph[to].size() == 1) {  tmp.push\_back(to);  }  }  q = tmp;  }  return vector<int>(q.begin(), q.end());  }  }; 类拓扑排序。找到只有一个邻居的点加入队列，每pop一个节点，它的邻居的邻接表就减去这一节点，直到只剩下一/两个节点 |

**301. 删除无效的括号**

删除最小数量的无效括号，使得输入的字符串有效，返回所有可能的结果。

说明: 输入可能包含了除 ( 和 ) 以外的字符。

输入: "()())()"

输出: ["()()()", "(())()"]

输入: "(a)())()"

输出: ["(a)()()", "(a())()"]

输入: ")("

输出: [""]

|  |
| --- |
| class Solution {  public:  vector<string> removeInvalidParentheses(string s) {  vector<string> res;  remove(move(s), {'(', ')'}, 0, 0, res);  return res;  }  void remove(std::string s, const vector<char>& par, int m, int n, vector<string>& res) {  int stack = 0, i = m;  for (int i = m; i < s.length(); ++i) {  if (s[i] == par[0]) stack++;  if (s[i] == par[1]) stack--;  if (stack >= 0) continue;  // "右"括号多出来了，删除一个右括号  for (int j = n; j <= i; ++j) {  if (s[j] == par[1] && (j == n || s[j-1] != par[1])) {  auto ss = s.substr(0, j) + s.substr(j + 1);  remove(move(ss), par, i, j, res);  }  }  return;  }  reverse(s.begin(), s.end());  if (par[0] == '(') {  remove(move(s), {par[1], par[0]}, 0, 0, res);  } else {  res.push\_back(move(s));  }  }  }; |

**513. 找树左下角的值**

给定一个二叉树，在树的最后一行找到最左边的值。

|  |
| --- |
| int findBottomLeftValue(TreeNode\* root) {  queue<TreeNode\*> q;  q.push(root);  int res = 0;  while (!q.empty()) {  auto node = q.front();  q.pop();  res = node->val;  if (node->right) q.push(node->right);  if (node->left) q.push(node->left);  }  return res;  } |

**301. 删除无效的括号**

|  |
| --- |
| class Solution {  public:  vector<string> removeInvalidParentheses(string s)  {  vector<string> result;  queue<string> q;  set<string> visited\_;  q.push(s);  visited\_.insert(s);  int layer = 0;  bool hasValid = false;  while (!q.empty()) {  auto s = q.front();  q.pop();  if (IsValid(s)) {  result.push\_back(s);  hasValid = true;  }  if (hasValid) {  continue;  }  for (size\_t i = 0; i < s.size(); i++) {  if (s[i] == '(' || s[i] == ')') {  string newStr = s.substr(0, i) + s.substr(i + 1);//0 len-1无需特殊处理  if (visited\_.count(newStr) == 0) {  visited\_.insert(newStr);  q.push(newStr);  }  }  }  }  return result.empty() ? vector<string>(1, "") : result;  }  bool IsValid(const string& s)  {  auto flags = 0;  for (auto c : s) {  if (c == '(') {  flags++;  } else if (c == ')'){  flags--;  }  if (flags < 0) {  return false;  }  }  return flags == 0;  }  };// 如果我们每次只删除一个括号，然后观察被删除一个括号后是否合法，如果已经合法了，就不用继续删除了，但需要遍历完本层；因此我们并不需要将遍历进行到底，而是层层深入，一旦达到需求，就不再深入了。 |

**515. 在每个树行中找最大值**

您需要在二叉树的每一行中找到最大的值。

输入:

1

/ \

3 2

/ \ \

5 3 9

输出: [1, 3, 9]

|  |
| --- |
| class Solution {  public List<Integer> largestValues(TreeNode root) {  List<Integer> res = new LinkedList<>();  if (null == root) {  return res;  }  LinkedList<TreeNode> queue = new LinkedList<>();  queue.add(root);  while (!queue.isEmpty()) {  int size = queue.size();  int max = queue.getFirst().val;  for (int i = 0; i < size; i++) {  TreeNode node = queue.removeFirst();  if (node.val > max) {  max = node.val;  }  if (node.left != null) {  queue.addLast(node.left);  }  if (node.right != null) {  queue.addLast(node.right);  }  }  res.add(max);  }  return res;  } |

|  |
| --- |
| class Solution {  public List<Integer> largestValues(TreeNode root) {  List<Integer> res = new ArrayList<>();  levelOrder(root, res, 0);  return res;  }  private void levelOrder(TreeNode root, List<Integer> collectors, int level) {  if (root == null) {  return;  }  if (level >= collectors.size()) {  collectors.add(level, root.val);  } else {  Integer val = collectors.get(level);  if (root.val > val) {  collectors.set(level, root.val);  }  }  levelOrder(root.left, collectors, level + 1);  levelOrder(root.right, collectors, level + 1);  }  }// DFS |

**542. 01 矩阵**

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 。

输入： 输出：

[[0,0,0], [[0,0,0],

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

[0,0,0]] [0,0,0]]

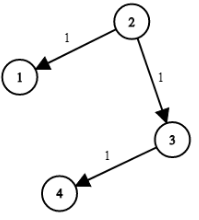
输入： 输出：

[[0,0,0], [[0,0,0],

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

[1,1,1]] [1,2,1]]

|  |
| --- |
| struct P{  int x, y, dis;  };  class Solution {  public:  int dx[4] = {1, -1, 0, 0}, dy[4] = {0, 0, 1, -1};  vector<vector<int>> updateMatrix(vector<vector<int>>& matrix) {  queue<P> q;  int m = matrix.size(), n = matrix[0].size();  vector<vector<int>> vis(m, vector<int>(n, 0));  vector<vector<int>> res(m, vector<int>(n, 0));  for(int i = 0; i < m; i++)  for (int j = 0; j < n; j++)  if(matrix[i][j] == 0) q.push({i, j, 0}), vis[i][j] = 1;  while(!q.empty())  {  P t = q.front();  q.pop();  int x = t.x, y = t.y, d = t.dis;  vis[x][y] = 1;  for(int k = 0; k < 4; k++)  {  int nx = x + dx[k], ny = y + dy[k];  if(nx<0 || nx>=m || ny<0 || ny>=n || vis[nx][ny]) continue;  q.push({nx, ny, d + 1});  res[nx][ny] = d + 1;  vis[nx][ny] = 1;  }  }  return res;  }  };// 多源BFS板题 |



**743. 网络延迟时间**

有 N 个网络节点，标记为 1 到 N。

给定一个列表 times，表示信号经过有向边的传递时间。 times[i] = (u, v, w)，其中 u 是源节点，v 是目标节点， w 是一个信号从源节点传递到目标节点的时间。

现在，我们从某个节点 K 发出一个信号。需要多久才能使所有节点都收到信号？如果不能使所有节点收到信号，返回 -1。

输入：times = [[2,1,1],[2,3,1],[3,4,1]], N = 4, K = 2

输出：2

|  |
| --- |
| class Solution { // DFS  vector<vector<pair<int, int>>> graph;  vector<int> dp;  int cost;  public:  int networkDelayTime(vector<vector<int>>& times, int N, int K) {  graph.resize(N+1);  dp.resize(N+1, 101);  for(auto time: times) {  graph[time[0]].emplace\_back(time[1], time[2]);  }  cost = 0;  dp[K] = 0;  dfs(K);  for(int i=1; i<=N; i++)  if(dp[i] > cost)  cost = dp[i];  return cost > 100 ? -1 : cost;  }  void dfs(int u) {  for(auto e: graph[u]) {  int v = e.first;  int w = e.second;  if(dp[u] + w < dp[v]) {  dp[v] = dp[u] + w;  dfs(v);  }  }  }  }; |

|  |
| --- |
| class Solution { // BFS  vector<vector<pair<int, int>>> graph;  vector<int> dp;  int cost;  public:  int networkDelayTime(vector<vector<int>>& times, int N, int K) {  graph.resize(N+1);  dp.resize(N+1, 101);  for(auto time: times) {  graph[time[0]].emplace\_back(time[1], time[2]);  }  cost = 0;  dp[K] = 0;  queue<int> Q;  Q.push(K);  while(!Q.empty()) {  int u = Q.front();  Q.pop();  for(auto e: graph[u]) {  int v = e.first;  int w = e.second;  if(dp[u] + w < dp[v]) {  dp[v] = dp[u] + w;  Q.push(v);  }  }  }  for(int i=1; i<=N; i++)  if(dp[i] > cost)  cost = dp[i];  return cost > 100 ? -1 : cost;  }  }; |

**752. 打开转盘锁**

你有一个带有四个圆形拨轮的转盘锁。每个拨轮都有10个数字： '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' 。每个拨轮可以自由旋转：例如把 '9' 变为 '0'，'0' 变为 '9' 。每次旋转都只能旋转一个拨轮的一位数字。

锁的初始数字为 '0000' ，一个代表四个拨轮的数字的字符串。

列表 deadends 包含了一组死亡数字，一旦拨轮的数字和列表里的任何一个元素相同，这个锁将会被永久锁定，无法再被旋转。

字符串 target 代表可以解锁的数字，你需要给出最小的旋转次数，如果无论如何不能解锁，返回 -1。

输入：deadends = ["0201","0101","0102","1212","2002"], target = "0202"

输出：6

解释：

可能的移动序列为 "0000" -> "1000" -> "1100" -> "1200" -> "1201" -> "1202" -> "0202"。

注意 "0000" -> "0001" -> "0002" -> "0102" -> "0202" 这样的序列是不能解锁的，

因为当拨动到 "0102" 时这个锁就会被锁定。

输入: deadends = ["8888"], target = "0009"

输出：1

解释：

把最后一位反向旋转一次即可 "0000" -> "0009"。

输入: deadends = ["0000"], target = "8888"

输出：-1

|  |
| --- |
| class Solution {  public:  int openLock(vector<string>& deadends, string target)  {  unordered\_set<string> deadlocks(deadends.begin(), deadends.end());  if (deadlocks.count(target) != 0 || deadlocks.count("0000") != 0) return -1;  queue<string> q;  q.push("0000");  unordered\_set<string> visited;  visited.insert("0000");  vector<int> directs = { -1,1 };  int res = 0;  while (!q.empty()) {  int size = q.size();  while (size-->0) {  auto t = q.front();  q.pop();  if (t == target) return res;  if (deadlocks.count(t)) return -1;  for (int i = 0; i<t.size(); i++) {  for (auto direct : directs) {  string newWord = t;  newWord[i] = (newWord[i] - '0' + 10 + direct) % 10 + '0';  if (visited.count(newWord) || deadlocks.count(newWord)) continue;  q.push(newWord);  visited.insert(newWord);  }  }  }  res++;  }  return -1;  }  }; |

**773. 滑动谜题**

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.一次移动定义为选择 0 与一个相邻的数字（上下左右）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 。

输入：board = [[1,2,3],[4,0,5]]

输出：1

解释：交换 0 和 5 ，1 步完成

输入：board = [[1,2,3],[5,4,0]]

输出：-1

解释：没有办法完成谜板

输入：board = [[4,1,2],[5,0,3]]

输出：5

解释：

最少完成谜板的最少移动次数是 5 ，

一种移动路径:

尚未移动: [[4,1,2],[5,0,3]]

移动 1 次: [[4,1,2],[0,5,3]]

移动 2 次: [[0,1,2],[4,5,3]]

移动 3 次: [[1,0,2],[4,5,3]]

移动 4 次: [[1,2,0],[4,5,3]]

移动 5 次: [[1,2,3],[4,5,0]]

输入：board = [[3,2,4],[1,5,0]]

输出：14

|  |
| --- |
| class Solution {  public:  int slidingPuzzle(vector<vector<int>>& board)  {  int row = board.size();  int col = board[0].size();  string begin = "";  string target = "123450";  int depth = 0;  // 0 所在位置要移动的下标数组  vector<vector<int>> moves{ { 1, 3 },{ 0, 2, 4 },{ 1, 5 },  { 0, 4 },{ 1, 3, 5 },{ 2, 4 } };  // 转换为一维字符串数组  for (int i = 0; i < row; i++) {  for (int j = 0; j < col; j++) {  begin += to\_string(board[i][j]);  }  }  unordered\_set<string> visited;  queue<string> q;  q.push(begin);  visited.insert(begin);  while (!q.empty()) {  int n = q.size();  for (int i = 0; i < n; i++) {  string curr = q.front();  q.pop();  if (curr == target) {  return depth;  }  int zero\_index = curr.find("0");  for (auto next : moves[zero\_index]) {  swap(curr[next], curr[zero\_index]);  if (!visited.count(curr)) {  q.push(curr);  visited.insert(curr);  }  // reverse the state  swap(curr[next], curr[zero\_index]);  }  }  depth++;  }  return -1;  }  }; |

**778. 水位上升的泳池中游泳**

在一个 N x N 的坐标方格 grid 中，每一个方格的值 grid[i][j] 表示在位置 (i,j) 的平台高度。

现在开始下雨了。当时间为 t 时，此时雨水导致水池中任意位置的水位为 t 。你可以从一个平台游向四周相邻的任意一个平台，但是前提是此时水位必须同时淹没这两个平台。假定你可以瞬间移动无限距离，也就是默认在方格内部游动是不耗时的。当然，在你游泳的时候你必须待在坐标方格里面。你从坐标方格的左上平台 (0，0) 出发。最少耗时多久你才能到达坐标方格的右下平台 (N-1, N-1)？

输入: [[0,2],[1,3]]

输出: 3

解释:

时间为0时，你位于坐标方格的位置为 (0, 0)。此时你不能游向任意方向，因为四个相邻方向平台的高度都大于当前时间为 0 时的水位。等时间到达 3 时，你才可以游向平台 (1, 1). 因为此时的水位是 3，坐标方格中的平台没有比水位 3 更高的，所以你可以游向坐标方格中的任意位置

输入: [[0,1,2,3,4],[24,23,22,21,5],[12,13,14,15,16],[11,17,18,19,20],[10,9,8,7,6]]

输出: 16

解释:

**0 1 2 3 4**

24 23 22 21  **5**

**12 13 14 15 16**

**11** 17 18 19 20

**10 9 8 7 6**

最终的路线用加粗进行了标记。我们必须等到时间为 16，此时才能保证平台 (0, 0) 和 (4, 4) 是连通的

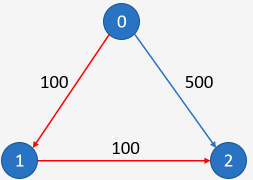
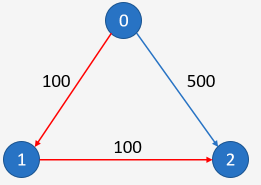
|  |
| --- |
| struct Position {  Position(int x, int y, int h) : x\_(x), y\_(y), h\_(h) {}  int x\_;  int y\_;  int h\_;  };  bool operator< (const Position& left, const Position& right)  {  return left.h\_ > right.h\_;  }  class Solution {  public:  int swimInWater(vector<vector<int>>& grid)  {  n\_ = grid.size();  m\_ = grid[0].size();  if (n\_ == 0 || m\_ == 0) {  return 0;  }  priority\_queue<Position> q;  q.push(Position(0, 0, grid[0][0]));  vector<vector<bool>> visited(n\_, vector<bool>(m\_, false));  visited[0][0] = true;  int res = 0;  while (!q.empty()) {  auto tmp = q.top();  q.pop();  res = max(res, tmp.h\_);  if (tmp.x\_ == m\_ - 1 && tmp.y\_ == n\_ - 1) {  break;  }  for (const auto& ori : orient\_) {  int newY = tmp.y\_ + ori[0];  int newX = tmp.x\_ + ori[1];  if (!ValidPos(newX, newY)) {  continue;  }  if (visited[newY][newX]) {  continue;  }  q.push(Position(newX, newY, grid[newY][newX]));  visited[newY][newX] = true;  }  }  return res;  }  bool ValidPos(int x, int y)  {  return (x >= 0) && (x < m\_) && (y >= 0) && (y < n\_);  }  int n\_{ 0 };  int m\_{ 0 };  vector<vector<int>> orient\_{ {-1, 0}, {1, 0},{0, -1}, {0, 1} };  };//宽度优先搜索。使用优先队列，每次选择值最小的作为下一个，并更新最大值作为结果。 |

**787. K 站中转内最便宜的航班**

有 n 个城市通过 m 个航班连接。每个航班都从城市 u 开始，以价格 w 抵达 v。

现在给定所有的城市和航班，以及出发城市 src 和目的地 dst，你的任务是找到从 src 到 dst 最多经过 k 站中转的最便宜的价格。 如果没有这样的路线，则输出 -1。

例1 例2



输入: n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]

src = 0, dst = 2, k = 1

输出: 200

解释: 从城市 0 到城市 2 在 1 站中转以内的最便宜价格是 200，如图中红色所示。

输入: n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]

src = 0, dst = 2, k = 0

输出: 500

解释: 从城市 0 到城市 2 在 0 站中转以内的最便宜价格是 500，如图中蓝色所示。

|  |
| --- |
| class Solution {  public:  int findCheapestPrice(int n, vector<vector<int>>& flights, int src, int dst, int K) {  int res = -1;  int minPrice = INT\_MAX;  queue<vector<int>> q;  q.push({ 0, src });  while (!q.empty()) {  auto tmp = q.front();  int n = tmp.size();  q.pop();  if (tmp[n - 1] == dst && tmp[0] < minPrice) {  minPrice = tmp[0];  }  if ((tmp.size() > K + 2) || tmp[0] > minPrice) {  continue;  }  for (const auto& flight : flights) {  if (tmp[n - 1] == flight[0] &&  find(tmp.begin() + 1, tmp.end(), flight[0]) != tmp.end()) {  vector<int> newTrace(tmp);  newTrace[0] += flight[2];  newTrace.push\_back(flight[1]);  q.push(newTrace);  }  }  }  return minPrice == INT\_MAX ? -1 : minPrice;  }  }; //由于有k的次数限制，用宽搜会更加直观。记录当前到达终点的最短路径，如果有更短的，把新的节点加入队列。 |

**847. 访问所有节点的最短路径**

给出 graph 为有 N 个节点（编号为 0, 1, 2, ..., N-1）的无向连通图。graph.length = N，且只有节点 i 和 j 连通时，j != i 在列表 graph[i] 中恰好出现一次。返回能够访问所有节点的最短路径的长度。你可以在任一节点开始和停止，也可以多次重访节点，并且可以重用边。

输入：[[1,2,3],[0],[0],[0]]

输出：4

解释：一个可能的路径为 [1,0,2,0,3]

输入：[[1],[0,2,4],[1,3,4],[2],[1,2]]

输出：4

解释：一个可能的路径为 [0,1,4,2,3]

提示：1 <= graph.length <= 12 0 <= graph[i].length < graph.length

|  |
| --- |
| class Solution {  struct Status {  Status(int pos, int visted) : visted\_(visted), pos\_(pos) {}  int visted\_{ 0 };  int pos\_{ 0 };  };  public:  int shortestPathLength(vector<vector<int>>& graph)  {  int n = graph.size();  queue<Status> q;  for (size\_t i = 0; i < graph.size(); i++) {  q.push(Status(i, 1 << i));  }  vector<vector<bool>> visited(1 << N, vector<bool>(N, false));  const int END\_STATUS = { (1 << n) - 1 };  int step = 0;  while (!q.empty()) {  int size = q.size();  for (size\_t i = 0; i < size; i++) {  auto status = q.front();  q.pop();  int curPos = status.pos\_;  int curStatus = status.visted\_;  if (curStatus == END\_STATUS) {  return step;  }  for (int pos : graph[curPos]) {  int nextStatus = curStatus | (1 << pos);  if (visited[nextStatus][pos]) {  continue;  }  visited[nextStatus][pos] = true;  q.push(Status(pos, nextStatus));  }  }  step++;  }  return -1;  }  int N{ 12 };  };//单源无权最短路径带状态,判断重复节点访问需要考虑当前状态 |

**787. K 站中转内最便宜的航班**

|  |
| --- |
| typedef tuple<int, int, int> ti;// fee station k  class Solution {  public:  int findCheapestPrice(int n, vector<vector<int>>& flights, int src, int dst, int K) {  vector<vector<pair<int, int>>> graph(n);  for (const auto& f : flights) {  graph[f[0]].emplace\_back(f[1], f[2]);  }  priority\_queue<ti, vector<ti>, greater<ti>> q;  q.emplace(0, src, K + 1);  while (!q.empty()) {  auto [cost, u, stops] = q.top();  q.pop();  if (u == dst) {  return cost;  }  if (stops == 0) {  continue;  }  for (auto to : graph[u]) {  auto [v, w] = to;  q.emplace(cost + w, v, stops - 1);  }  }  return -1;  }  }; |

**854. 相似度为 K 的字符串**

如果可以通过将 A 中的两个小写字母精确地交换位置 K 次得到与 B 相等的字符串，我们称字符串 A 和 B 的相似度为 K（K 为非负整数）。

给定两个字母异位词 A 和 B ，返回 A 和 B 的相似度 K 的最小值。

输入：A = "ab", B = "ba"

输出：1

输入：A = "abc", B = "bca"

输出：2

输入：A = "abac", B = "baca"

输出：2

输入：A = "aabc", B = "abca"

输出：2

|  |
| --- |
| class Solution {  public:  int kSimilarity(string A, string B)  {  if (A == B) {  return 0;  }  if (A.size() == 1) {  return -1;  }  set<string> visted;  int k = 0;  queue<string> q;  q.push(A);  visted.insert(A);  while (!q.empty()) {  int n = q.size();  for (size\_t layer = 0; layer < n; layer++) {  string tmp = q.front();  q.pop();  if (tmp == B) {  return k;  }  int i = 0;  while (tmp[i] == B[i]) {  i++;// 找到第一个不相等的位置  }  for (size\_t j = i + 1; j < tmp.size(); j++) {  string newStr = tmp;  //避免本来所在字符相等; 再后找到与B[i]相同的字符对调  if (tmp[j] == B[j] || tmp[j] != B[i]) {  continue;  }  swap(newStr[i], newStr[j]);  if (visted.find(newStr) != visted.end()) {  continue;  }  q.push(newStr);  visted.insert(newStr);  }  }  k++;  }  return -1;  }  }; |

**863. 二叉树中所有距离为 K 的结点**

给定一个二叉树（具有根结点 root）， 一个目标结点 target ，和一个整数值 K 。

返回到目标结点 target 距离为 K 的所有结点的值的列表。 答案可以以任何顺序返回。

输入：root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, K = 2

输出：[7,4,1]

解释：

所求结点为与目标结点（值为 5）距离为 2 的结点，

值分别为 7，4，以及 1

注意，输入的 "root" 和 "target" 实际上是树上的结点。

上面的输入仅仅是对这些对象进行了序列化描述。

|  |
| --- |
| class Solution {  public:  vector<int> distanceK(TreeNode\* root, TreeNode\* target, int K)  {  map<int, vector<int>> graph;  queue<TreeNode\*> q;  q.push(root);  while (!q.empty()) {  auto\* node = q.front();  q.pop();  auto\* left = node->left;  auto\* right = node->right;  if (left != nullptr) {  graph[node->val].push\_back(left->val);  graph[left->val].push\_back(node->val);  q.push(left);  }  if (right != nullptr) {  graph[node->val].push\_back(right->val);  graph[right->val].push\_back(node->val);  q.push(right);  }  }  int targetVal = target->val;  queue<int> bfsQueue;  set<int> visited;  bfsQueue.push(targetVal);  visited.insert(targetVal);  int layer = 0;  vector<int> res;  while (!bfsQueue.empty()) {  int n = bfsQueue.size();  for (size\_t i = 0; i < n; i++) {  int cur = bfsQueue.front();  bfsQueue.pop();  if (layer == K) {  res.push\_back(cur);  continue;  }  for (auto next : graph[cur]) {  if (visited.find(next) != visited.end()) {  continue;  }  bfsQueue.push(next);  visited.insert(next);  }  }  layer++;  }  return res;  }  }; |

// DFS

|  |
| --- |
| class Solution {  map<TreeNode\*, TreeNode\*> parents\_;  set<TreeNode\*> visited\_;  TreeNode\* target\_{nullptr};  public:  vector<int> distanceK(TreeNode\* root, TreeNode\* target, int K) {  if (root == nullptr || target == nullptr) {  return {};  }  FillParent(root, nullptr);  vector<int> res;  Dfs(target, res, K);  return res;  }  void FillParent(TreeNode\* node, TreeNode\* parent) {  if (node == nullptr) {  return;  }  parents\_[node] = parent;  FillParent(node->left, node);  FillParent(node->right, node);  }  void Dfs(TreeNode\* node, vector<int>& res, int k) {  if (node == nullptr || visited\_.count(node) > 0) {  return;  }  visited\_.insert(node);  if (k == 0) {  res.push\_back(node->val);  return;  }  Dfs(node->left, res, k - 1);  Dfs(node->right, res, k - 1);  Dfs(parents\_[node], res, k - 1);  }  }; |

**864. 获取所有钥匙的最短路径**

给定一个二维网格 grid。 "." 代表一个空房间， "#" 代表一堵墙， "@" 是起点，（"a", "b", ...）代表钥匙，（"A", "B", ...）代表锁。

我们从起点开始出发，一次移动是指向四个基本方向之一行走一个单位空间。我们不能在网格外面行走，也无法穿过一堵墙。如果途经一个钥匙，我们就把它捡起来。除非我们手里有对应的钥匙，否则无法通过锁。

假设 K 为钥匙/锁的个数，且满足 1 <= K <= 6，字母表中的前 K 个字母在网格中都有自己对应的一个小写和一个大写字母。换言之，每个锁有唯一对应的钥匙，每个钥匙也有唯一对应的锁。另外，代表钥匙和锁的字母互为大小写并按字母顺序排列。返回获取所有钥匙所需要的移动的最少次数。如果无法获取所有钥匙，返回 -1 。

输入：["@.a.#","###.#","b.A.B"]

输出：8

输入：["@..aA","..B#.","....b"]

输出：6

|  |
| --- |
| struct Status {  Status(int x, int y, int status) : x\_(x), y\_(y), status\_(status) {}  int x\_;  int y\_;  **int status\_;**  };  class Solution {  public:  int shortestPathAllKeys(vector<string>& grid)  {  n\_ = grid.size();  m\_ = grid[0].size();  auto startPos = GetStartPos(grid);  auto locks = GetLocks(grid);  int lockNum = locks.size();  int END\_STATUS = (1 << lockNum) - 1;  vector<vector<vector<bool>>> visited((1 << lockNum), vector<vector<bool>>(n\_, vector<bool>(m\_, false)));  queue<Status> q;  q.push(Status(startPos.first, startPos.second, 0));  visited[0][startPos.second][startPos.first] = true;  int step = 0;  while (!q.empty()) {  int size = q.size();  for (size\_t i = 0; i < size; i++) {  auto status = q.front();  q.pop();  if (status.status\_ == END\_STATUS) {  return step;  }  for (auto& ori : orient\_) {  int newX = status.x\_ + ori[0];  int newY = status.y\_ + ori[1];  if (!IsValidPos(newX, newY)) {  continue;  }  if (visited[status.status\_][newY][newX]) {  continue;  }  char c = grid[newY][newX];  if (c == '#') {  continue;  }  // 是房间，但没有锁，无法继续  if (IsHouse(locks, c) && ((1 << (c - 'A')) & status.status\_) == false) {  continue;  }  int newStatus = status.status\_;  if (IsLock(locks, c)) {  newStatus |= (1 << (c - 'a'));  }  q.push(Status(newX, newY, newStatus));  visited[newStatus][newY][newX] = true;  }  }  step++;  }  return -1;  }  pair<int, int> GetStartPos(const vector<string>& grid) {  for (int i = 0; i < grid.size(); i++) {  for (int j = 0; j < grid[0].size(); j++) {  if (grid[i][j] == '@') {  return { j, i };  }  }  }  return { -1, -1 };  }  set<char> GetLocks(const vector<string>& grid) {  set<char> locks;  for (int i = 0; i < grid.size(); i++) {  for (int j = 0; j < grid[0].size(); j++) {  if (grid[i][j] >= 'a' && grid[i][j] <= 'f') {  locks.insert(grid[i][j]);  }  }  }  return locks;  }  bool IsLock(const set<char>& locks, char c) {  return islower(c) && locks.find(c) != locks.end();  }  bool IsHouse(const set<char>& locks, char c)  {  return isupper(c) && locks.find(tolower(c)) != locks.end();  }  bool IsValidPos(int x, int y) {  return (x >= 0) && (x < m\_) && (y >= 0) && (y < n\_);  }  vector<vector<int>> orient\_{ {1, 0},{ -1, 0 },{ 0, 1 },{ 0, -1 }};  int n\_{ 0 };  int m\_{ 0 };  }; |

**934. 最短的桥**

在给定的二维二进制数组 A 中，存在两座岛。（岛是由四面相连的 1 形成的一个最大组。）现在，我们可以将 0 变为 1，以使两座岛连接起来，变成一座岛。返回必须翻转的 0 的最小数目。（可以保证答案至少是 1。）

输入：[[0,1],[1,0]]

输出：1

输入：[[0,1,0],[0,0,0],[0,0,1]]

输出：2

输入：[[1,1,1,1,1],[1,0,0,0,1],[1,0,1,0,1],[1,0,0,0,1],[1,1,1,1,1]]

输出：1

|  |
| --- |
| class Solution {  public:  int shortestBridge(vector<vector<int>>& A)  {  n\_ = A.size();  m\_ = A[0].size();  bool finded = false;  for (size\_t i = 0; i < n\_; i++) {  for (size\_t j = 0; j < m\_; j++) {  if (A[i][j] == 1) {  Dfs(A, i, j);  finded = true;  break;  }  }  if (finded) {  break;  }  }  int step = 0;  while (!island\_.empty()) {  int n = island\_.size();  for (size\_t i = 0; i < n; i++) {  auto pos = island\_.front();  island\_.pop();  for (auto& ori : orient\_) {  int newY = pos.first + ori.first;  int newX = pos.second + ori.second;  if (!IsValidPos(newY, newX)) {  continue;  }  if (A[newY][newX] == 2) {  continue;  }  if (A[newY][newX] == 1) {  return step;  }  island\_.push({ newY, newX });  A[newY][newX] = 2;  }  }  step++;  }  return 0;  }  void Dfs(vector<vector<int>>& A, int y, int x) {  A[y][x] = 2;  island\_.push({ y, x });  for (auto& ori : orient\_) {  int newY = y + ori.first;  int newX = x + ori.second;  if (IsValidPos(newY, newX) && A[newY][newX] == 1) {  Dfs(A, newY, newX);  }  }  }  bool IsValidPos(int y, int x) {  return (x >= 0) && (x < m\_) && (y >= 0) && (y < n\_);  }  int n\_{ 0 };  int m\_{ 0 };  queue<pair<int, int>> island\_;  vector<pair<int, int>> orient\_{ {-1, 0},{ 1, 0 },{ 0, -1 },{ 0, 1 } };  }; |

**1091. 二进制矩阵中的最短路径**

在一个 N × N 的方形网格中，每个单元格有两种状态：空（0）或者阻塞（1）。

一条从左上角到右下角、长度为 k 的畅通路径，由满足下述条件的单元格 C\_1, C\_2, ..., C\_k 组成：

相邻单元格 C\_i 和 C\_{i+1} 在八个方向之一上连通（此时，C\_i 和 C\_{i+1} 不同且共享边或角）

C\_1 位于 (0, 0)（即，值为 grid[0][0]）

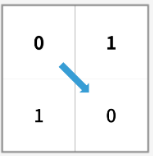
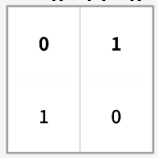
C\_k 位于 (N-1, N-1)（即，值为 grid[N-1][N-1]）

如果 C\_i 位于 (r, c)，则 grid[r][c] 为空（即，grid[r][c] == 0）

返回这条从左上角到右下角的最短畅通路径的长度。如果不存在这样的路径，返回 -1 。

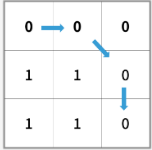
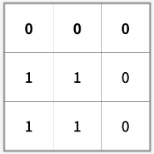
输入：[[0,1],[1,0]]

输出：2



输入：[[0,0,0],[1,1,0],[1,1,0]]

输出：4



|  |
| --- |
| class Solution {  public:  int shortestPathBinaryMatrix(vector<vector<int>>& grid)  {  n\_ = grid.size();  m\_ = grid[0].size();  if (grid[0][0] == 1 || grid[n\_ - 1][m\_ - 1] == 1) {  return -1;  }  queue<vector<int>> q;  q.push({ 0, 0, 1 });  set<pair<int, int>> visited\_;  visited\_.insert({ 0, 0 });  while (!q.empty()) {  int n = q.size();  for (int i = 0; i < n; i++) {  const auto& tmp = q.front();  int y = tmp[0];  int x = tmp[1];  int dis = tmp[2];  if (y == n\_ - 1 && x == m\_ - 1) {  return dis;  }  for (auto& ori : orient\_) {  int newY = y + ori[0];  int newX = x + ori[1];  int newDis = dis + ori[2];  if (!IsValidPos(newY, newX)) {  continue;  }  if (grid[newY][newX] == 1) {  continue;  }  if (visited\_.find({ newY, newX }) != visited\_.end()) {  continue;  }  q.push({ newY, newX, newDis });  visited\_.insert({ newY, newX });  }  q.pop();  }  }  return -1;  }  bool IsValidPos(int y, int x)  {  return x >= 0 && x < m\_ && y >= 0 && y < n\_;  }  int n\_{ 0 };  int m\_{ 0 };  vector<vector<int>> orient\_{ {-1, -1, 1}, {0, -1, 1},  {1, -1, 1}, {1, 0, 1},  { 1, 1, 1},{ 0, 1, 1 },  { -1, 1, 1 },{-1, 0, 1 } };  }; |

**1129. 颜色交替的最短路径**

在一个有向图中，节点分别标记为 0, 1, ..., n-1。这个图中的每条边不是红色就是蓝色，且存在自环或平行边。

red\_edges 中的每一个 [i, j] 对表示从节点 i 到节点 j 的红色有向边。类似地，blue\_edges 中的每一个 [i, j] 对表示从节点 i 到节点 j 的蓝色有向边。

返回长度为 n 的数组 answer，其中 answer[X] 是从节点 0 到节点 X 的红色边和蓝色边交替出现的最短路径的长度。如果不存在这样的路径，那么 answer[x] = -1。

输入：n = 3, red\_edges = [[0,1],[1,2]], blue\_edges = []

输出：[0,1,-1]

输入：n = 3, red\_edges = [[0,1]], blue\_edges = [[2,1]]

输出：[0,1,-1]

输入：n = 3, red\_edges = [[1,0]], blue\_edges = [[2,1]]

输出：[0,-1,-1]

|  |
| --- |
| class Solution {  public:  vector<int> shortestAlternatingPaths(int n, vector<vector<int>>& red\_edges, vector<vector<int>>& blue\_edges) {  vector<map<int, set<int>>> grid(n);  for (auto edge : red\_edges) {  grid[edge[0]][edge[1]].insert(1);  }  for (auto edge : blue\_edges) {  grid[edge[0]][edge[1]].insert(2);  }  vector<int> res(n, -1);  set<pair<int, int>> visted;  queue<vector<int>> q;  res[0] = 0;  for (const auto& nodes : grid[0]) {  if (nodes.first != 0) {  res[nodes.first] = 1;  for (auto color : nodes.second) {  q.push({nodes.first, color, 1});  visted.insert({ nodes.first, color });  }  }  }  while (!q.empty()) {  const auto node = q.front();  int curIndex = node[0];  int curColor = node[1];  int step = node[2];  q.pop();  for (const auto& nodes : grid[curIndex]) {  for (auto color : nodes.second) {  if (color != curColor && visted.count({ nodes.first, color }) == 0) {  visted.insert({ nodes.first, color });  q.push({ nodes.first, color, step + 1 });  if (res[nodes.first] == -1) {  res[nodes.first] = step + 1;  }  }  }  }  }  return res;  }  }; |

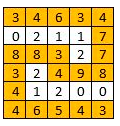
**1102. 得分最高的路径**

给你一个 R 行 C 列的整数矩阵 A。矩阵上的路径从 [0,0] 开始，在 [R-1,C-1] 结束。

路径沿四个基本方向（上、下、左、右）展开，从一个已访问单元格移动到任一相邻的未访问单元格。

路径的得分是该路径上的 最小 值。例如，路径 8 → 4 → 5 → 9 的值为 4 。

找出所有路径中得分 最高 的那条路径，返回其得分。



输入：[[5,4,5],[1,2,6],[7,4,6]]

输出：4



输入：[[2,2,1,2,2,2],[1,2,2,2,1,2]]

输出：2

输入：[[3,4,6,3,4],[0,2,1,1,7],

[8,8,3,2,7],[3,2,4,9,8],

[4,1,2,0,0],[4,6,5,4,3]]

输出：3

|  |
| --- |
| class Solution {  public:  int maximumMinimumPath(vector<vector<int>>& A) {  int R = A.size();  int C = A[0].size();  vector<vector<int> > visited(R, vector<int>(C, false));  visited[0][0] = true;  priority\_queue<Point> pq;  pq.push(Point(0, 0, A[0][0]));  int res = min(A[0][0], A[R - 1][C - 1]);  while (!pq.empty()) {  Point p = pq.top();  pq.pop();  for (int i = 0; i < 4; ++i) {  int r = p.x + dirs[i][0];  int c = p.y + dirs[i][1];  if (r >= 0 && r < R && c >= 0 && c < C && !visited[r][c]) {  res = min(res, p.val);  if (r == R - 1 && c == C - 1) return res;  visited[r][c] = true;  pq.push(Point(r, c, A[r][c]));  }  }  }  return res;  }  private:  struct Point {  int x, y, val;  Point(int \_x, int \_y, int \_val) : x(\_x), y(\_y), val(\_val) {}  bool operator < (const Point& other) const {  return this->val < other.val;  }  };  int dirs[4][2] = {{-1, 0}, {1, 0}, {0, 1}, {0, -1}};  }; |

**909. 蛇梯棋**

N x N 的棋盘 board 上，按从 1 到 N\*N 的数字给方格编号，编号 从左下角开始，每一行交替方向。例如，一块 6 x 6 大小的棋盘

r 行 c 列的棋盘，按前述方法编号，棋盘格中可能存在 “蛇” 或 “梯子”；如果 board[r][c] != -1，那个蛇或梯子的目的地将会是 board[r][c]。

玩家从棋盘上的方格 1 （总是在最后一行、第一列）开始出发。

每一回合，玩家需要从当前方格 x 开始出发，按下述要求前进：

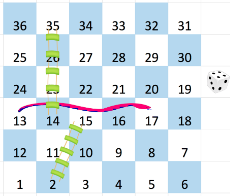
选定目标方格：选择从编号 x+1，x+2，x+3，x+4，x+5，或者 x+6 的方格中选出一个目标方格 s ，目标方格的编号 <= N\*N。

该选择模拟了掷骰子的情景，无论棋盘大小如何，你的目的地范围也只能处于区间 [x+1, x+6] 之间。

传送玩家：如果目标方格 S 处存在蛇或梯子，那么玩家会传送到蛇或梯子的目的地。否则，玩家传送到目标方格 S。

注意: 玩家在每回合的前进过程中最多只能爬过蛇或梯子一次：就算目的地是另一条蛇或梯子的起点，你也不会继续移动。

返回达到方格 N\*N 所需的最少移动次数，如果不可能，则返回 -1。

输入：[

[-1,-1,-1,-1,-1,-1],

[-1,-1,-1,-1,-1,-1],

[-1,-1,-1,-1,-1,-1],

[-1,35,-1,-1,13,-1],

[-1,-1,-1,-1,-1,-1],

[-1,15,-1,-1,-1,-1]]

输出：4

解释：

首先，从方格 1 [第 5 行，第 0 列] 开始。

你决定移动到方格 2，并必须爬过梯子移动到到方格 15。

然后你决定移动到方格 17 [第 3 行，第 5 列]，必须爬过蛇到方格 13。

然后你决定移动到方格 14，且必须通过梯子移动到方格 35。

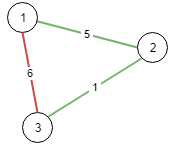
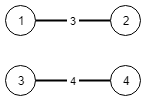
然后你决定移动到方格 36, 游戏结束。

可以证明你需要至少 4 次移动才能到达第 N\*N 个方格，所以答案是 4。

|  |
| --- |
| class Solution {  public:  int snakesAndLadders(vector<vector<int>>& board) {  int n = board.size();  int dstIndex = n \* n;  map<int, int> visitedStep;  visitedStep[1] = {0};  queue<int> q;  q.push(1);  while (!q.empty()) {  auto start = q.front();  q.pop();  int step = visitedStep[start];  if (start >= dstIndex) {  return step;  }  for (int i = start + 1; i <= min(start + 6, dstIndex); ++i) {  auto cord = GetCord(i, n);  int val = board[cord.first][cord.second];  int newStart = val;  if (val == -1) {  newStart = i;  } else {  newStart = val;  }  if (visitedStep.count(newStart) == 0) {  q.push(newStart);  visitedStep.insert({newStart, step +1});  // printf("push fast: %d [%d][%d]\n", newStart, cord.first, cord.second);  }  }  }  return -1;  }  pair<int, int> GetCord(int index, int n)  {  int y = (n - 1) - (index - 1) / n;  int x = 0;  if (((index - 1) / n) % 2 == 0) {  x = (index - 1) % n;  }  else {  x = (n - 1) - (index - 1) % n;  }  return { y, x };  }  } |

1135. 最低成本联通所有城市

想象一下你是个城市基建规划者，地图上有 N 座城市，它们按以 1 到 N 的次序编号。给你一些可连接的选项 conections，其中每个选项 conections[i] = [city1, city2, cost] 表示将城市 city1 和城市 city2 连接所要的成本。（连接是双向的，也就是说城市 city1 和城市 city2 相连也同样意味着城市 city2 和城市 city1 相连）。返回使得每对城市间都存在将它们连接在一起的连通路径（可能长度为 1 的）最小成本。该最小成本应该是所用全部连接代价的综合。如果根据已知条件无法完成该项任务，则请你返回 -1。

输入：N = 3, conections = [[1,2,5],[1,3,6],[2,3,1]]

输出：6

解释：选出任意 2 条边都可以连接所有城市，我们从中选取成本最小的 2 条。

输入：N = 4, conections = [[1,2,3],[3,4,4]]

输出：-1

解释：即使连通所有的边，也无法连接所有城市。

|  |
| --- |
| class Solution {  public:  vector<unordered\_map<int,int>> graph;  int minimumCost(int N, vector<vector<int>>& conections) {  graph.resize(N + 1);  for(int i = 0; i < conections.size();i++) {  int v1 = conections[i][0];  int v2 = conections[i][1];  int val = conections[i][2];  if(graph[v1].find(v2) != graph[v1].end()) {  graph[v1][v2] = min(graph[v1][v2], val);  graph[v2][v1] = min(graph[v2][v1], val);  } else {  graph[v1][v2] = val;  graph[v2][v1] = val;  }  }  priority\_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>> pq;  for(int i = 1;i <= N;i++) {  if(graph[i].size() == 0) {  return -1;  }  }  vector<bool> visit(N + 1, false);  pq.push({0,1});  int res = 0;  int count = 0;  while(!pq.empty()) {  auto cur = pq.top();  pq.pop();  if(visit[cur.second]) continue;  res += cur.first;  visit[cur.second] = true;  count++;  if(count == N) break;  for(auto& neighbor : graph[cur.second]) {  if(!visit[neighbor.first])  {  pq.push({neighbor.second, neighbor.first});  }  }  }  if(count != N) return -1;  return res;  }  }; |

**1101. 彼此熟识的最早时间（排序+并查集）**

在一个社交圈子当中，有 N 个人。每个人都有一个从 0 到 N-1 唯一的 id 编号。我们有一份日志列表 logs，其中每条记录都包含一个非负整数的时间戳，以及分属两个人的不同 id，logs[i] = [timestamp, id\_A, id\_B]。每条日志标识出两个人成为好友的时间，友谊是相互的：如果 A 和 B 是好友，那么 B 和 A 也是好友。如果 A 是 B 的好友，或者 A 是 B 的好友的好友，那么就可以认为 A 也与 B 熟识。返回圈子里所有人之间都熟识的最早时间。如果找不到最早时间，就返回 -1 。

输入：logs = [[20190101,0,1],[20190104,3,4],[20190107,2,3],[20190211,1,5],

[20190224,2,4],[20190301,0,3],[20190312,1,2],[20190322,4,5]],

N = 6

输出：20190301

解释：

第一次结交发生在 timestamp = 20190101，0 和 1 成为好友，

社交朋友圈如下 [0,1], [2], [3], [4], [5]。

第二次结交发生在 timestamp = 20190104，3 和 4 成为好友，

社交朋友圈如下 [0,1], [2], [3,4], [5].

第三次结交发生在 timestamp = 20190107，2 和 3 成为好友，

社交朋友圈如下 [0,1], [2,3,4], [5].

第四次结交发生在 timestamp = 20190211，1 和 5 成为好友，

社交朋友圈如下 [0,1,5], [2,3,4].

第五次结交发生在 timestamp = 20190224，2 和 4 已经是好友了。

第六次结交发生在 timestamp = 20190301，0 和 3 成为好友，大家都互相熟识了。

提示：

1 <= N <= 100

1 <= logs.length <= 10^4

0 <= logs[i][0] <= 10^9

0 <= logs[i][1], logs[i][2] <= N - 1

保证 logs[i][0] 中的所有时间戳都不同 Logs 不一定按某一标准排序 logs[i][1] != logs[i][2]

|  |
| --- |
| class UF {  vector<int> f;  public:  UF(int n) {  f.resize(n);  for(int i = 0; i < n; ++i)  f[i] = i;  }  void merge(int a, int b) {  int fa = find(a);  int fb = find(b);  f[fa] = fb;  }  int find(int a) {  int origin = a;  while(a != f[a])  a = f[a];  return f[origin] = a;  }  bool onlyOne() {  int count = 0;  for(int i = 0; i < f.size(); ++i)  {  if(i == find(i))  count++;  if(count > 1)  return false;  }  return true;  }  };  class Solution {  public:  int earliestAcq(vector<vector<int>>& logs, int N) {  sort(logs.begin(), logs.end(),[&](auto a, auto b){  return a[0] < b[0];  });  UF u(N);  for(auto& lg : logs)  {  u.merge(lg[1], lg[2]);  if(u.onlyOne())  return lg[0];  }  return -1;  }  }; |

**53. 最大子序和**

给定一个整数数组 nums ，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

输入: [-2,1,-3,4,-1,2,1,-5,4]

输出: 6

解释: 连续子数组 [4,-1,2,1] 的和最大，为 6。

|  |
| --- |
| class Solution {  public:  int maxSubArray(vector<int>& nums) {  if (nums.empty()) {  return 0;  }  maxSum = nums[0];  divide(nums, nums.size() - 1);  return maxSum;  }  int divide(vector<int>& nums, int index) {  if (index < 0) {  return 0;  }  int sum = max(divide(nums, index - 1) + nums[index], nums[index]);  maxSum = max(maxSum, sum);  return sum;  }  private:  int maxSum = 0;  }; |

**95. 不同的二叉搜索树 II**

给定一个整数 n，生成所有由 1 ... n 为节点所组成的 二叉搜索树 。

输入：3

输出：

[

[1,null,3,2],

[3,2,null,1],

[3,1,null,null,2],

[2,1,3],

[1,null,2,null,3]

]

解释：

以上的输出对应以下 5 种不同结构的二叉搜索树：

1 3 3 2 1

\ / / / \ \

3 2 1 1 3 2

/ / \ \

2 1 2 3

提示：

0 <= n <= 8

|  |
| --- |
| class Solution {  public:  vector<TreeNode\*> generateTrees(int n) {  vector<int> nums(n, 0);  iota(nums.begin(), nums.end(), 1);  return generateTrees(std::move(nums));  }  vector<TreeNode\*> generateTrees(vector<int> nums) {  if (nums.empty()) {  return {};  }  if (nums.size() == 1) {  return { new TreeNode(nums[0]) };  }  vector<TreeNode\*> res;  for (size\_t i = 0; i < nums.size(); i++) {  auto leftTree = generateTrees(std::move(vector<int>(nums.begin(), nums.begin() + i)));  auto rightTree = generateTrees(std::move(vector<int>(nums.begin() + i + 1, nums.end())));  if (leftTree.empty()) {  leftTree.push\_back(nullptr);  }  if (rightTree.empty()) {  rightTree.push\_back(nullptr);  }  for (auto left : leftTree) {  for (auto right : rightTree) {  TreeNode\* node = new TreeNode(nums[i]);  node->left = left;  node->right = right;  res.push\_back(node);  }  }  }  return res;  }  }; |

**1143. 最长公共子序列**

给定两个字符串 text1 和 text2，返回这两个字符串的最长公共子序列的长度。一个字符串的 子序列是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。

例如，"ace" 是 "abcde" 的子序列，但 "aec" 不是 "abcde" 的子序列。两个字符串的「公共子序列」是这两个字符串所共同拥有的子序列。若这两个字符串没有公共子序列，返回 0。

输入：text1 = "abcde", text2 = "ace"

输出：3 解释：最长公共子序列是 "ace"，它的长度为 3。

输入：text1 = "abc", text2 = "abc"

输出：3 解释：最长公共子序列是 "abc"，它的长度为 3。

输入：text1 = "abc", text2 = "def"

输出：0 解释：两个字符串没有公共子序列，返回 0。

|  |
| --- |
| class Solution {  public:  int longestCommonSubsequence(string text1, string text2) {  int n = text1.size();  int m = text2.size();  vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));  for (int i = 0; i < n; ++i) {  for (int j = 0; j < m; ++j) {  dp[i + 1][j + 1] = max(dp[i][j + 1], dp[i + 1][j]);  if (text1[i] == text2[j]) {  dp[i + 1][j + 1] = max(dp[i + 1][j + 1], dp[i][j] + 1);  }  }  }  return dp[n][m];  }  }; |