第三周

[50. Pow(x, n)](https://leetcode-cn.com/problems/powx-n/)

实现 pow(x, n) ，即计算 x 的 n 次幂函数。

示例 1:

输入: 2.00000, 10

输出: 1024.00000

示例 2:

输入: 2.10000, 3

输出: 9.26100

示例 3:

输入: 2.00000, -2

输出: 0.25000

解释: 2-2 = 1/22 = 1/4 = 0.25

链接：<https://leetcode-cn.com/problems/powx-n>

方法 1：蛮力

直观想法

直接模拟该过程，将 x 连乘 n 次。

如果 n < 0，我们可以用1/x,−n 代替 x,n 来保证 n ≥0 。这个限制可以简化我们下面的讨论。

但是我们仍需关注边界条件，尤其是正整数和负整数的不同范围限制

class Solution {

public double myPow(double x, int n) {

long N = n;

if (N < 0) {

x = 1 / x;

N = -N;

}

double ans = 1;

for (long i = 0; i < N; i++)

ans = ans \* x;

return ans;

}

};

算法 时间复杂度： *O*(log*n*) 空间复杂度： *O*(log*n*)

假定我们已经得到了x^n/2的结果，并且我们现在想得到x^n的结果。我们令 A 是x^n/2的结果，我们可以根据 n 的奇偶性来分别讨论 x^n的值。如果 n 为偶数，我们可以用公式 (x^n)^2 = x^{2 \* n}来得到 x^n = A\*A,如果 n 为奇数，那么 A\*A=x^{n-1}。直观上看，我们需要再乘一次 x，即 x^n=A∗A∗x。该方法可以很方便的使用递归实现。我们称这种方法为 "快速幂"，因为我们只需最多O(logn) 次运算来得到 x^n。

class Solution {

public double myPow(double x, int n) {

if (n < 0) {

x = 1 / x;

n = -n;

}

return recursive(x, n);

}

private double recursive(double x, int n) {

// recursive terminator

if (n == 0) {

return 1.0;

}

// process on

double res = recursive(x, n / 2);

// drill down

if (n % 2 == 0) {

return res \* res;

} else {

return res \* res \* x;

}

// reverse status

}

}

* 时间复杂度*O*(log*n*). 空间复杂的：O(1)

class Solution {

public double myPow(double x, int n) {

long N = n;

if (N < 0) {

x = 1 / x;

N = -N;

}

double ans = 1; current\_product = x;

for (long i = N; i > 0; i /= 2) {

if ((i % 2) == 1) {

ans = ans \* current\_product;

}

current\_product = current\_product \* current\_product;

}

return ans;

}

};

#### [78. 子集](https://leetcode-cn.com/problems/subsets/)

给定一组不含重复元素的整数数组 nums，返回该数组所有可能的子集（幂集）。

说明：解集不能包含重复的子集。

示例:

输入: nums = [1,2,3]

输出:

[

[3],

  [1],

  [2],

  [1,2,3],

  [1,3],

  [2,3],

  [1,2],

  []

]

链接：<https://leetcode-cn.com/problems/subsets>

方法一：回溯

复杂度分析

时间复杂度O(N×2^ N )，生成所有子集，并复制到输出集合中。

空间复杂度：O(N×2^ N )，存储所有子集，共 n 个元素，每个元素都有可能存在或者不存在。

class Solution {

public List<List<Integer>> subsets(int[] nums) {

List<List<Integer>> resList = new ArrayList<>();

backRack(nums, resList, new ArrayList<>(), 0);

return resList;

}

private void backRack(int[] nums, List<List<Integer>> resList, List<Integer> tempList, int index) {

resList.add(new ArrayList<>(tempList));

for (int i = index; i < nums.length; i++) {

tempList.add(nums[i]);

backRack(nums, resList, tempList, i + 1);

// 移除最后一个，后退一步

tempList.remove(tempList.size() - 1);

}

}

}

方法二：

逐个枚举，空集的幂集只有空集，每增加一个元素，让之前幂集中的每个集合，追加这个元素，就是新增的子集。

/\*\*

\* 循环枚举

\*/

public static List<List<Integer>> enumerate(int[] nums) {

List<List<Integer>> res = new ArrayList<List<Integer>>();

res.add(new ArrayList<Integer>());

for (Integer n : nums) {

int size = res.size();

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

List<Integer> newSub = new ArrayList<Integer>(res.get(i));

newSub.add(n);

res.add(newSub);

}

}

return res;

}

/\*\*

\* 递归枚举

\*/

public static void recursion(int[] nums, int i, List<List<Integer>> res) {

if (i >= nums.length) return;

int size = res.size();

for (int j = 0; j < size; j++) {

List<Integer> newSub = new ArrayList<Integer>(res.get(j));

newSub.add(nums[i]);

res.add(newSub);

}

recursion(nums, i + 1, res);

}

#### [169. 多数元素](https://leetcode-cn.com/problems/majority-element/) 给定一个大小为 n 的数组，找到其中的多数元素。多数元素是指在数组中出现次数**大于** ⌊ n/2 ⌋ 的元素。你可以假设数组是非空的，并且给定的数组总是存在多数元素。

**示例 1:**

**输入:** [3,2,3]

**输出:** 3

**示例 2:**

**输入:** [2,2,1,1,1,2,2]

**输出:** 2

方法 1：暴力

想法

我们可以在平方级的时间里穷举所有情况，来检测每个数是不是众数。

算法

暴力算法遍历整个数组，然后用另一重循环统计每个数字出现的次数。将出现次数比其他数字加起来出现次数还多的元素返回。

class Solution {

public int majorityElement(int[] nums) {

int majorityCount = nums.length/2;

for (int num : nums) {

int count = 0;

for (int elem : nums) {

if (elem == num) {

count += 1;

}

}

if (count > majorityCount) {

return num;

}

}

return -1;

}

}

复杂度分析

时间复杂度：O(n^2)

暴力算法包含两重嵌套的 for 循环，每一层n次迭代，所以总的是平方级的时间复杂度。

空间复杂度：O(1)

暴力方法没有分配任何与输入规模成比例的额外的空间

方法 2：哈希表

想法

我们知道出现次数最多的元素大于 [n/2] 次，所以可以用哈希表来快速统计每个元素出现的次数。

算法

我们使用哈希表来存储每个元素，然后用一个循环在线性时间内遍历 nums ，然后我们只需要返回有最大值的键。

public int majorityElement(int[] nums) {

int maxMajority = nums.length / 2;

// key count

Map<Integer, Integer> keyMap = new HashMap<>();

for (int i = 0; i < nums.length; i++) {

int nowCount = keyMap.get(nums[i]) == null ? 1 : 1 + keyMap.get(nums[i]);

if (nowCount > maxMajority) {

return nums[i];

}

keyMap.put(nums[i], nowCount);

}

return 0;

}

复杂度分析

时间复杂度：O(n)

我们将 nums 迭代一次，哈希表的插入是常数时间的。所以总时间复杂度为 O(n) 时间的。

空间复杂度：O(n)

哈希表最多包含 n-[n/2]个关系，所以占用的空间为 O(n)。这是因为任意一个长度为 n 的数组最多只能包含 nn个不同的值，但题中保证 nums 一定有一个众数，会占用（最少）[n/2]+1 个数字。因此最多有 n - ([n/2]+1) 个不同的其他数字，所以最多有 n -[n/2]个不同的元素。

方法 3：排序

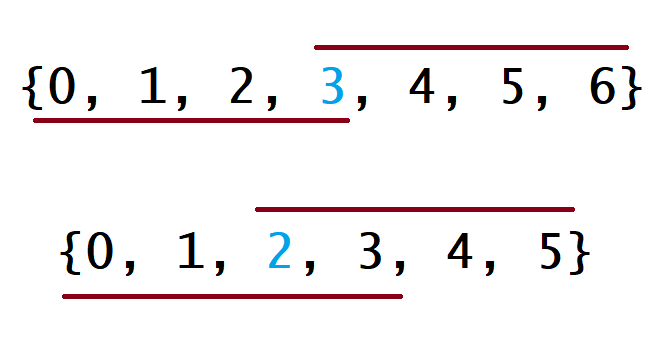
想法

如果所有数字被单调递增或者单调递减的顺序排了序，那么众数的下标为[n/2]

（当 nn 是偶数时，下标为 [n/2]+1 ）

算法

对于这种算法，我们先将 nums 数组排序，然后返回上面所说的数字。下面解释了为什么这种策略是有效的。考虑下图（上面的例子是一个可能的奇数的情况，下面的例子是一个可能的偶数的情况）：



对于每种情况，数组下面的线表示如果众数是数组中最小值的情况下覆盖的下标。数组上面的线是数组中最大值的情况。其他情况，这条线会在这两种极端情况的中间。但我们看到即使是这两种极端情况，它们也会在下标为[n/2] 的地方有重叠。因此，无论众数是多少，返回 [n/2]下标对应的值都是正确的。

class Solution {

public int majorityElement(int[] nums) {

Arrays.sort(nums);

return nums[nums.length/2];

}

}

复杂度分析

时间复杂度：O(nlgn)

用 Python 和 Java 将数组排序开销都为 O(nlgn)，它占据了运行的主要时间。

空间复杂度：O(1) 或者 O(n)

我们将 nums 就地排序，如果不能就低排序，我们必须使用线性空间将 nums 数组拷贝，然后再排序。

方法 6：Boyer-Moore 投票算法

想法

如果我们把众数记为 +1+1 ，把其他数记为 -1−1 ，将它们全部加起来，显然和大于 0 ，从结果本身我们可以看出众数比其他数多。

算法

本质上， Boyer-Moore 算法就是找 nums 的一个后缀 sufsuf ，其中 suf[0]suf[0] 就是后缀中的众数。我们维护一个计数器，如果遇到一个我们目前的候选众数，就将计数器加一，否则减一。只要计数器等于 0 ，我们就将 nums 中之前访问的数字全部 忘记 ，并把下一个数字当做候选的众数。直观上这个算法不是特别明显为何是对的，我们先看下面这个例子（竖线用来划分每次计数器归零的情况）

[7, 7, 5, 7, 5, 1 | 5, 7 | 5, 5, 7, 7 | 7, 7, 7, 7]

首先，下标为 0 的 7 被当做众数的第一个候选。在下标为 5 处，计数器会变回0 。所以下标为 6 的 5 是下一个众数的候选者。由于这个例子中 7 是真正的众数，所以通过忽略掉前面的数字，我们忽略掉了同样多数目的众数和非众数。因此， 7 仍然是剩下数字中的众数。

[7, 7, 5, 7, 5, 1 | 5, 7 | 5, 5, 7, 7 | 5, 5, 5, 5]

现在，众数是 5 （在计数器归零的时候我们把候选从 7 变成了 5）。此时，我们的候选者并不是真正的众数，但是我们在 遗忘 前面的数字的时候，要去掉相同数目的众数和非众数（如果遗忘更多的非众数，会导致计数器变成负数）。

因此，上面的过程说明了我们可以放心地遗忘前面的数字，并继续求解剩下数字中的众数。最后，总有一个后缀满足计数器是大于 0 的，此时这个后缀的众数就是整个数组的众数。

class Solution {

public int majorityElement(int[] nums) {

int count = 0;

Integer candidate = null;

for (int num : nums) {

if (count == 0) {

candidate = num;

}

count += (num == candidate) ? 1 : -1;

}

return candidate;

}

}

#### [17. 电话号码的字母组合](https://leetcode-cn.com/problems/letter-combinations-of-a-phone-number/)

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。



示例:

输入："23"

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

方法：回溯

回溯是一种通过穷举所有可能情况来找到所有解的算法。如果一个候选解最后被发现并不是可行解，回溯算法会舍弃它，并在前面的一些步骤做出一些修改，并重新尝试找到可行解。

给出如下回溯函数 backtrack(combination, next\_digits) ，它将一个目前已经产生的组合 combination 和接下来准备要输入的数字 next\_digits 作为参数。

如果没有更多的数字需要被输入，那意味着当前的组合已经产生好了。

如果还有数字需要被输入：

遍历下一个数字所对应的所有映射的字母。

将当前的字母添加到组合最后，也就是 combination = combination + letter 。

重复这个过程，输入剩下的数字： backtrack(combination + letter, next\_digits[1:]) 。

方法一：回溯

复杂度分析：

时间复杂度：O(3^N × 4^M)，N 为电话号码中拥有 3 个字母的按键个数，M 为电话号码中拥有 4个字母的按键个数，。

空间复杂度：O(3^N × 4^M)，同上。

public class Leetcode\_17\_566 {

/\*\*

\* 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。

\*/

static Map<String, String> phoneCharMap = new HashMap<>();

List<String> resList = new ArrayList<>();

Leetcode\_17\_566() {

phoneCharMap.put("2", "abc");

phoneCharMap.put("3", "def");

phoneCharMap.put("4", "ghi");

phoneCharMap.put("5", "jkl");

phoneCharMap.put("6", "mno");

phoneCharMap.put("7", "pqrs");

phoneCharMap.put("8", "tuv");

phoneCharMap.put("9", "wxyz");

}

public static void main(String[] args) {

Leetcode\_17\_566 code=new Leetcode\_17\_566();

List<String> stringList = code.letterCombinations("23");

}

public List<String> letterCombinations(String digits) {

if (digits.length() != 0) {

backTrack(digits, "");

}

return resList;

}

/\*\*

\* 当前操作的字符数字

\*

\* @param digits

\* @param nowChar

\*/

private void backTrack(String digits, String nowChar) {

if (digits.length() == 0) {

resList.add(nowChar);

return;

}

// 获取第一个数字

String substringChar = digits.substring(0, 1);

// 字符获取

String charString = phoneCharMap.get(substringChar);

for (int i = 0; i < charString.length(); i++) {

String letter = phoneCharMap.get(substringChar).substring(i, i + 1);

backTrack(digits.substring(1), nowChar + letter);

}

}

}

#### [51. N皇后](https://leetcode-cn.com/problems/n-queens/)

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。

链接：https://leetcode-cn.com/problems/n-queens

示例:

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。

方法1：回溯

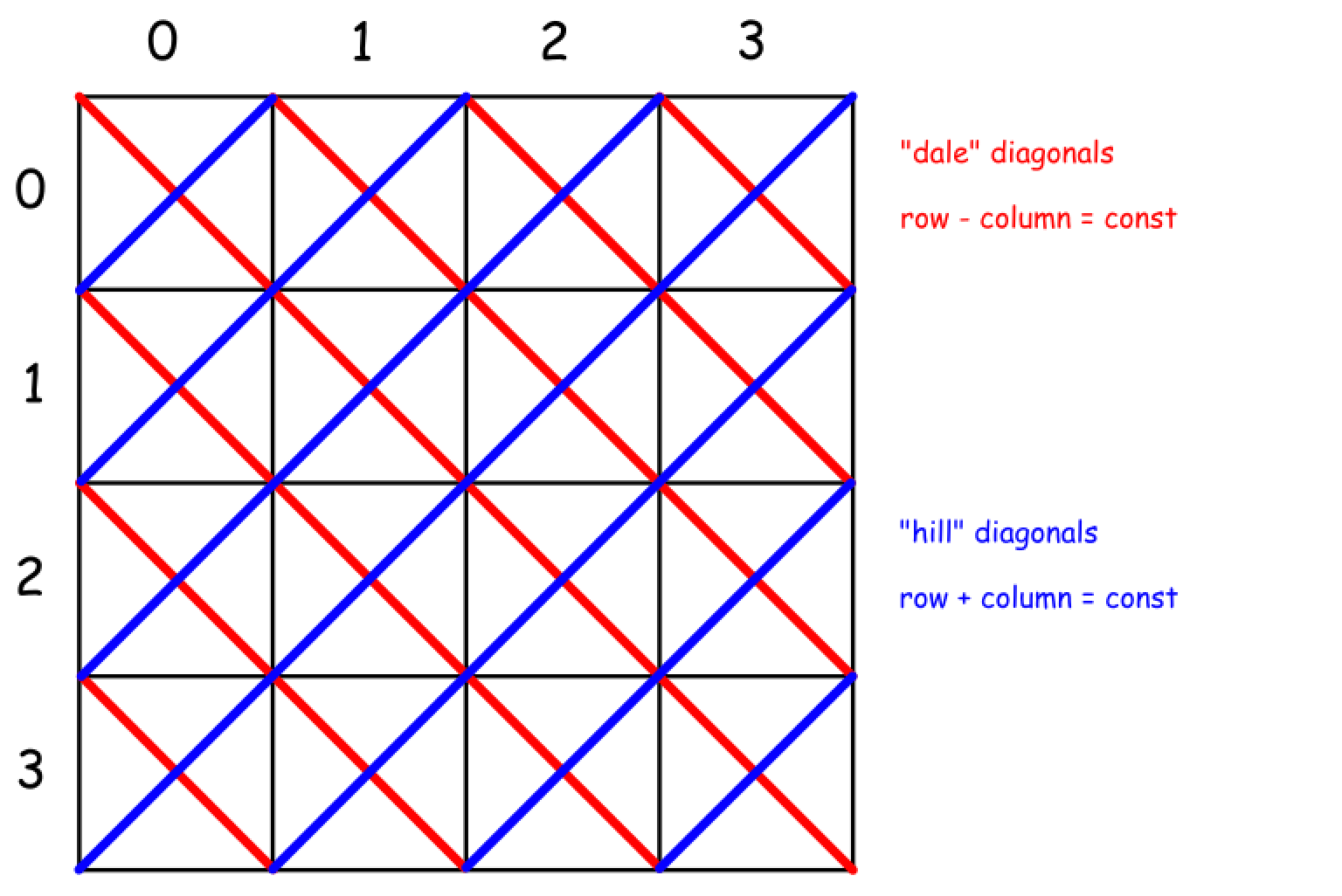
在建立算法之前，我们来考虑两个有用的细节。

一行只可能有一个皇后且一列也只可能有一个皇后。

这意味着没有必要再棋盘上考虑所有的方格。只需要按列循环即可。

对于所有的主对角线有 行号 + 列号 = 常数，对于所有的次对角线有 行号 - 列号 = 常数.

这可以让我们标记已经在攻击范围下的对角线并且检查一个方格 (行号, 列号) 是否处在攻击位置。



现在已经可以写回溯函数 backtrack(row = 0).

从第一个 row = 0 开始.

循环列并且试图在每个 column 中放置皇后.

如果方格 (row, column) 不在攻击范围内

在 (row, column) 方格上放置皇后。

排除对应行，列和两个对角线的位置。

If 所有的行被考虑过，row == N

意味着我们找到了一个解

Else

继续考虑接下来的皇后放置 backtrack(row + 1).

回溯：将在 (row, column) 方格的皇后移除.

下面是上述算法的一个直接的实现。

class Solution {

int rows[];

// "hill" diagonals

int hills[];

// "dale" diagonals

int dales[];

int n;

// output

List<List<String>> output = new ArrayList();

// queens positions

int queens[];

public boolean isNotUnderAttack(int row, int col) {

int res = rows[col] + hills[row - col + 2 \* n] + dales[row + col];

return (res == 0) ? true : false;

}

public void placeQueen(int row, int col) {

queens[row] = col;

rows[col] = 1;

hills[row - col + 2 \* n] = 1; // "hill" diagonals

dales[row + col] = 1; //"dale" diagonals

}

public void removeQueen(int row, int col) {

queens[row] = 0;

rows[col] = 0;

hills[row - col + 2 \* n] = 0;

dales[row + col] = 0;

}

public void addSolution() {

List<String> solution = new ArrayList<String>();

for (int i = 0; i < n; ++i) {

int col = queens[i];

StringBuilder sb = new StringBuilder();

for(int j = 0; j < col; ++j) sb.append(".");

sb.append("Q");

for(int j = 0; j < n - col - 1; ++j) sb.append(".");

solution.add(sb.toString());

}

output.add(solution);

}

public void backtrack(int row) {

for (int col = 0; col < n; col++) {

if (isNotUnderAttack(row, col)) {

placeQueen(row, col);

// if n queens are already placed

if (row + 1 == n) addSolution();

// if not proceed to place the rest

else backtrack(row + 1);

// backtrack

removeQueen(row, col);

}

}

}

public List<List<String>> solveNQueens(int n) {

this.n = n;

rows = new int[n];

hills = new int[4 \* n - 1];

dales = new int[2 \* n - 1];

queens = new int[n];

backtrack(0);

return output;

}

}

#### [102. 二叉树的层次遍历](https://leetcode-cn.com/problems/binary-tree-level-order-traversal/)

给定一个二叉树，返回其按层次遍历的节点值。 （即逐层地，从左到右访问所有节点）。

例如:

给定二叉树: [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]

]

链接：https://leetcode-cn.com/problems/binary-tree-level-order-traversal

方法 1：递归

算法

最简单的解法就是递归，首先确认树非空，然后调用递归函数 helper(node, level)，参数是当前节点和节点的层次。程序过程如下：

输出列表称为 levels，当前最高层数就是列表的长度 len(levels)。比较访问节点所在的层次 level 和当前最高层次 len(levels) 的大小，如果前者更大就向 levels 添加一个空列表。

将当前节点插入到对应层的列表 levels[level] 中。

递归非空的孩子节点：helper(node.left / node.right, level + 1)。

class Solution {

List<List<Integer>> levels = new ArrayList<List<Integer>>();

public void helper(TreeNode node, int level) {

// start the current level

if (levels.size() == level)

levels.add(new ArrayList<Integer>());

// fulfil the current level

levels.get(level).add(node.val);

// process child nodes for the next level

if (node.left != null)

helper(node.left, level + 1);

if (node.right != null)

helper(node.right, level + 1);

}

public List<List<Integer>> levelOrder(TreeNode root) {

if (root == null) return levels;

helper(root, 0);

return levels;

}

}

复杂度分析

时间复杂度：O(N)，因为每个节点恰好会被运算一次。

空间复杂度：O(N)，保存输出结果的数组包含 N 个节点的值。

方法 2：迭代

算法

上面的递归方法也可以写成迭代的形式。

我们将树上顶点按照层次依次放入队列结构中，队列中元素满足 FIFO（先进先出）的原则。在 Java 中可以使用 Queue 接口中的 LinkedList实现。在 Python 中如果使用 Queue 结构，但因为它是为多线程之间安全交换而设计的，所以使用了锁，会导致性能不佳。因此在 Python 中可以使用 deque 的 append() 和 popleft() 函数来快速实现队列的功能。

第 0 层只包含根节点 root ，算法实现如下：

初始化队列只包含一个节点 root 和层次编号 0 ： level = 0。

当队列非空的时候：

在输出结果 levels 中插入一个空列表，开始当前层的算法。

计算当前层有多少个元素：等于队列的长度。

将这些元素从队列中弹出，并加入 levels 当前层的空列表中。

将他们的孩子节点作为下一层压入队列中。

进入下一层 level++。

class Solution {

public List<List<Integer>> levelOrder(TreeNode root) {

List<List<Integer>> levels = new ArrayList<List<Integer>>();

if (root == null) return levels;

Queue<TreeNode> queue = new LinkedList<TreeNode>();

queue.add(root);

int level = 0;

while ( !queue.isEmpty() ) {

// start the current level

levels.add(new ArrayList<Integer>());

// number of elements in the current level

int level\_length = queue.size();

for(int i = 0; i < level\_length; ++i) {

TreeNode node = queue.remove();

// fulfill the current level

levels.get(level).add(node.val);

// add child nodes of the current level

// in the queue for the next level

if (node.left != null) queue.add(node.left);

if (node.right != null) queue.add(node.right);

}

// go to next level

level++;

}

return levels;

}

}

// bfs

class Solution {

public List<List<Integer>> levelOrder(TreeNode root) {

Queue<TreeNode> queue = new LinkedList<TreeNode>();

List<List<Integer>> res = new LinkedList<>();

if (root == null) return res;

queue.offer(root);

while (! queue.isEmpty()){

int size = queue.size();

List<Integer> level = new LinkedList<Integer>();

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

if (queue.peek().left != null) queue.offer(queue.peek().left);

if (queue.peek().right != null) queue.offer(queue.peek().right);

level.add(queue.poll().val);

}

res.add(level);

}

return res;

}

}

// dfs

class Solution {

public List<List<Integer>> levelOrder(TreeNode root) {

List<List<Integer>> res = new LinkedList<>();

dfs(res, root, 0);

return res;

}

private void dfs(List<List<Integer>> res, TreeNode root, int depth) {

if (root == null) return;

if (depth >= res.size())

res.add(new LinkedList<>());

res.get(depth).add(root.val);

dfs(res, root.left, depth+1);

dfs(res, root.right, depth+1);

}

}

#### [433. 最小基因变化](https://leetcode-cn.com/problems/minimum-genetic-mutation/)

一条基因序列由一个带有8个字符的字符串表示，其中每个字符都属于 "A", "C", "G", "T"中的任意一个。

假设我们要调查一个基因序列的变化。一次基因变化意味着这个基因序列中的一个字符发生了变化。

例如，基因序列由"AACCGGTT" 变化至 "AACCGGTA" 即发生了一次基因变化。

与此同时，每一次基因变化的结果，都需要是一个合法的基因串，即该结果属于一个基因库。

现在给定3个参数 — start, end, bank，分别代表起始基因序列，目标基因序列及基因库，请找出能够使起始基因序列变化为目标基因序列所需的最少变化次数。如果无法实现目标变化，请返回 -1。

注意:

起始基因序列默认是合法的，但是它并不一定会出现在基因库中。

所有的目标基因序列必须是合法的。

假定起始基因序列与目标基因序列是不一样的。

示例 1:

start: "AACCGGTT"

end: "AACCGGTA"

bank: ["AACCGGTA"]

返回值: 1

示例 2:

start: "AACCGGTT"

end: "AAACGGTA"

bank: ["AACCGGTA", "AACCGCTA", "AAACGGTA"]

返回值: 2

示例 3:

start: "AAAAACCC"

end: "AACCCCCC"

bank: ["AAAACCCC", "AAACCCCC", "AACCCCCC"]

返回值: 3

一般能想到的回溯的方式是逐个改变基因中的碱基(A、C、G、T)，当改变后的基因在基因库中 步数+1 并进入下一层。

但题目中说明了每一次基因变化都属于基因库，那么遍历基因库，找到库中与当前基因相差一个碱基的就是下一步变化的基因，这时步数+1并进入下一层会比上面的方式省很多。

class Solution {

int minStepCount = Integer.MAX\_VALUE;

public int minMutation(String start, String end, String[] bank) {

dfs(new HashSet<String>(), 0, start, end, bank);

return (minStepCount == Integer.MAX\_VALUE) ? -1 : minStepCount;

}

private void dfs (HashSet<String> step, int stepCount,

String current, String end, String[] bank) {

if (current.equals(end))

minStepCount = Math.min(stepCount, minStepCount);

for (String str: bank) {

int diff = 0;

for (int i = 0; i < str.length(); i++)

if (current.charAt(i) != str.charAt(i))

if (++diff > 1) break;

if (diff == 1 && !step.contains(str)) {

step.add(str);

dfs(step, stepCount + 1, str, end, bank);

step.remove(str);

}

}

}

}

BFS

public class Solution {

public int minMutation(String start, String end, String[] bank) {

if(start.equals(end)) return 0;

Set<String> bankSet = new HashSet<>();

for(String b: bank) bankSet.add(b);

char[] charSet = new char[]{'A', 'C', 'G', 'T'};

int level = 0;

Set<String> visited = new HashSet<>();

Queue<String> queue = new LinkedList<>();

queue.offer(start);

visited.add(start);

while(!queue.isEmpty()) {

int size = queue.size();

while(size-- > 0) {

String curr = queue.poll();

if(curr.equals(end)) return level;

char[] currArray = curr.toCharArray();

for(int i = 0; i < currArray.length; i++) {

char old = currArray[i];

for(char c: charSet) {

currArray[i] = c;

String next = new String(currArray);

if(!visited.contains(next) && bankSet.contains(next)) {

visited.add(next);

queue.offer(next);

}

}

currArray[i] = old;

}

}

level++;

}

return -1;

}

}

#### [515. 在每个树行中找最大值](https://leetcode-cn.com/problems/find-largest-value-in-each-tree-row/)

您需要在二叉树的每一行中找到最大的值。

示例：

输入:

1

/ \

3 2

/ \ \

5 3 9

输出: [1, 3, 9]

链接：https://leetcode-cn.com/problems/find-largest-value-in-each-tree-row

/\*\*

\* 515. 在每个树行中找最大值

\* 采用广度优先搜索遍历

\*

\* @param root

\* @return

\*/

public List<Integer> largestValues(TreeNode root) {

if (root == null) {

return resList;

}

Queue<TreeNode> queue = new LinkedList();

queue.add(root);

int queueSize = queue.size();

while (queueSize > 0) {

int maxValue = Integer.MIN\_VALUE;

for (int i = 0; i < queueSize; i++) {

TreeNode treeNode = queue.poll();

maxValue = Math.max(maxValue, treeNode.val);

// 加入下一个左节点

if (treeNode.left != null) {

queue.add(treeNode.left);

}

// 加入下一个右节点

if (treeNode.right != null) {

queue.add(treeNode.right);

}

}

// 重写队列中元素个数

queueSize = queue.size();

// 队列清空后加入最大数字

resList.add(maxValue);

}

return resList;

}

/\*\*

\* 深度优先遍历进行数据查找

\*

\* @param root

\* @return

\*/

public List<Integer> dfsLargestValue(TreeNode root) {

if (root == null) {

return resList;

}

dfs(root, 0);

return resList;

}

/\*\*

\* 深度优先遍历

\*

\* @param root 节点

\* @param level 层数

\*/

private void dfs(TreeNode root, int level) {

// terminator

if (root == null) {

return;

}

// process logic

if (resList.isEmpty() || resList.size() - 1 < level ) {

resList.add(level, root.val);

}else if(resList.get(level) < root.val){

resList.remove(level);

resList.add(level,root.val);

}

// drill down

if (root.left != null) {

dfs(root.left, level + 1);

}

if (root.right != null) {

dfs(root.right, level + 1);

}

// reverse status

}

#### [127. 单词接龙](https://leetcode-cn.com/problems/word-ladder/)

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：

1. 每次转换只能改变一个字母。
2. 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。

示例 1:

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。

示例 2:

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。

链接：<https://leetcode-cn.com/problems/word-ladder>

复杂度分析

时间复杂度O(M×N)，其中 M 是单词的长度 N 是单词表中单词的总数。找到所有的变换需要对每个单词做 M 次操作。同时，最坏情况下广度优先搜索也要访问所有的 N 个单词。

空间复杂度： O(M×N)，要在 all\_combo\_dict 字典中记录每个单词的 M 个通用状态。访问数组的大小是 N。广搜队列最坏情况下需要存储 N 个单词。

class Solution {

public int ladderLength(String beginWord, String endWord, List<String> wordList) {

// 单词的长度

int len = beginWord.length();

// 构建一个词典中所有字符的通用状态字典

HashMap<String, List<String>> allComboDict = new HashMap<>();

// 填充通用字符字典

wordList.forEach(word -> {

for (int i = 0; i < len; i++) {

String newCommonWord = word.substring(0, i) + "\*" + word.substring(i + 1, len);

List<String> transWordList = allComboDict.getOrDefault(newCommonWord, new ArrayList<>());

transWordList.add(word);

allComboDict.put(newCommonWord, transWordList);

}

});

Queue<Pair<String, Integer>> q = new LinkedList<>();

q.add(new Pair<>(beginWord, 1));

// 使用记录

HashMap<String, Boolean> visitedMap = new HashMap<>();

visitedMap.put(beginWord, true);

while (!q.isEmpty()) {

Pair<String, Integer> node = q.remove();

String word = node.getKey();

Integer level = node.getValue();

for (int i = 0; i < len; i++) {

String newBeginWord = word.substring(0, i) + "\*" + word.substring(i + 1, len);

// 获取跟当前通配字符相似的所有字典中的单词

for (String adjWord : allComboDict.getOrDefault(newBeginWord, new ArrayList<>())) {

if (adjWord.equalsIgnoreCase(endWord)) {

return level + 1;

}

// 增加访问记录，并转向下一个字符单词

if (!visitedMap.containsKey(adjWord)) {

visitedMap.put(adjWord, true);

q.add(new Pair<>(adjWord, level + 1));

}

}

}

}

return 0;

}

}

   /\*\*

     \* 所有可能的路径结果

     \*/

    List<List<String>> resList = new ArrayList<>();

    /\*\*

     \* 最短路径

     \*/

    int minDepth = Integer.MAX\_VALUE;

    /\*\*

     \* 127. 单词接龙

     \* 此方法能完成，但是存在重复路径问题，会超出时间限制

     \*

     \* @param beginWord 开始单词

     \* @param endWord   结束单词

     \* @param wordList  词典列表

     \* @return

     \*/

    public int ladderLength(String beginWord, String endWord, List<String> wordList) {

        // 先判断目标单词是否存在于列表中

        LinkedList<String> tempList = new LinkedList<>();

        if (wordList.isEmpty() || !wordList.contains(endWord)) {

            return 0;

        }

        // 将转换单词加入列表

        tempList.add(beginWord);

        // 生成全部能到达节点的路径

        backTrack(endWord, wordList, tempList);

        // 选出最小路径

        if (!resList.isEmpty()) {

            return minDepth;

        } else {

            return 0;

        }

    }

    /\*\*

     \* @param endWord  结束单词

     \* @param wordList 词典

     \* @param tempList 暂存结果集

     \*/

 private void backTrack(String endWord, List<String> wordList, LinkedList<String> tempList) {

        // 满足条件终止

        if (tempList.getLast().equalsIgnoreCase(endWord) && tempList.size() <= minDepth) {

            // 记录所有可能的路径

            ArrayList<String> pathList = new ArrayList<>(tempList);

            if(pathList.size()<minDepth){

                resList.clear();

            }

            if (pathList.size() == minDepth || resList.isEmpty()) {

                resList.add(pathList);

            }

            minDepth = pathList.size();

            return;

        }

        // 大于最短路径了，直接不走了

        if (tempList.size() > minDepth) {

            return;

        }

        // 获取整个转换列表最后一个单词

        String lastWord = tempList.getLast();

        // 挑选下一个能进行转换的单词选择列表，

        for (String word : wordList) {

            // 如果已经存在检测路径中，则跳过

            if (tempList.contains(word)) {

                continue;

            }

            // 做出选择，选择出能进行下一步转换的单词

            if (judgeCanBeChange(lastWord, word)) {

                tempList.add(word);

                //将下一个单词记入临时列表，继续排查

                backTrack(endWord, wordList, tempList);

                // 撤销选择

                tempList.removeLast();

            }

        }

    }

    /\*\*

     \* 检验能不能转换

     \*

     \* @param lastWord

     \* @param targetWord

     \*/

    private Boolean judgeCanBeChange(String lastWord, String targetWord) {

        int diff = 0;

        for (int i = 0; i < targetWord.length(); i++) {

            if (lastWord.charAt(i) != targetWord.charAt(i)) {

                if (++diff > 1) {

                    return false;

                }

            }

        }

        return true;

    }

#### [126. 单词接龙 II](https://leetcode-cn.com/problems/word-ladder-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。

转换过程中的中间单词必须是字典中的单词。

示例 1:

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"]

]

示例 2:

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。

链接：https://leetcode-cn.com/problems/word-ladder-ii

class Solution {

   /\*\*

     \* 所有可能的路径结果

     \*/

    List<List<String>> resList = new ArrayList<>();

    /\*\*

     \* 最短路径

     \*/

    int minDepth = Integer.MAX\_VALUE;

    public List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {

        // 先判断目标单词是否存在于列表中

        LinkedList<String> tempList = new LinkedList<>();

        if (wordList.isEmpty() || !wordList.contains(endWord)) {

            return resList;

        }

        // 将转换单词加入列表

        tempList.add(beginWord);

        // 生成全部能到达节点的路径

        backTrack(endWord, wordList, tempList);

        // 选出最小路径

        return resList;

    }

    /\*\*

     \* @param endWord  结束单词

     \* @param wordList 词典

     \* @param tempList 暂存结果集

     \*/

    private void backTrack(String endWord, List<String> wordList, LinkedList<String> tempList) {

        // 满足条件终止

        if (tempList.getLast().equalsIgnoreCase(endWord) && tempList.size() <= minDepth) {

            // 记录所有可能的路径

            ArrayList<String> pathList = new ArrayList<>(tempList);

            if(pathList.size()<minDepth){

                resList.clear();

            }

            if (pathList.size() == minDepth || resList.isEmpty()) {

                resList.add(pathList);

            }

            minDepth = pathList.size();

            return;

        }

        // 大于最短路径了，直接不走了

        if (tempList.size() > minDepth) {

            return;

        }

        // 获取整个转换列表最后一个单词

        String lastWord = tempList.getLast();

        // 挑选下一个能进行转换的单词选择列表，

        for (String word : wordList) {

            // 如果已经存在检测路径中，则跳过

            if (tempList.contains(word)) {

                continue;

            }

            // 做出选择，选择出能进行下一步转换的单词

            if (judgeCanBeChange(lastWord, word)) {

                tempList.add(word);

                //将下一个单词记入临时列表，继续排查

                backTrack(endWord, wordList, tempList);

                // 撤销选择

                tempList.removeLast();

            }

        }

    }

    /\*\*

     \* 检验能不能转换

     \*

     \* @param lastWord

     \* @param targetWord

     \*/

    private Boolean judgeCanBeChange(String lastWord, String targetWord) {

        int diff = 0;

        for (int i = 0; i < targetWord.length(); i++) {

            if (lastWord.charAt(i) != targetWord.charAt(i)) {

                if (++diff > 1) {

                    return false;

                }

            }

        }

        return true;

    }

}

解法二 BFS

如果理解了上边的 DFS 过程，接下来就很好讲了。上边 DFS 借助了 BFS 把所有的邻接关系保存了起来，再用 DFS 进行深度搜索。

我们可不可以只用 BFS，一边进行层次遍历，一边就保存结果。当到达结束单词的时候，就把结果存储。省去再进行 DFS 的过程。

是完全可以的，BFS 的队列就不去存储 String 了，直接去存到目前为止的路径，也就是一个 List。

public List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {

List<List<String>> ans = new ArrayList<>();

// 如果不含有结束单词，直接结束，不然后边会造成死循环

if (!wordList.contains(endWord)) {

return ans;

}

bfs(beginWord, endWord, wordList, ans);

return ans;

}

public void bfs(String beginWord, String endWord, List<String> wordList, List<List<String>> ans) {

Queue<List<String>> queue = new LinkedList<>();

List<String> path = new ArrayList<>();

path.add(beginWord);

queue.offer(path);

boolean isFound = false;

Set<String> dict = new HashSet<>(wordList);

Set<String> visited = new HashSet<>();

visited.add(beginWord);

while (!queue.isEmpty()) {

int size = queue.size();

Set<String> subVisited = new HashSet<>();

for (int j = 0; j < size; j++) {

List<String> p = queue.poll();

//得到当前路径的末尾单词

String temp = p.get(p.size() - 1);

// 一次性得到所有的下一个的节点

ArrayList<String> neighbors = getNeighbors(temp, dict);

for (String neighbor : neighbors) {

//只考虑之前没有出现过的单词

if (!visited.contains(neighbor)) {

//到达结束单词

if (neighbor.equals(endWord)) {

isFound = true;

p.add(neighbor);

ans.add(new ArrayList<String>(p));

p.remove(p.size() - 1);

}

//加入当前单词

p.add(neighbor);

queue.offer(new ArrayList<String>(p));

p.remove(p.size() - 1);

subVisited.add(neighbor);

}

}

}

visited.addAll(subVisited);

if (isFound) {

break;

}

}

}

private ArrayList<String> getNeighbors(String node, Set<String> dict) {

ArrayList<String> res = new ArrayList<String>();

char chs[] = node.toCharArray();

for (char ch = 'a'; ch <= 'z'; ch++) {

for (int i = 0; i < chs.length; i++) {

if (chs[i] == ch)

continue;

char old\_ch = chs[i];

chs[i] = ch;

if (dict.contains(String.valueOf(chs))) {

res.add(String.valueOf(chs));

}

chs[i] = old\_ch;

}

}

return res;

}

#### [200. 岛屿数量](https://leetcode-cn.com/problems/number-of-islands/)

给定一个由 '1'（陆地）和 '0'（水）组成的的二维网格，计算岛屿的数量。一个岛被水包围，并且它是通过水平方向或垂直方向上相邻的陆地连接而成的。你可以假设网格的四个边均被水包围。

链接：https://leetcode-cn.com/problems/number-of-islands

示例 1:

输入:

11110

11010

11000

00000

输出: 1

示例 2:

输入:

11000

11000

00100

00011

输出: 3

方法一：深度优先搜索 【通过】

直觉

将二维网格看成一个无向图，竖直或水平相邻的 1 之间有边。

算法

线性扫描整个二维网格，如果一个结点包含 1，则以其为根结点启动深度优先搜索。在深度优先搜索过程中，每个访问过的结点被标记为 0。计数启动深度优先搜索的根结点的数量，即为岛屿的数量。

class Solution {

void dfs(char[][] grid, int r, int c) {

int nr = grid.length;

int nc = grid[0].length;

if (r < 0 || c < 0 || r >= nr || c >= nc || grid[r][c] == '0') {

return;

}

grid[r][c] = '0';

dfs(grid, r - 1, c);

dfs(grid, r + 1, c);

dfs(grid, r, c - 1);

dfs(grid, r, c + 1);

}

public int numIslands(char[][] grid) {

if (grid == null || grid.length == 0) {

return 0;

}

int nr = grid.length;

int nc = grid[0].length;

int num\_islands = 0;

for (int r = 0; r < nr; ++r) {

for (int c = 0; c < nc; ++c) {

if (grid[r][c] == '1') {

++num\_islands;

dfs(grid, r, c);

}

}

}

return num\_islands;

}

}

复杂度分析

时间复杂度 : O(M×N)，其中 M 和 N 分别为行数和列数。

空间复杂度 : 最坏情况下为 O(M×N)，此时整个网格均为陆地，深度优先搜索的深度达到 M×N。

方法二: 广度优先搜索 【通过】

算法

线性扫描整个二维网格，如果一个结点包含 1，则以其为根结点启动广度优先搜索。将其放入队列中，并将值设为 0 以标记访问过该结点。迭代地搜索队列中的每个结点，直到队列为空。

class Solution {

public int numIslands(char[][] grid) {

if (grid == null || grid.length == 0) {

return 0;

}

int nr = grid.length;

int nc = grid[0].length;

int num\_islands = 0;

for (int r = 0; r < nr; ++r) {

for (int c = 0; c < nc; ++c) {

if (grid[r][c] == '1') {

++num\_islands;

grid[r][c] = '0'; // mark as visited

Queue<Integer> nei\*\*\*ors = new LinkedList<>();

nei\*\*\*ors.add(r \* nc + c);

while (!nei\*\*\*ors.isEmpty()) {

int id = nei\*\*\*ors.remove();

int row = id / nc;

int col = id % nc;

if (row - 1 >= 0 && grid[row-1][col] == '1') {

nei\*\*\*ors.add((row-1) \* nc + col);

grid[row-1][col] = '0';

}

if (row + 1 < nr && grid[row+1][col] == '1') {

nei\*\*\*ors.add((row+1) \* nc + col);

grid[row+1][col] = '0';

}

if (col - 1 >= 0 && grid[row][col-1] == '1') {

nei\*\*\*ors.add(row \* nc + col-1);

grid[row][col-1] = '0';

}

if (col + 1 < nc && grid[row][col+1] == '1') {

nei\*\*\*ors.add(row \* nc + col+1);

grid[row][col+1] = '0';

}

}

}

}

}

return num\_islands;

}

}

复杂度分析

时间复杂度 : O(M×N)，其中 M 和N 分别为行数和列数。

空间复杂度 :O(min(M,N))，在最坏的情况下（全部为陆地），队列的大小可以达到 min(M，N)。

#### [529. 扫雷游戏](https://leetcode-cn.com/problems/minesweeper/)

让我们一起来玩扫雷游戏！

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：

1. 如果一个地雷（'M'）被挖出，游戏就结束了- 把它改为 **'X'**。
2. 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方块都应该被递归地揭露。
3. 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。
4. 如果在此次点击中，若无更多方块可被揭露，则返回面板。

示例 1：

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:

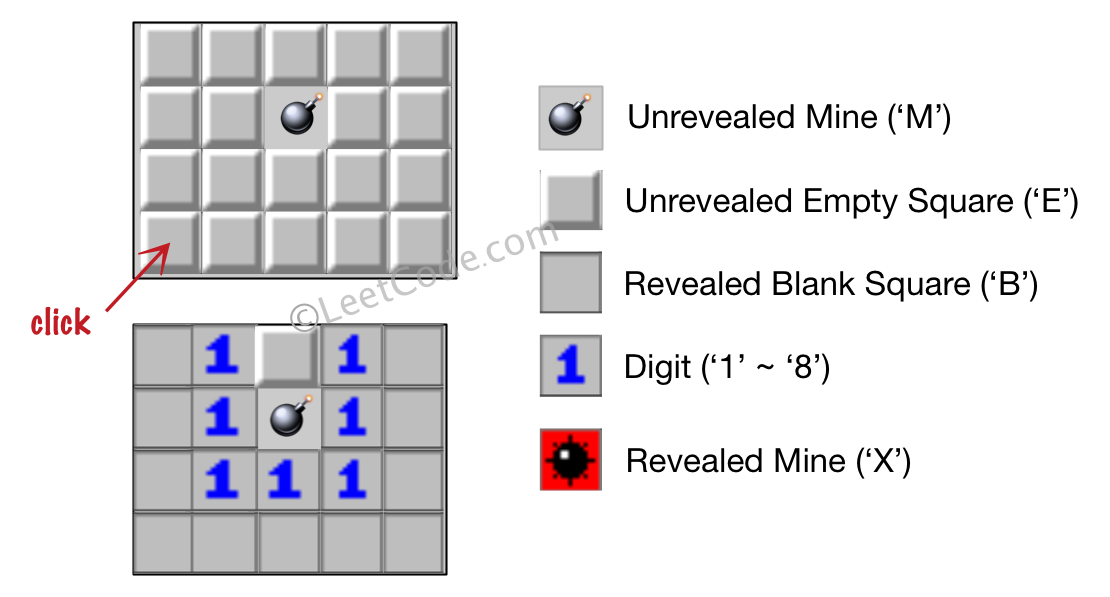
[['B', '1', 'E', '1', 'B'],

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

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

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

链接：https://leetcode-cn.com/problems/minesweeper



示例 2：

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]

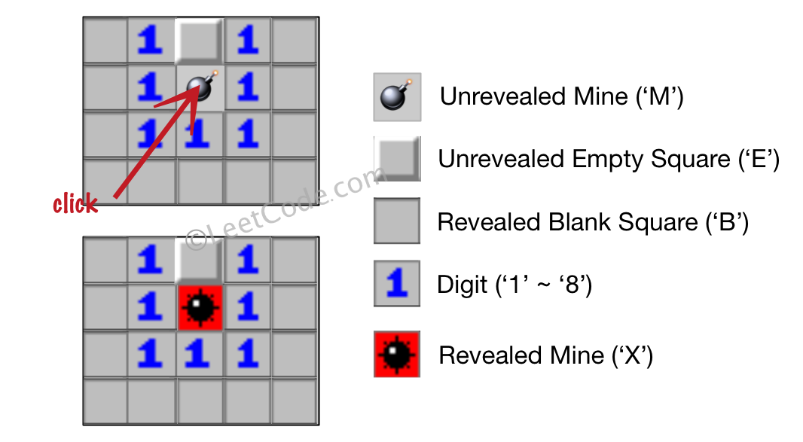
输出:

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

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

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

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



DFS

class Solution {

int[] dx = {-1, -1, 0, 1, 1, 1, 0, -1}; // 相邻位置

int[] dy = {0, 1, 1, 1, 0, -1, -1, -1};

public char[][] updateBoard(char[][] board, int[] click) {

dfs(board, click[0], click[1]);

return board;

}

public void dfs(char[][] board, int x, int y) {

int r = board.length;

int c = board[0].length;

if (x < 0 || x >= r || y < 0 || y >= c) {

return;

}

if (board[x][y] == 'E') { // 如果当前为E，才进行判断是否要递归相邻结点

board[x][y] = 'B';

int count = judge(board, x, y);

if (count == 0) { // 如果为0，则进行递归

for (int i = 0; i < 8; i++) {

dfs(board, x + dx[i], y + dy[i]);

}

} else { // 如果不为0，则更新当前结点的值为地雷数量

board[x][y] = (char) (count + '0');

}

} else if (board[x][y] == 'M'){ // 注意不要用else，否则会递归修改掉已经是数字的位置

board[x][y] = 'X';

}

}

// 获取当前借点相邻的地雷数量

public int judge(char[][] board, int x, int y) {

int r = board.length;

int c = board[0].length;

int count = 0;

for (int i = 0; i < 8; i++) {

int newX = x + dx[i];

int newY = y + dy[i];

if (newX < 0 || newX >= r || newY < 0 || newY >= c) {

continue;

}

if (board[newX][newY] == 'M') {

count++;

}

}

return count;

}

static class Node {

int x;

int y;

Node(int x, int y) {

this.x = x;

this.y = y;

}

}

}

BFS

class Solution {

int[] dx = {-1, -1, 0, 1, 1, 1, 0, -1}; // 相邻位置

int[] dy = {0, 1, 1, 1, 0, -1, -1, -1};

public char[][] updateBoard(char[][] board, int[] click) {

int r = board.length;

int c = board[0].length;

Queue<Node> queue = new LinkedList<>();

queue.offer(new Node(click[0], click[1]));

while(!queue.isEmpty()) {

Node top = queue.poll();

int x = top.x;

int y = top.y;

if (board[x][y] == 'E') { // 如果当前为E，才进行判断是否要递归相邻结点

board[x][y] = 'B';

int count = judge(board, x, y);

if (count == 0) { // 如果为0，则进行递归（感觉这里可以与jude进行优化，但是反而时间增加了，以后再优化吧==）

for (int i = 0; i < 8; i++) {

int newX = x + dx[i];

int newY = y + dy[i];

if (newX < 0 || newX >= r || newY < 0 || newY >= c) {

continue;

}

queue.offer(new Node(newX, newY));

}

} else { // 如果不为0，则不加入队列，同时更新当前结点的值为地雷数量

board[x][y] = (char) (count + '0');

}

} else if (board[x][y] == 'M') { // 如果当前为M，则只更新当前结点

board[x][y] = 'X';

}

}

return board;

}

// 获取当前借点相邻的地雷数量

public int judge(char[][] board, int x, int y) {

int r = board.length;

int c = board[0].length;

int count = 0;

for (int i = 0; i < 8; i++) {

int newX = x + dx[i];

int newY = y + dy[i];

if (newX < 0 || newX >= r || newY < 0 || newY >= c) {

continue;

}

if (board[newX][newY] == 'M') {

count++;

}

}

return count;

}

static class Node {

int x;

int y;

Node(int x, int y) {

this.x = x;

this.y = y;

}

}

}

#### [860. 柠檬水找零](https://leetcode-cn.com/problems/lemonade-change/)

在柠檬水摊上，每一杯柠檬水的售价为 5 美元。顾客排队购买你的产品，（按账单 bills 支付的顺序）一次购买一杯。每位顾客只买一杯柠檬水，然后向你付 5 美元、10 美元或 20 美元。你必须给每个顾客正确找零，也就是说净交易是每位顾客向你支付 5 美元。

注意，一开始你手头没有任何零钱。如果你能给每位顾客正确找零，返回 true ，否则返回 false 。

示例 1：

输入：[5,5,5,10,20]

输出：true

解释：

前 3 位顾客那里，我们按顺序收取 3 张 5 美元的钞票。

第 4 位顾客那里，我们收取一张 10 美元的钞票，并返还 5 美元。

第 5 位顾客那里，我们找还一张 10 美元的钞票和一张 5 美元的钞票。

由于所有客户都得到了正确的找零，所以我们输出 true。

链接：https://leetcode-cn.com/problems/lemonade-change

示例 2：

输入：[5,5,10]

输出：true

示例 3：

输入：[10,10]

输出：false

示例 4：

输入：[5,5,10,10,20]

输出：false

时间复杂度：O(N)，其中 N*N* 是 bills 的长度。空间复杂度：O(1)。

class Solution {

public:

bool lemonadeChange(const vector<int>& bills) {

int five = 0, ten = 0;

for (const auto& bill : bills)

if (bill == 5) five++;

else if (bill == 10 && 0 < five) --five, ++ten;

else if (0 < ten && 0 < five) --five, --ten;

else if (2 < five) five-=3;

else return false;

return true;

}

};

#### [122. 买卖股票的最佳时机 II](https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-ii/)

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

一次遍历

class Solution {

public int maxProfit(int[] prices) {

int maxprofit = 0;

for (int i = 1; i < prices.length; i++) {

if (prices[i] > prices[i - 1])

maxprofit += prices[i] - prices[i - 1];

}

return maxprofit;

}

}

#### [455. 分发饼干](https://leetcode-cn.com/problems/assign-cookies/)

假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。对每个孩子 i ，都有一个胃口值 gi ，这是能让孩子们满足胃口的饼干的最小尺寸；并且每块饼干 j ，都有一个尺寸 sj 。如果 sj >= gi ，我们可以将这个饼干 j 分配给孩子 i ，这个孩子会得到满足。你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。

注意：

你可以假设胃口值为正。

一个小朋友最多只能拥有一块饼干。

示例 1:

输入: [1,2,3], [1,1]

输出: 1

输入: [1,2], [1,2,3]

输出: 2

链接：https://leetcode-cn.com/problems/assign-cookies

解题思路：  
由于分配饼干这件事前后的步骤不会产生关联，所以根据贪心算法的原理，分配的最优策略是每次分配只关注未分配饼干的最小胃口的小朋友。

class Solution {

public int findContentChildren(int[] g, int[] s) {

if(g == null || s == null) return 0;

Arrays.sort(g);

Arrays.sort(s);

int gi = 0, si = 0;

while(gi < g.length && si < s.length){

if(g[gi] <= s[si]) gi++;

si++;

}

return gi;

}

}

#### [874. 模拟行走机器人](https://leetcode-cn.com/problems/walking-robot-simulation/)

机器人在一个无限大小的网格上行走，从点 (0, 0) 处开始出发，面向北方。该机器人可以接收以下三种类型的命令：

-2：向左转 90 度

-1：向右转 90 度

1 <= x <= 9：向前移动 x 个单位长度

在网格上有一些格子被视为障碍物。

第 i 个障碍物位于网格点  (obstacles[i][0], obstacles[i][1])

如果机器人试图走到障碍物上方，那么它将停留在障碍物的前一个网格方块上，但仍然可以继续该路线的其余部分。

返回从原点到机器人的最大欧式距离的平方。

链接：https://leetcode-cn.com/problems/walking-robot-simulation

示例 1：

输入: commands = [4,-1,3], obstacles = []

输出: 25

解释: 机器人将会到达 (3, 4)

示例 2：

输入: commands = [4,-1,4,-2,4], obstacles = [[2,4]]

输出: 65

解释: 机器人在左转走到 (1, 8) 之前将被困在 (1, 4) 处

思路

我们将一步步模拟机器人的路径。由于最多只有 90000 步，这足以通过给定的输入限制。

算法

我们将会存储机器人的位置和方向。如果机器人得到转弯的指令，我们就更新方向；否则就沿给定的方向走指定的步数。

必须注意使用 集合 Set 作为对障碍物使用的数据结构，以便我们可以有效地检查下一步是否受阻。如果不这样做，我们检查 该点是障碍点吗 可能会慢大约 10000 倍。

在某些语言中，我们需要将每个障碍物的坐标编码为 长整型 long 数据，以便它可以放入 集合 Set 数据结构中。或者，我们也可以将坐标编码为 字符串 string。

class Solution {

public int robotSim(int[] commands, int[][] obstacles) {

int[] dx = new int[]{0, 1, 0, -1};

int[] dy = new int[]{1, 0, -1, 0};

int x = 0, y = 0, di = 0;

// Encode obstacles (x, y) as (x+30000) \* (2^16) + (y+30000)

Set<Long> obstacleSet = new HashSet();

for (int[] obstacle: obstacles) {

long ox = (long) obstacle[0] + 30000;

long oy = (long) obstacle[1] + 30000;

obstacleSet.add((ox << 16) + oy);

}

int ans = 0;

for (int cmd: commands) {

if (cmd == -2) //left

di = (di + 3) % 4;

else if (cmd == -1) //right

di = (di + 1) % 4;

else {

for (int k = 0; k < cmd; ++k) {

int nx = x + dx[di];

int ny = y + dy[di];

long code = (((long) nx + 30000) << 16) + ((long) ny + 30000);

if (!obstacleSet.contains(code)) {

x = nx;

y = ny;

ans = Math.max(ans, x\*x + y\*y);

}

}

}

}

return ans;

}

}

#### [55. 跳跃游戏](https://leetcode-cn.com/problems/jump-game/)

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个位置。

示例 1:

输入: [2,3,1,1,4]

输出: true

解释: 我们可以先跳 1 步，从位置 0 到达 位置 1, 然后再从位置 1 跳 3 步到达最后一个位置。

示例 2:

输入: [3,2,1,0,4]

输出: false

解释: 无论怎样，你总会到达索引为 3 的位置。但该位置的最大跳跃长度是 0 ， 所以你永远不可能到达最后一个位置。

一：回溯

这是一个低效的解决方法。我们模拟从第一个位置跳到最后位置的所有方案。从第一个位置开始，模拟所有可以跳到的位置，然后从当前位置重复上述操作，当没有办法继续跳的时候，就回溯。

public class Solution {

public boolean canJumpFromPosition(int position, int[] nums) {

if (position == nums.length - 1) {

return true;

}

int furthestJump = Math.min(position + nums[position], nums.length - 1);

for (int nextPosition = position + 1; nextPosition <= furthestJump; nextPosition++) {

if (canJumpFromPosition(nextPosition, nums)) {

return true;

}

}

return false;

}

public boolean canJump(int[] nums) {

return canJumpFromPosition(0, nums);

}

}

二：贪心 时间复杂度O(n) 空间复杂度O(1)

public class Solution {

public boolean canJump(int[] nums) {

int lastPos = nums.length - 1;

for (int i = nums.length - 1; i >= 0; i--) {

if (i + nums[i] >= lastPos) {

lastPos = i;

}

}

return lastPos == 0;

}

}

#### [45. 跳跃游戏 II](https://leetcode-cn.com/problems/jump-game-ii/)

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

示例:

输入: [2,3,1,1,4]

输出: 2

解释: 跳到最后一个位置的最小跳跃数是 2。从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。

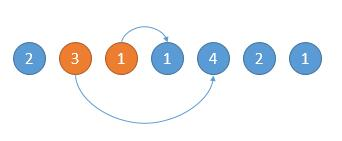
解题思路：

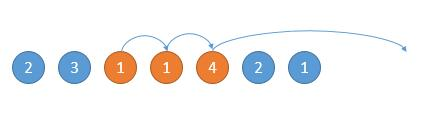
从数组的第 0 个位置开始跳，跳的距离小于等于数组上对应的数。求出跳到最后个位置需要的最短步数。比如上图中的第 0 个位置是 2，那么可以跳 1 个距离，或者 2 个距离，我们选择跳 1 个距离，就跳到了第 1 个位置，也就是 3 上。然后我们可以跳 1，2，3 个距离，我们选择跳 3 个距离，就直接到最后了。所以总共需要 2 步。

解法一 ：顺藤摸瓜

LeetCode 讨论里，大部分都是这个思路，贪婪算法，我们每次在可跳范围内选择可以使得跳的更远的位置。

如下图，开始的位置是 2，可跳的范围是橙色的。然后因为 3 可以跳的更远，所以跳到 3 的位置



如下图，然后现在的位置就是 3 了，能跳的范围是橙色的，然后因为 4 可以跳的更远，所以下次跳到 4 的位置。

写代码的话，我们用 end 表示当前能跳的边界，对于上边第一个图的橙色 1，第二个图中就是橙色的 4，遍历数组的时候，到了边界，我们就重新更新新的边界。

public int jump(int[] nums) {

int end = 0;

int maxPosition = 0;

int steps = 0;

for(int i = 0; i < nums.length - 1; i++){

//找能跳的最远的

maxPosition = Math.max(maxPosition, nums[i] + i);

if( i == end){ //遇到边界，就更新边界，并且步数加一

end = maxPosition;

steps++;

}

}

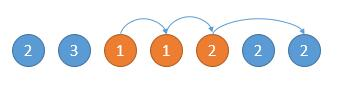
return steps;

}

时间复杂度：O(n)。

空间复杂度：O(1)。

这里要注意一个细节，就是 for 循环中，i < nums.length - 1，少了末尾。因为开始的时候边界是第 00 个位置，steps 已经加 11 了。如下图，如果最后一步刚好跳到了末尾，此时 steps 其实不用加 11 了。如果是 i < nums.length，i 遍历到最后的时候，会进入 if 语句中，steps 会多加 11。



解法二：顺瓜摸藤

我们知道最终要到达最后一个位置，然后我们找前一个位置，遍历数组，找到能到达它的位置，离它最远的就是要找的位置。然后继续找上上个位置，最后到了第 0 个位置就结束了。

至于离它最远的位置，其实我们从左到右遍历数组，第一个满足的位置就是我们要找的。

public int jump(int[] nums) {

int position = nums.length - 1; //要找的位置

int steps = 0;

while (position != 0) { //是否到了第 0 个位置

for (int i = 0; i < position; i++) {

if (nums[i] >= position - i) {

position = i; //更新要找的位置

steps++;

break;

}

}

}

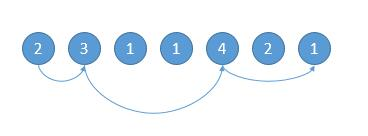
return steps;

}

时间复杂度：O(n²)，因为最坏的情况比如 1111111，position 会从 5 更新到 0，并且每次更新都会经历一个 for 循环。

空间复杂度：O(1)。

这种想法看起来更简单了，为什么奏效呢？我们可以这样想。



从左到右跳的话，2 -> 3 -> 4 -> 12−>3−>4−>1。

从右到左的话，我们找能跳到 11 的最左边的位置，我们找的只能是 44 或者是 44 左边的。

找到 4 的话，不用说，刚好完美。

如果是中间范围 3 和 4 之间的第 2 个 1 变成了 3，那么这个位置也可以跳到末尾的 1，按我们的算法我们就找到了这个 3，也就是 4 左边的位置。但其实并不影响我们的 steps，因为这个数字是 3 到 4 中间范围的数，左边界 3 也可以到这个数，所以下次找的话，会找到边界 3，或者边界 3 左边的数。 会不会直接找到上个边界 2 呢？不会的，如果找到了上一个边界 2，那么意味着从 2 直接跳到 3 和 4 之间的那个数，再从这个数跳到末尾就只需 2 步了，但是其实是需要 3 步的。

#### [69. x 的平方根](https://leetcode-cn.com/problems/sqrtx/)

实现 int sqrt(int x) 函数。

计算并返回 x 的平方根，其中 x 是非负整数。

由于返回类型是整数，结果只保留整数的部分，小数部分将被舍去。

链接：<https://leetcode-cn.com/problems/sqrtx>

示例 1:

输入: 4

输出: 2

输入: 8

输出: 2

说明: 8 的平方根是 2.82842..., 由于返回类型是整数，小数部分将被舍去

二分法：

class Solution {

    public int mySqrt(int x) {

        long left = 0;

        long right = x;

        while (left <= right) {

            long mid = (right + left + 1) / 2;

            if (mid \* mid > x) {

                right = mid - 1;

            } else {

                left = mid + 1;

            }

        }

        return (int) left;

    }

}

#### [367. 有效的完全平方数](https://leetcode-cn.com/problems/valid-perfect-square/)

给定一个正整数 num，编写一个函数，如果 num 是一个完全平方数，则返回 True，否则返回 False。

说明：不要使用任何内置的库函数，如  sqrt。

示例：

输入：16 输出：True

输入：14 输出：False

链接：https://leetcode-cn.com/problems/valid-perfect-square

class Solution {

    public boolean isPerfectSquare(int num) {

        int i = 1;

        while (num > 0) {

            num -= i;

            i += 2;

        }

        return num == 0;

    }

}

#### [33. 搜索旋转排序数组](https://leetcode-cn.com/problems/search-in-rotated-sorted-array/)

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

( 例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2] )。

搜索一个给定的目标值，如果数组中存在这个目标值，则返回它的索引，否则返回 -1 。

你可以假设数组中不存在重复的元素。

你的算法时间复杂度必须是 O(log n) 级别。

链接：https://leetcode-cn.com/problems/search-in-rotated-sorted-array

示例 :

输入: nums = [4,5,6,7,0,1,2], target = 0 输出: 4

输入: nums = [4,5,6,7,0,1,2], target = 3 输出: -1

二分查找

class Solution {

    public int search(int[] nums, int target) {

        // 左下标

        int left = 0;

        // 右下标

        int right = nums.length - 1;

        while (left <= right) {

            int mid = left + (right - left) / 2;

            if (nums[mid] == target) {

                return mid;

            }

            // 需要判断一波截断的位置，左边升序

            if (nums[left] <= nums[mid]) {

                if (target >= nums[left] && target <= nums[mid]) {

                    // 在左边范围内

                    right = mid - 1;

                } else {

                    // 只能从右边找

                    left = mid + 1;

                }

            } else {

                // 右边升序

                if (target >= nums[mid] && target <= nums[right]) {

                    // 只能从右边找

                    left = mid + 1;

                } else {

                    // 在右边范围内

                    right = mid - 1;

                }

            }

        }

        return -1;

    }

}

#### [74. 搜索二维矩阵](https://leetcode-cn.com/problems/search-a-2d-matrix/)

输入:

matrix = [

[1, 3, 5, 7],

[10, 11, 16, 20],

[23, 30, 34, 50]

]

target = 3

输出: true

输入:

matrix = [

[1, 3, 5, 7],

[10, 11, 16, 20],

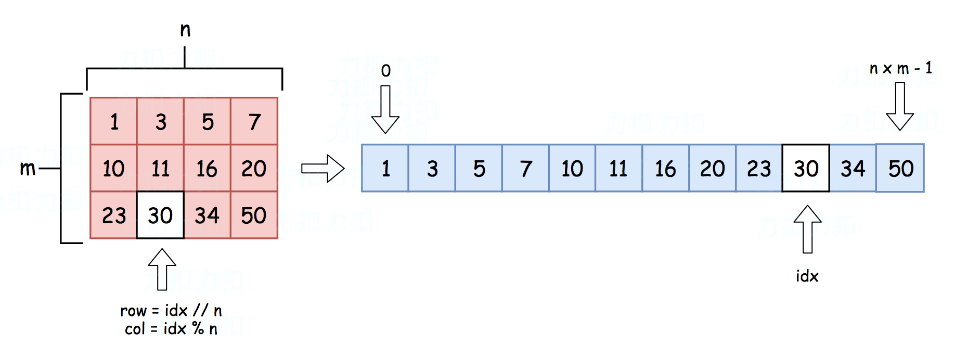
[23, 30, 34, 50]

]

target = 13

输出: false

解题思路：注意到输入的 m x n 矩阵可以视为长度为 m x n的有序数组。



由于该虚数组的序号可以由下式方便地转化为原矩阵中的行和列 (我们当然不会真的创建一个新数组) ，该有序数组非常适合二分查找。

row = idx // n ， col = idx % n。

复杂度分析

时间复杂度 : 由于是标准的二分查找，时间复杂度为O(log(mn))。

空间复杂度 : O(1)。

class Solution {

public boolean searchMatrix(int[][] matrix, int target) {

int m = matrix.length;

if (m == 0) return false;

int n = matrix[0].length;

// 二分查找

int left = 0, right = m \* n - 1;

int pivotIdx, pivotElement;

while (left <= right) {

pivotIdx = (left + right) / 2;

pivotElement = matrix[pivotIdx / n][pivotIdx % n];

if (target == pivotElement) return true;

else {

if (target < pivotElement) right = pivotIdx - 1;

else left = pivotIdx + 1;

}

}

return false;

}

}

#### [153. 寻找旋转排序数组中的最小值](https://leetcode-cn.com/problems/find-minimum-in-rotated-sorted-array/)

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

( 例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2] )。

请找出其中最小的元素。

你可以假设数组中不存在重复元素。

示例 1:

输入: [3,4,5,1,2] 输出: 1

输入: [4,5,6,7,0,1,2] 输出: 0

class Solution {

    public int findMin(int[] nums) {

         if (nums.length == 1) {

            return nums[0];

        }

        int left = 0;

        int right = nums.length - 1;

        // 如果是有序递增的，则第一个是最小的

        if (nums[right] > nums[left]) {

            return nums[0];

        }

        while (left <= right) {

            int mid = left + (right - left) / 2;

            if (nums[mid] < nums[mid + 1]  && nums[mid - 1] > nums[mid]) {

                return nums[mid];

            }

            if (nums[mid] > nums[0]) {

                left = mid + 1;

            } else {

                right = mid - 1;

            }

        }

        return -1;

    }

}

class Solution {

public int findMin(int[] nums) {

// 数组只有一个元素或者未旋转，直接返回0位置

if (nums[nums.length - 1] >= nums[0]) return nums[0];

return findMin(nums, 0, nums.length - 1);

}

private int findMin(int[] nums, int l, int r) {

// 最终情况落差点，即数组中只有两个元素，l最大元素和r最小元素

if (r == l + 1) return nums[r];

int m = l + (r - l) / 2;

// 根据中值信息缩小查找范围，判断条件多样，可以m与l比较也可以m与r比较，还可以m与m-1或m+1比

return nums[m] > nums[l] ? findMin(nums, m, r) : findMin(nums, l, m);

}

}