

- 206. 反转链表 Easy
- 3. 无重复字符的最长子串 Medium
- 146. LRU 缓存机制 Medium
- 215. 数组中的第K个最大元素 Medium
- 1. 两数之和 Easy
- 53. 最大子序和 Easy
- 15. 三数之和 Medium
- 141. 环形链表 Easy
- 21. 合并两个有序链表 Easy
- 160. 相交链表 Easy
- 102. 二叉树的层序遍历 Medium
- 121. 买卖股票的最佳时机 Easy
- 20. 有效的括号 Easy
- 236. 二叉树的最近公共祖先 Medium
- 142. 环形链表 II Medium
- 46. 全排列 Medium
- 47. 全排列 II Medium
- 33. 搜索旋转排序数组 Medium
- 81. 搜索旋转排序数组 II Medium
- 5. 最长回文子串 Medium
- 200. 岛屿数量 Medium
- 300. 最长递增子序列 Medium
- 42. 接雨水 Hard
- 94. 二叉树的中序遍历 Easy
- 70. 爬楼梯 Easy
- 23. 合并K个升序链表 Hard
- 2. 两数相加 Medium
- 19. 删除链表的倒数第 N 个结点 Medium
- 56. 合并区间 Medium
- 104. 二叉树的最大深度 Easy
- 124. 二叉树中的最大路径和 Hard
- 31. 下一个排列 Medium
- 105. 从前序与中序遍历序列构造二叉树 Medium
- 239. 滑动窗口最大值 Hard
- 155. 最小栈 Easy
- 76. 最小覆盖子串 Hard
- 4. 寻找两个正序数组的中位数 Hard
- 72. 编辑距离 Hard
- 148. 排序链表 Medium
- 543. 二叉树的直径 Easy
- 169. 多数元素 Easy
- 98. 验证二叉搜索树 Medium
- 101. 对称二叉树 Easy
- 226. 翻转二叉树 Easy
- 234. 回文链表 Easy
- 78. 子集 Medium
- 90. 子集 II Medium
- 32. 最长有效括号 Hard

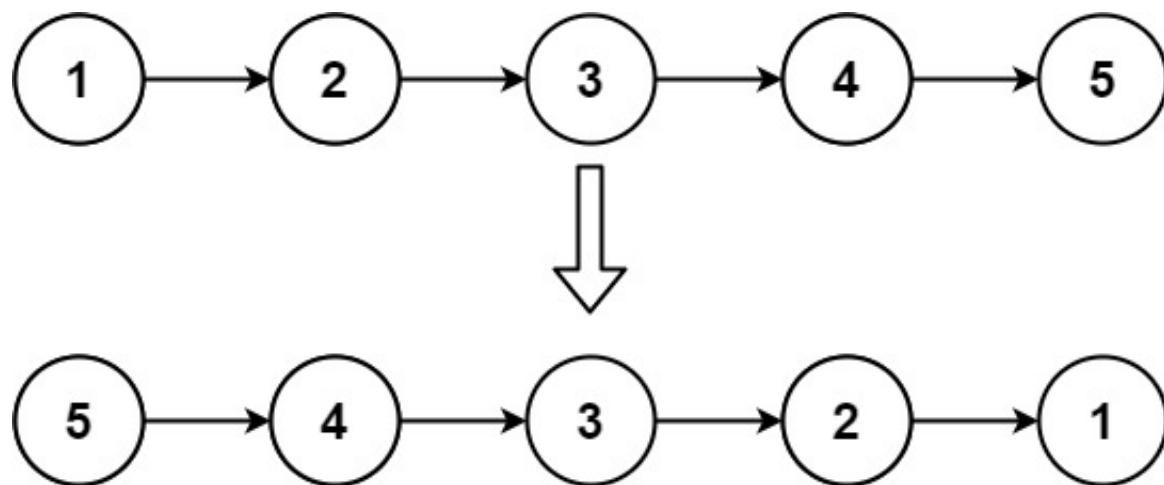
- 48. 旋转图像 Medium
- 322. 零钱兑换 Medium
- 518. 零钱兑换 II Medium
- 34. 在排序数组中查找元素的第一个和最后一个位置 Medium
- 22. 括号生成 Medium
- 64. 最小路径和 Medium
- 136. 只出现一次的数字 Easy
- 39. 组合总和 Medium
- 283. 移动零 Easy
- 297. 二叉树的序列化与反序列化 Hard
- 240. 搜索二维矩阵 II Medium
- 62. 不同路径 Medium
- 128. 最长连续序列 Medium
- 221. 最大正方形 Medium
- 198. 打家劫舍 Medium
- 213. 打家劫舍 II Medium
- 337. 打家劫舍 III Medium
- 79. 单词搜索 Medium
- 152. 乘积最大子数组 Medium
- 739. 每日温度 Medium
- 560. 和为 K 的子数组 Medium
- 287. 寻找重复数 Medium
- 394. 字符串解码 Medium
- 208. 实现 Trie (前缀树) Medium
- 55. 跳跃游戏 Medium
- 207. 课程表 Medium
- 210. 课程表 II Medium
- 494. 目标和 Medium
- 647. 回文子串 Medium
- 17. 电话号码的字母组合 Medium
- 84. 柱状图中最大的矩形 Hard
- 85. 最大矩形 Hard
- 96. 不同的二叉搜索树 Medium
- 238. 除自身以外数组的乘积 Medium
- 617. 合并二叉树 Easy
- 279. 完全平方数 Medium
- 448. 找到所有数组中消失的数字 Easy
- 309. 最佳买卖股票时机含冷冻期 Medium
- 49. 字母异位词分组 Medium
- 416. 分割等和子集 Hard
- 437. 路径总和 III Medium
- 438. 找到字符串中所有字母异位词 Medium
- 312. 戳气球 Hard
- 75. 颜色分类 Medium
- 114. 二叉树展开为链表 Medium
- 139. 单词拆分 Medium
- 10. 正则表达式匹配 Hard
- 11. 盛最多水的容器 Medium
- 347. 前 K 个高频元素

- 581. 最短无序连续子数组 Medium
- 621. 任务调度器 Medium
- 399. 除法求值 Medium
- 191. 位1的个数 Easy
- 338. 比特位计数 Easy
- 406. 根据身高重建队列 Medium
- 151. 翻转字符串里的单词 Medium
- 538. 把二叉搜索树转换为累加树 Medium
- 301. 删除无效的括号 Hard

206. 反转链表 Easy

给你单链表的头节点

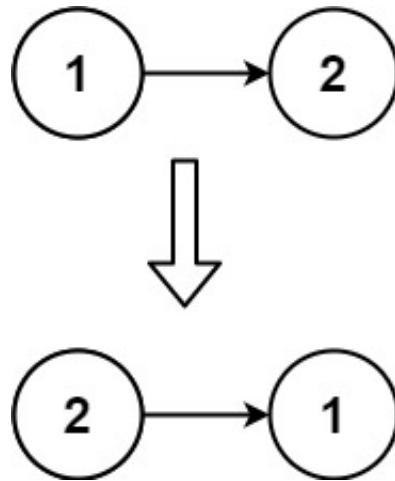
示例 1：



输入： head = [1,2,3,4,5]

输出： [5,4,3,2,1]

示例 2：



输入: head = [1, 2]

输出: [2, 1]

示例 3:

输入: head = []

输出: []

提示:

- 链表中节点的数目范围是 [0, 5000]
- $-5000 \leq \text{Node.val} \leq 5000$

进阶: 链表可以选用迭代或递归方式完成反转。你能否用两种方法解决这道题?

迭代

```

class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode pre = null;
        ListNode cur = head;
        while (cur != null) {
            ListNode next = cur.next;
            cur.next = pre;
            pre = cur;
            cur = next;
        }
        return pre;
    }
}

```

递归

```
class Solution {
    public ListNode reverseList(ListNode head) {
        if (head == null || head.next == null)
            return head;
        ListNode lastNode = reverseList(head.next);
        head.next.next = head;
        head.next = null;
        return lastNode;
    }
}
```

3. 无重复字符的最长子串 Medium

给定一个字符串 `s`，请你找出其中不含有重复字符的 **最长子串** 的长度。

示例 1:

```
输入: s = "abcabcbb"
输出: 3
解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。
```

示例 2:

```
输入: s = "bbbbb"
输出: 1
解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。
```

示例 3:

```
输入: s = "pwwkew"
输出: 3
解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。
请注意，你的答案必须是 子串 的长度，"pwke" 是一个子序列，不是子串。
```

示例 4:

```
输入: s = ""
输出: 0
```

提示:

- `0 <= s.length <= 5 * 104`
- `s` 由英文字母、数字、符号和空格组成

```

class Solution {
    public int lengthOfLongestSubstring(String s) {
        Map<Character, Integer> hashMap = new HashMap<>();
        int left = -1, right = 0;
        int len = 0;
        while (right < s.length()) {
            char key = s.charAt(right);
            if (hashMap.containsKey(key))
                left = Math.max(left, hashMap.get(key));
            hashMap.put(key, right);
            len = Math.max(len, right - left);
            right++;
        }
        return len;
    }
}

```

```

class Solution {
    public int lengthOfLongestSubstring(String s) {
        int[] eleIndex = new int[128];
        Arrays.fill(eleIndex, -1);
        char[] cs = s.toCharArray();
        int left = -1;
        int len = 0;
        for (int i = 0; i < cs.length; i++) {
            left = Math.max(left, eleIndex[cs[i]]);
            len = Math.max(len, i - left);
            eleIndex[cs[i]] = i;
        }
        return len;
    }
}

```

146. LRU 缓存机制 Medium

运用你所掌握的数据结构，设计和实现一个 [LRU\(最近最少使用\)缓存机制](#)。

实现 `LRUCache` 类：

- `LRUCache(int capacity)` 以正整数作为容量 `capacity` 初始化 LRU 缓存
- `int get(int key)` 如果关键字 `key` 存在于缓存中，则返回关键字的值，否则返回 `-1`。
- `void put(int key, int value)` 如果关键字已经存在，则变更其数据值；如果关键字不存在，则插入该组「关键字-值」。当缓存容量达到上限时，它应该在写入新数据之前删除最久未使用的数据值，从而为新的数据值留出空间。

进阶：你是否可以在 $O(1)$ 时间复杂度内完成这两种操作？

示例：

输入

```
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
```

输出

```
[null, null, null, 1, null, -1, null, -1, 3, 4]
```

解释

```
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // 缓存是 {1=1}
lRUCache.put(2, 2); // 缓存是 {1=1, 2=2}
lRUCache.get(1);    // 返回 1
lRUCache.put(3, 3); // 该操作会使得关键字 2 作废, 缓存是 {1=1, 3=3}
lRUCache.get(2);    // 返回 -1 (未找到)
lRUCache.put(4, 4); // 该操作会使得关键字 1 作废, 缓存是 {4=4, 3=3}
lRUCache.get(1);    // 返回 -1 (未找到)
lRUCache.get(3);    // 返回 3
lRUCache.get(4);    // 返回 4
```

提示：

- $1 \leq \text{capacity} \leq 3000$
- $0 \leq \text{key} \leq 10000$
- $0 \leq \text{value} \leq 105$
- 最多调用 $2 * 105$ 次 `get` 和 `put`

```
class LRUCache {
    private HashMap<Integer, deLinkedNode> hashMap;
    private int capacity;
    private int len;
    private deLinkedNode dummyHead;
    private deLinkedNode tail;

    public LRUCache(int capacity) {
        this.capacity = capacity;
        this.len = 0;
        hashMap = new HashMap<>();
        dummyHead = new deLinkedNode();
        tail = new deLinkedNode();
        dummyHead.next = tail;
        tail.pre = dummyHead;
    }

    public int get(int key) {
        if (!hashMap.containsKey(key))
```

```

        return -1;
    deLinkedNode node = hashMap.get(key);
    moveToHead(node);
    return node.val;
}

private void moveToHead(deLinkedNode node) {
    deleteNode(node);
    insertNode(node);
}

private void deleteNode(deLinkedNode node) {
    if (len == 0)
        return;
    hashMap.remove(node.key);
    len--;
    node.pre.next = node.next;
    node.next.pre = node.pre;
}

private void insertNode(deLinkedNode node) {
    while (len >= capacity)
        deleteNode(tail.pre);
    hashMap.put(node.key, node);
    len++;
    deLinkedNode next = dummyHead.next;
    dummyHead.next = node;
    node.pre = dummyHead;
    node.next = next;
    next.pre = node;
}

public void put(int key, int value) {
    if (!hashMap.containsKey(key))
        insertNode(new deLinkedNode(key, value));
    else {
        deLinkedNode node = hashMap.get(key);
        node.val = value;
        moveToHead(node);
    }
}

class deLinkedNode {
    deLinkedNode pre;
    deLinkedNode next;
    int key;
    int val;
}

```

```

public deLinkedNode() {
}

public deLinkedNode(int key, int val) {
    this.key = key;
    this.val = val;
}
}

```

215. 数组中的第K个最大元素 Medium

给定整数数组 `nums` 和整数 `k`，请返回数组中第 `k` 个最大的元素。

请注意，你需要找的是数组排序后的第 `k` 个最大的元素，而不是第 `k` 个不同的元素。

示例 1:

输入: [3,2,1,5,6,4] 和 `k` = 2
 输出: 5

示例 2:

输入: [3,2,3,1,2,4,5,5,6] 和 `k` = 4
 输出: 4

提示:

- `1 <= k <= nums.length <= 104`
- `-104 <= nums[i] <= 104`

```

class Solution {
    public int findKthLargest(int[] nums, int k) {
        Arrays.sort(nums);
        return nums[nums.length - k];
    }
}

```

```

class Solution {
    public int findKthLargest(int[] nums, int k) {
        PriorityQueue<Integer> pq = new PriorityQueue<Integer>(new Comparator<Integer>()
        () {
            @Override
            public int compare(Integer o1, Integer o2) {
                return o2 - o1;
            }
        });
        for (int num : nums) {
            pq.offer(num);
            if (pq.size() > k) {
                pq.poll();
            }
        }
        return pq.peek();
    }
}

```

```

        }
    });
    for (int num : nums) {
        pq.offer(num);
    }
    while (--k > 0)
        pq.poll();
    return pq.peek();
}
}

```

```

class Solution {
    private Random random = new Random();

    public int findKthLargest(int[] nums, int k) {
        quickSort(nums, 0, nums.length - 1);
        return nums[k - 1];
    }

    private void quickSort(int[] nums, int left, int right) {
        if (left >= right)
            return;
        int randomIndex = randomPartition(nums, left, right);
        quickSort(nums, left, randomIndex - 1);
        quickSort(nums, randomIndex + 1, right);
    }

    private int randomPartition(int[] nums, int left, int right) {
        int index = random.nextInt(right - left + 1) + left;
        swap(nums, left, index);
        return partition(nums, left, right);
    }

    private int partition(int[] nums, int left, int right) {
        int pivot = nums[left];
        int pivotIndex = left;
        while (left < right) {
            // 把下面两个while语句内和pivot进行比较的大于小于符号改变一下即可实现数组降序排列
            while (left < right && nums[right] <= pivot)
                right--;
            while (left < right && nums[left] >= pivot)
                left++;
            swap(nums, left, right);
        }
        swap(nums, left, pivotIndex);
        return left;
    }
}

```

```
private void swap(int[] nums, int i, int j) {
    int temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
}
```

```
class Solution {
    private Random random;

    public int findKthLargest(int[] nums, int k) {
        random = new Random();
        int left = 0, right = nums.length - 1;
        while (left <= right) {
            int randomIndex = randomPartition(nums, left, right);
            if (randomIndex == nums.length - k)
                break;
            else if (randomIndex < nums.length - k)
                left = randomIndex + 1;
            else
                right = randomIndex - 1;
        }
        return nums[nums.length - k];
    }

    private int randomPartition(int[] nums, int left, int right) {
        if (left > right)
            return -1;
        int randomIndex = random.nextInt(right - left + 1) + left;
        swap(nums, right, randomIndex);
        return partition(nums, left, right);
    }

    private int partition(int[] nums, int left, int right) {
        int pivot = nums[right];
        int pivotIndex = right;
        while (left < right) {
            while (left < right && pivot >= nums[left])
                left++;
            while (left < right && pivot <= nums[right])
                right--;
            swap(nums, left, right);
        }
        swap(nums, left, pivotIndex);
        return left;
    }

    private void swap(int[] nums, int i, int j) {
```

```
    int temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
}
}
```

1. 两数之和 Easy

给定一个整数数组 `nums` 和一个整数目标值 `target`，请你在该数组中找出 **和为目标值 `target`** 的那 **两个** 整数，并返回它们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。你可以按任意顺序返回答案。

示例 1：

```
输入: nums = [2,7,11,15], target = 9
输出: [0,1]
解释: 因为 nums[0] + nums[1] == 9 , 返回 [0, 1] 。
```

示例 2：

```
输入: nums = [3,2,4], target = 6
输出: [1,2]
```

示例 3：

```
输入: nums = [3,3], target = 6
输出: [0,1]
```

提示：

- $2 \leq \text{nums.length} \leq 104$
- $-109 \leq \text{nums}[i] \leq 109$
- $-109 \leq \text{target} \leq 109$
- 只会存在一个有效答案

进阶：你可以想出一个时间复杂度小于 $O(n^2)$ 的算法吗？

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> hashMap = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            int k = target - nums[i];
            if (hashMap.containsKey(k))
                return new int[]{hashMap.get(k), i};
            hashMap.put(nums[i], i);
        }
        return null;
    }
}
```

53. 最大子序和 Easy

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。

示例 1：

```
输入: nums = [-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 。
```

示例 2：

```
输入: nums = [1]
输出: 1
```

示例 3：

```
输入: nums = [0]
输出: 0
```

示例 4：

```
输入: nums = [-1]
输出: -1
```

示例 5：

```
输入: nums = [-100000]
输出: -100000
```

提示:

- $1 \leq \text{nums.length} \leq 3 * 10^4$
- $-105 \leq \text{nums}[i] \leq 105$

进阶: 如果你已经实现复杂度为 $O(n)$ 的解法, 尝试使用更为精妙的 分治法 求解

动态规划

```
class Solution {  
    public int maxSubArray(int[] nums) {  
        int[] dp = new int[nums.length];  
        dp[0] = nums[0];  
        for (int i = 1; i < nums.length; i++) {  
            if (dp[i - 1] >= 0)  
                dp[i] = dp[i - 1] + nums[i];  
            else  
                dp[i] = nums[i];  
        }  
        Arrays.sort(dp);  
        return dp[nums.length - 1];  
    }  
}
```

自己想的

```
class Solution {  
    public int maxSubArray(int[] nums) {  
        int max = Integer.MIN_VALUE;  
        int temp = 0;  
        for (int i = 0; i < nums.length; i++) {  
            if (nums[i] >= 0 || temp + nums[i] >= 0) {  
                temp += nums[i];  
                max = Math.max(temp, max);  
            } else {  
                max = Math.max(max, nums[i]);  
                temp = 0;  
            }  
        }  
        return max;  
    }  
}
```

分治

```
class Solution {
```

```

public int maxSubArray(int[] nums) {
    return divideAndConquer(nums, 0, nums.length - 1);
}

private int divideAndConquer(int[] nums, int left, int right) {
    if (left > right)
        return Integer.MIN_VALUE;
    int mid = (left + right) / 2;
    int leftSum = 0, maxLeftSum = 0;
    int rightSum = 0, maxRightSum = 0;
    for (int i = mid - 1; i >= left; i--) {
        leftSum += nums[i];
        maxLeftSum = Math.max(maxLeftSum, leftSum);
    }
    for (int i = mid + 1; i <= right; i++) {
        rightSum += nums[i];
        maxRightSum = Math.max(maxRightSum, rightSum);
    }
    return Math.max(maxLeftSum + maxRightSum + nums[mid],
        Math.max(divideAndConquer(nums, left, mid - 1), divideAndConquer(nums, mid + 1,
        right)));
}
}

```

贪心

```

class Solution {
    public int maxSubArray(int[] nums) {
        int sum = Integer.MIN_VALUE;
        int maxSum = Integer.MIN_VALUE;
        for (int num : nums) {
            if (sum < 0)
                sum = 0;
            sum += num;
            maxSum = Math.max(maxSum, sum);
        }
        return maxSum;
    }
}

```

15. 三数之和 Medium

给你一个包含 n 个整数的数组 `nums`，判断 `nums` 中是否存在三个元素 a, b, c ，使得 $a + b + c = 0$ ？请你找出所有和为 0 且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例 1：

```
输入: nums = [-1,0,1,2,-1,-4]
输出: [[-1,-1,2],[-1,0,1]]
```

示例 2：

```
输入: nums = []
输出: []
```

示例 3：

```
输入: nums = [0]
输出: []
```

提示：

- $0 \leq \text{nums.length} \leq 3000$
- $-105 \leq \text{nums}[i] \leq 105$

题目要求三元组不能重复，所以需要先将数组排序，再进行适当的优化。

暴力方法超时，上排序+双指针。

```
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        List<List<Integer>> res = new ArrayList<>();
        Arrays.sort(nums);
        for (int i = 0; i < nums.length - 2; i++) {
            if (nums[i] > 0)
                break;
            if (i > 0 && nums[i] == nums[i - 1])
                continue;
            int x = i + 1, y = nums.length - 1;
            while (x < y) {
                int sum = nums[i] + nums[x] + nums[y];
                if (sum < 0)
                    x++;
                else if (sum > 0)
                    y--;
                else
                    res.add(Arrays.asList(nums[i], nums[x], nums[y]));
            }
        }
        return res;
    }
}
```

```

        res.add(Arrays.asList(nums[i], nums[x++], nums[y--]));
    while (x < y && x > i + 1 && nums[x - 1] == nums[x]) {
        x++;
    }
    while (x < y && y < nums.length - 1 && nums[y] == nums[y + 1]) {
        y--;
    }
}
return res;
}
}

```

141. 环形链表 Easy

给定一个链表，判断链表中是否有环。

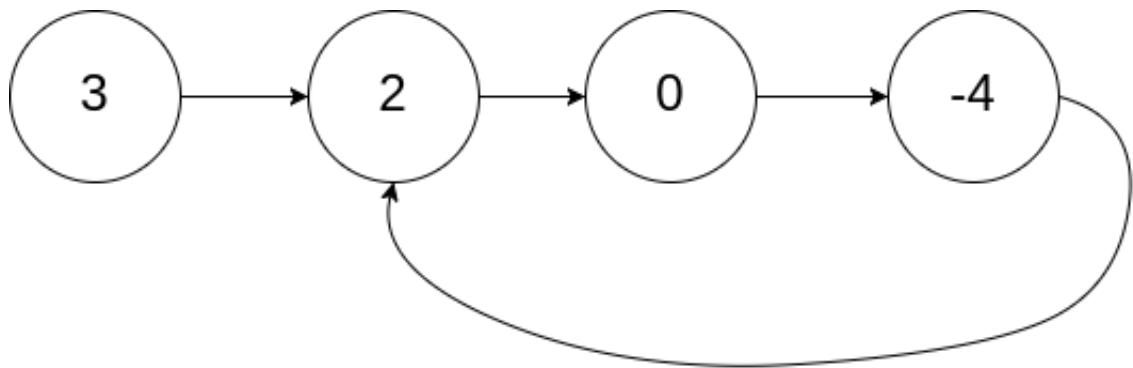
如果链表中有某个节点，可以通过连续跟踪 `next` 指针再次到达，则链表中存在环。为了表示给定链表中的环，我们使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。如果 `pos` 是 `-1`，则在该链表中没有环。注意：`pos` 不作为参数进行传递，仅仅是为了标识链表的实际情况。

如果链表中存在环，则返回 `true`。否则，返回 `false`。

进阶：

你能用 $O(1)$ （即，常量）内存解决此问题吗？

示例 1：

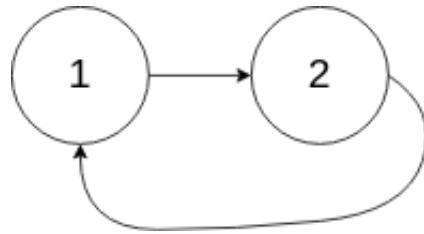


输入： `head = [3, 2, 0, -4], pos = 1`

输出： `true`

解释： 链表中有一个环，其尾部连接到第二个节点。

示例 2：

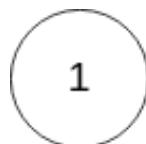


输入: head = [1,2], pos = 0

输出: true

解释: 链表中有一个环, 其尾部连接到第一个节点。

示例 3:



输入: head = [1], pos = -1

输出: false

解释: 链表中没有环。

提示:

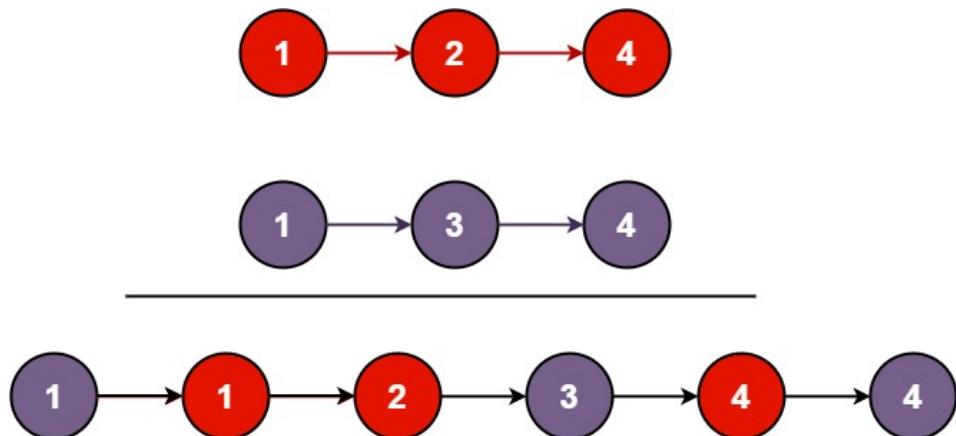
- 链表中节点的数目范围是 `[0, 104]`
- `-105 <= Node.val <= 105`
- `pos` 为 `-1` 或者链表中的一个 **有效索引**。

```
public class Solution {
    public boolean hasCycle(ListNode head) {
        if (head == null || head.next == null)
            return false;
        ListNode slow = head, fast = head;
        do {
            if (fast != null && fast.next != null) {
                slow = slow.next;
                fast = fast.next.next;
            } else
                return false;
        } while (slow != fast);
        return true;
    }
}
```

21. 合并两个有序链表 Easy

将两个升序链表合并为一个新的 升序 链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

示例 1：



输入: `l1 = [1,2,4], l2 = [1,3,4]`

输出: `[1,1,2,3,4,4]`

示例 2：

输入: `l1 = [], l2 = []`

输出: `[]`

示例 3：

输入: `l1 = [], l2 = [0]`

输出: `[0]`

提示：

- 两个链表的节点数目范围是 `[0, 50]`
- `-100 <= Node.val <= 100`
- `l1` 和 `l2` 均按 非递减顺序 排列

```
class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummyHead = new ListNode();
        ListNode cur = dummyHead;
        while (l1 != null && l2 != null) {
            if (l1.val < l2.val) {
                cur.next = l1;
                l1 = l1.next;
            } else {
                cur.next = l2;
                l2 = l2.next;
            }
            cur = cur.next;
        }
        if (l1 == null)
            cur.next = l2;
        else
            cur.next = l1;
        return dummyHead.next;
    }
}
```

```

        }
        cur = cur.next;
    }
    if (l1 != null)
        cur.next = l1;
    else cur.next = l2;
    return dummyHead.next;
}
}

```

```

class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if (l1 == null)
            return l2;
        if (l2 == null)
            return l1;
        if (l1.val < l2.val) {
            l1.next = mergeTwoLists(l1.next, l2);
            return l1;
        } else {
            l2.next = mergeTwoLists(l1, l2.next);
            return l2;
        }
    }
}

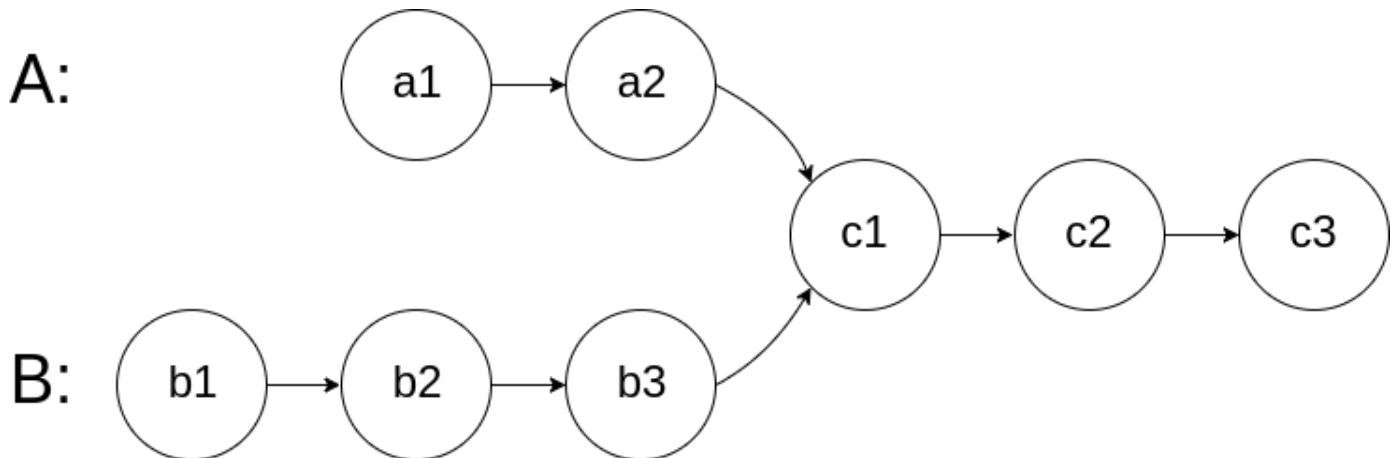
```

递归写法，没有想到。

160. 相交链表 Easy

给你两个单链表的头节点 `headA` 和 `headB`，请你找出并返回两个单链表相交的起始节点。如果两个链表没有交点，返回 `null`。

图示两个链表在节点 `c1` 开始相交：

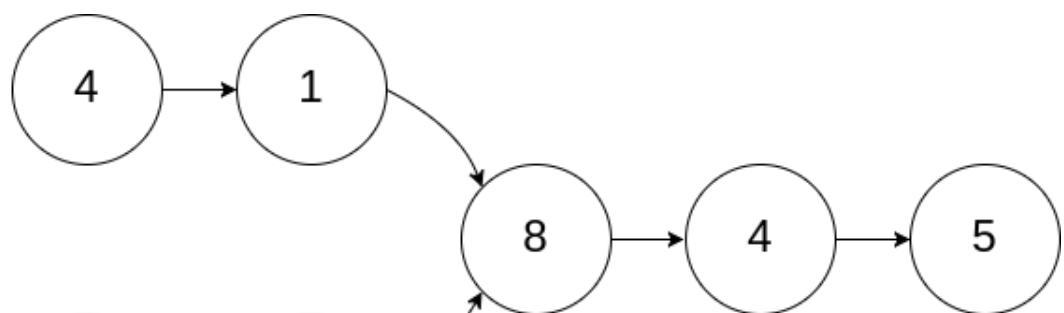


题目数据 **保证** 整个链式结构中不存在环。

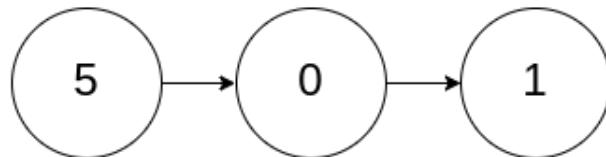
注意，函数返回结果后，链表必须 **保持其原始结构**。

示例 1：

A:



B:



输入: intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
输出: Intersected at '8'

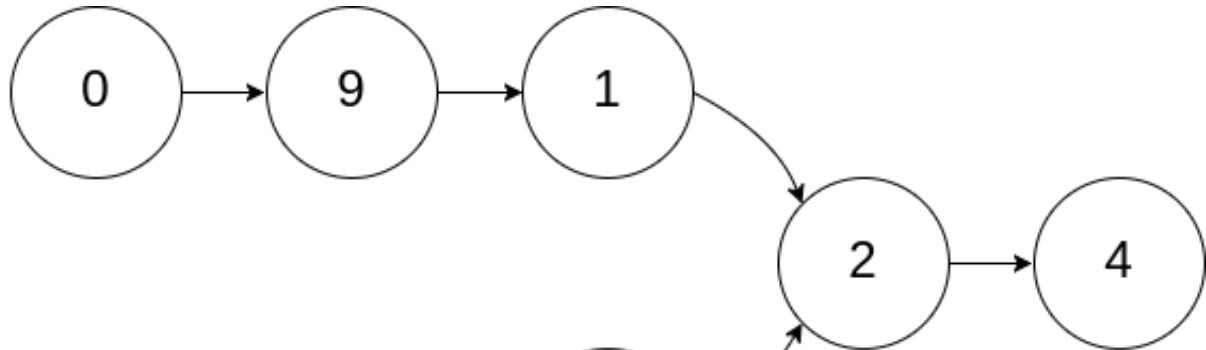
解释：相交节点的值为 8（注意，如果两个链表相交则不能为 0）。

从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。

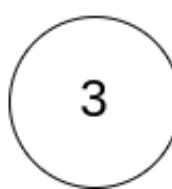
在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。

示例 2：

A:



B:



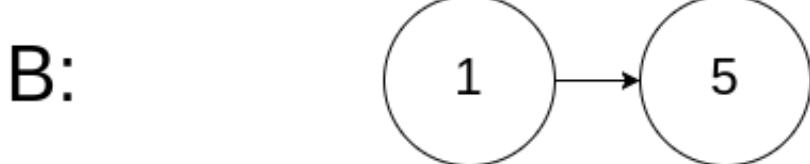
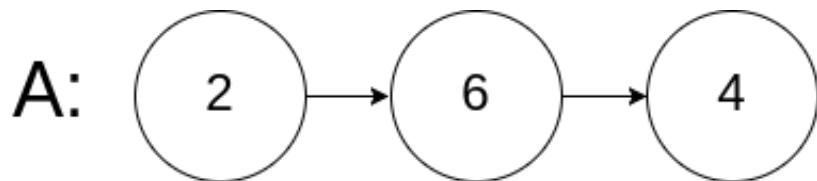
输入: intersectVal = 2, listA = [0,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1
输出: Intersected at '2'

解释：相交节点的值为 2（注意，如果两个链表相交则不能为 0）。

从各自的表头开始算起，链表 A 为 [0,9,1,2,4]，链表 B 为 [3,2,4]。

在 A 中，相交节点前有 3 个节点；在 B 中，相交节点前有 1 个节点。

示例 3：



输入: `intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2`
 输出: `null`

解释: 从各自的表头开始算起, 链表 A 为 `[2,6,4]`, 链表 B 为 `[1,5]`。

由于这两个链表不相交, 所以 `intersectVal` 必须为 0, 而 `skipA` 和 `skipB` 可以是任意值。

这两个链表不相交, 因此返回 `null`。

提示:

- `listA` 中节点数目为 `m`
- `listB` 中节点数目为 `n`
- `0 <= m, n <= 3 * 104`
- `1 <= Node.val <= 105`
- `0 <= skipA <= m`
- `0 <= skipB <= n`
- 如果 `listA` 和 `listB` 没有交点, `intersectVal` 为 `0`
- 如果 `listA` 和 `listB` 有交点, `intersectVal == listA[skipA + 1] == listB[skipB + 1]`

进阶: 你能否设计一个时间复杂度 `O(n)`、仅用 `O(1)` 内存的解决方案?

```

public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if (headA == null || headB == null)
            return null;
        ListNode l1 = headA, l2 = headB;
        while (l1 != l2) {
            l1 = l1 == null ? headB : l1.next;
            l2 = l2 == null ? headA : l2.next;
        }
        return l1;
    }
}

```

官方解答思路清晰, 简洁明了。

102. 二叉树的层序遍历 Medium

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)。

示例：

二叉树： [3,9,20,null,null,15,7]，

```
3
/ \
9   20
 /   \
15    7
```

返回其层序遍历结果：

```
[
[3],
[9,20],
[15,7]
]
```

```
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> ret = new ArrayList<List<Integer>>();
        if (root == null)
            return ret;

        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        queue.offer(root);
        while (!queue.isEmpty()) {
            List<Integer> level = new ArrayList<Integer>();
            int currentLevelSize = queue.size();
            for (int i = 1; i <= currentLevelSize; ++i) {
                TreeNode node = queue.poll();
                level.add(node.val);
                if (node.left != null) {
                    queue.offer(node.left);
                }
                if (node.right != null) {
                    queue.offer(node.right);
                }
            }
            ret.add(level);
        }
    }
}
```

```

        return ret;
    }
}

class Solution {
    List<List<Integer>> res;

    public List<List<Integer>> levelOrder(TreeNode root) {
        res = new ArrayList<>();
        dfs(1, root);
        return res;
    }

    private void dfs(int level, TreeNode root) {
        if (root == null)
            return;
        if (level > res.size())
            res.add(new ArrayList<>());
        res.get(level - 1).add(root.val);
        dfs(level + 1, root.left);
        dfs(level + 1, root.right);
    }
}

```

121. 买卖股票的最佳时机 Easy

给定一个数组 `prices`，它的第 `i` 个元素 `prices[i]` 表示一支给定股票第 `i` 天的价格。

你只能选择 **某一天** 买入这只股票，并选择在 **未来的某一个不同的日子** 卖出该股票。设计一个算法来计算你所能获取的最大利润。

返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 `0`。

示例 1：

输入： `[7,1,5,3,6,4]`

输出： 5

解释： 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5

。

注意利润不能是 $7-1 = 6$ ，因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。

示例 2：

```
输入: prices = [7,6,4,3,1]
输出: 0
解释: 在这种情况下, 没有交易完成, 所以最大利润为 0。
```

提示:

- `1 <= prices.length <= 105`
- `0 <= prices[i] <= 104`

```
class Solution {
    public int maxProfit(int[] prices) {
        int minPrice = Integer.MAX_VALUE;
        int profit = Integer.MIN_VALUE;
        for (int price : prices) {
            minPrice = Math.min(minPrice, price);
            profit = Math.max(profit, price - minPrice);
        }
        return profit;
    }
}
```

在线处理, 不知道是不是 dp 的一种

```
class Solution {
    public int maxProfit(int[] prices) {
        int[][] dp = new int[prices.length][2];
        dp[0][0] = 0;
        dp[0][1] = -prices[0];
        for (int i = 1; i < prices.length; i++) {
            dp[i][0] = Math.max(prices[i] + dp[i - 1][1], dp[i - 1][0]);
            // 让如果可以买卖多次, 那么下面一行应该改为
            // Math.max(dp[i - 1][1], dp[i - 1][0] - prices[i])
            dp[i][1] = Math.max(dp[i - 1][1], dp[0][0] - prices[i]);
        }
        return dp[prices.length - 1][0];
    }
}
```

20. 有效的括号 Easy

给定一个只包括 `'()'`, `'{}'`, `'[]'` 的字符串 `s`，判断字符串是否有效。

有效字符串需满足：

1. 左括号必须用相同类型的右括号闭合。
2. 左括号必须以正确的顺序闭合。

示例 1：

```
输入: s = "()"
输出: true
```

示例 2：

```
输入: s = "()[]{}"
输出: true
```

示例 3：

```
输入: s = "(]"
输出: false
```

示例 4：

```
输入: s = "([)"
输出: false
```

示例 5：

```
输入: s = "{}"
输出: true
```

提示：

- `1 <= s.length <= 104`
- `s` 仅由括号 `'()'[]{}'` 组成

```
class Solution {
    public boolean isValid(String s) {
        Stack<Character> stack = new Stack<>();
        char[] charArray = s.toCharArray();
        for (char item : charArray) {
            if (item == '(' || item == '{' || item == '[')
                stack.push(item);
            else if (stack.isEmpty())
                return false;
            else if (item == ')') {
                if (stack.pop() != '(')
                    return false;
            } else if (item == '}') {
                if (stack.pop() != '{')
                    return false;
            } else if (item == ']') {
                if (stack.pop() != '[')
                    return false;
            }
        }
        return stack.isEmpty();
    }
}
```

```

        else if (parenthesisMatched(stack.pop(), item))
            continue;
        else
            return false;
    }
    return stack.isEmpty();
}

private boolean parenthesisMatched(char left, char right) {
    switch (left) {
        case '(':
            return ')' == right;
        case '{':
            return '}' == right;
        case '[':
            return ']' == right;
        default:
            return false;
    }
}
}

```

下面是两个比较巧妙的方法

```

class Solution {
    public boolean isValid(String s) {
        Stack<Character> stack = new Stack<>();
        char[] charArray = s.toCharArray();
        for (char item : charArray) {
            if (item == '(')
                stack.push(')');
            else if (item == '{')
                stack.push('}');
            else if (item == '[')
                stack.push(']');
            else if (stack.isEmpty() || item != stack.pop())
                return false;
        }
        return stack.isEmpty();
    }
}

```

```

class Solution {
    public boolean isValid(String s) {
        int len = s.length();
        for (int i = 0; i < len / 2; i++)
            s = s.replace("()", "").replace("{}","").replace("[]","");
        return s.length() == 0;
    }
}

```

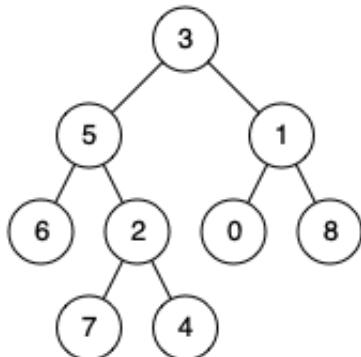
这个方法耗时比前面的两种方法要高，虽然看起来简单，但是编写的过程中有一点陷阱。

236. 二叉树的最近公共祖先 Medium

给定一个二叉树，找到该树中两个指定节点的最近公共祖先。

[百度百科](#)中最近公共祖先的定义为：“对于有根树 T 的两个节点 p、q，最近公共祖先表示为一个节点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

示例 1：

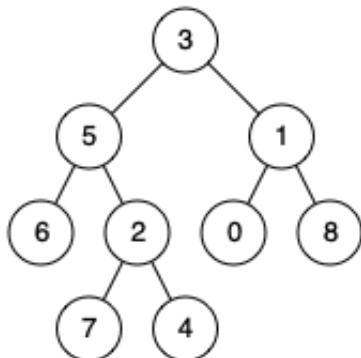


输入：root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

输出：3

解释：节点 5 和节点 1 的最近公共祖先是节点 3。

示例 2：



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

输出: 5

解释: 节点 5 和节点 4 的最近公共祖先是节点 5 。因为根据定义最近公共祖先节点可以为节点本身。

示例 3:

输入: root = [1,2], p = 1, q = 2

输出: 1

提示:

- 树中节点数目在范围 [2, 105] 内。
- $-10^9 \leq \text{Node.val} \leq 10^9$
- 所有 `Node.val` 互不相同。
- $p \neq q$
- `p` 和 `q` 均存在于给定的二叉树中。

```
class Solution {  
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {  
        if (root == null)  
            return null;  
        if (root.val == p.val || root.val == q.val)  
            return root;  
        else if (hasChild(root.left, p, q) && hasChild(root.right, p, q))  
            return root;  
        return lowestCommonAncestor(root.left, p, q) == null ?  
lowestCommonAncestor(root.right, p, q) : null;  
    }  
  
    public boolean hasChild(TreeNode node, TreeNode p, TreeNode q) {  
        if (node == null)  
            return false;  
        if (node.val == p.val || node.val == q.val)  
            return true;  
        return hasChild(node.left, p, q) || hasChild(node.right, p, q);  
    }  
}
```

超时

方法二：哈希表存储父节点

思路：

我们可以用哈希表存储所有节点的父节点，然后我们就可以利用节点的父节点信息从 p 结点开始不断往上跳，并记录已经访问过的节点，再从 q 节点开始不断往上跳，如果碰到已经访问过的节点，那么这个节点就是我们要找的最近公共祖先。

算法：

- 1、从根节点开始遍历整棵二叉树，用哈希表记录每个节点的父节点指针。
- 2、从 p 节点开始不断往它的祖先移动，并用数据结构记录已经访问过的祖先节点。
- 3、同样，我们再从 q 节点开始不断往它的祖先移动，如果有祖先已经被访问过，即意味着这是 p 和 q 的深度最深的公共祖先，即 LCA 节点。

```
class Solution {  
    HashMap<Integer, TreeNode> parents = new HashMap<>();  
    HashSet<Integer> visited = new HashSet<>();  
  
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {  
        dfs(root);  
        while (p != null) {  
            visited.add(p.val);  
            p = parents.get(p.val);  
        }  
        while (q != null) {  
            if (visited.contains(q.val))  
                return q;  
            q = parents.get(q.val);  
        }  
        return null;  
    }  
  
    private void dfs(TreeNode node) {  
        if (node == null)  
            return;  
        if (node.left != null)  
            parents.put(node.left.val, node);  
        if (node.right != null)  
            parents.put(node.right.val, node);  
        dfs(node.left);  
        dfs(node.right);  
    }  
}
```

```
class Solution {  
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {  
        if (root == null || root == p || root == q)  
            return root;
```

```

TreeNode left = lowestCommonAncestor(root.left, p, q);
TreeNode right = lowestCommonAncestor(root.right, p, q);
if (left != null && right != null)
    return root;
else if (left != null)
    return left;
else if (right != null)
    return right;
else
    return null;
}
}

```

142. 环形链表 II Medium

给定一个链表，返回链表开始入环的第一个节点。如果链表无环，则返回 `null`。

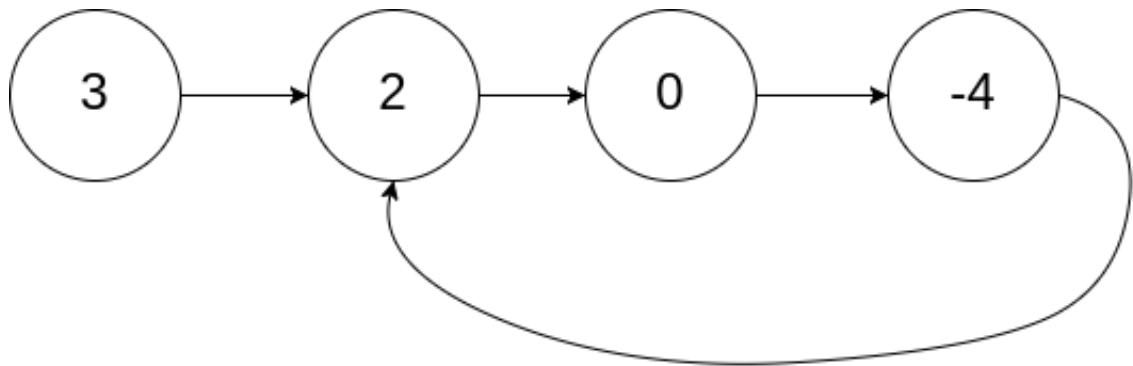
为了表示给定链表中的环，我们使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。如果 `pos` 是 `-1`，则在该链表中没有环。注意，`pos` 仅仅是用于标识环的情况，并不会作为参数传递到函数中。

说明：不允许修改给定的链表。

进阶：

- 你是否可以使用 $O(1)$ 空间解决此题？

示例 1：

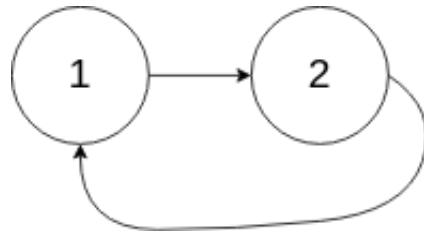


输入: `head = [3,2,0,-4], pos = 1`

输出: 返回索引为 1 的链表节点

解释: 链表中有一个环，其尾部连接到第二个节点。

示例 2：

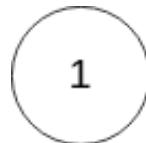


输入: head = [1,2], pos = 0

输出: 返回索引为 0 的链表节点

解释: 链表中有一个环, 其尾部连接到第一个节点。

示例 3:



输入: head = [1], pos = -1

输出: 返回 null

解释: 链表中没有环。

提示:

- 链表中节点的数目范围在范围 [0, 104] 内
- $-105 \leq \text{Node.val} \leq 105$
- pos 的值为 -1 或者链表中的一个有效索引

```
public class Solution {
    public ListNode detectCycle(ListNode head) {
        if (head == null || head.next == null)
            return null;
        ListNode slow = head, fast = head;
        do {
            if (fast != null && fast.next != null) {
                slow = slow.next;
                fast = fast.next.next;
            } else
                return null;
        } while (slow != fast);
        slow = head;
        while (slow != fast) {
            slow = slow.next;
            fast = fast.next;
        }
        return slow;
    }
}
```

46. 全排列 Medium

给定一个不含重复数字的数组 `nums`，返回其 所有可能的全排列。你可以 按任意顺序 返回答案。

示例 1：

```
输入: nums = [1,2,3]
输出: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

示例 2：

```
输入: nums = [0,1]
输出: [[0,1],[1,0]]
```

示例 3：

```
输入: nums = [1]
输出: [[1]]
```

提示：

- `1 <= nums.length <= 6`
- `-10 <= nums[i] <= 10`
- `nums` 中的所有整数 互不相同

```
class Solution {
    List<List<Integer>> res;
    List<Integer> item;

    public List<List<Integer>> permute(int[] nums) {
        res = new ArrayList<>();
        item = new ArrayList<>();
        boolean[] visited = new boolean[nums.length];
        backtracking(nums, 0, nums.length - 1, visited);
        return res;
    }

    private void backtracking(int[] nums, int curDigit, int totalDigit, boolean[] visited) {
        if (curDigit > totalDigit) {
            res.add(new ArrayList<>(item));
            return;
        }
        for (int i = 0; i < nums.length; i++) {
            if (visited[i])
                continue;
            item.add(nums[i]);
            visited[i] = true;
            backtracking(nums, curDigit + 1, totalDigit, visited);
            item.remove(item.size() - 1);
            visited[i] = false;
        }
    }
}
```

```

        if (visited[i])
            continue;
        visited[i] = true;
        item.add(nums[i]);
        backtracking(nums, curDigit + 1, totalDigit, visited);
        visited[i] = false;
        item.remove(curDigit);
    }
}
}

```

47. 全排列 II Medium

给定一个可包含重复数字的序列 `nums`，按任意顺序 返回所有不重复的全排列。

示例 1：

输入: `nums = [1,1,2]`

输出:

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

示例 2：

输入: `nums = [1,2,3]`

输出: `[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]`

提示：

- `1 <= nums.length <= 8`
- `-10 <= nums[i] <= 10`

```

class Solution {
    List<List<Integer>> res;

    public List<List<Integer>> permuteUnique(int[] nums) {
        res = new ArrayList<>();
        boolean[] visited = new boolean[nums.length];
        Arrays.sort(nums);
        backtracking(nums, visited, new ArrayList<>());
        return res;
    }
}

```

```

private void backtracking(int[] nums, boolean[] visited, List<Integer> list) {
    if (list.size() == nums.length) {
        res.add(new ArrayList<>(list));
        return;
    }
    for (int i = 0; i < nums.length; i++) {
        if (visited[i] || i > 0 && !visited[i - 1] && nums[i] == nums[i - 1])
            continue;
        visited[i] = true;
        list.add(nums[i]);
        backtracking(nums, visited, list);
        visited[i] = false;
        list.remove(list.size() - 1);
    }
}

```

33. 搜索旋转排序数组 Medium

整数数组 `nums` 按升序排列，数组中的值互不相同。

在传递给函数之前，`nums` 在预先未知的某个下标 `k` ($0 \leq k < \text{nums.length}$) 上进行了 **旋转**，使数组变为 `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]`（下标从 0 开始计数）。例如，`[0,1,2,4,5,6,7]` 在下标 `3` 处经旋转后可能变为 `[4,5,6,7,0,1,2]`。

给你 **旋转后** 的数组 `nums` 和一个整数 `target`，如果 `nums` 中存在这个目标值 `target`，则返回它的下标，否则返回 `-1`。

示例 1：

```

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

```

示例 2：

```

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

```

示例 3：

```

输入: nums = [1], target = 0
输出: -1

```

提示：

- $1 \leq \text{nums.length} \leq 5000$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- `nums` 中的每个值都 **独一无二**
- 题目数据保证 `nums` 在预先未知的某个下标上进行了旋转
- $-10^4 \leq \text{target} \leq 10^4$

进阶：你可以设计一个时间复杂度为 $O(\log n)$ 的解决方案吗？

```
class Solution {
    public int search(int[] nums, int target) {
        return biSearch(nums, 0, nums.length - 1, target);
    }

    private int biSearch(int[] nums, int left, int right, int target) {
        while (left <= right) {
            int mid = (left + right) / 2;
            if (nums[mid] == target)
                return mid;
            if (nums[left] <= nums[mid])
                if (nums[left] <= target && nums[mid] > target)
                    right = mid - 1;
                else
                    left = mid + 1;
            else if (nums[mid] <= nums[right])
                if (nums[mid] < target && nums[right] >= target)
                    left = mid + 1;
                else
                    right = mid - 1;
            else
                break;
        }
        return -1;
    }
}
```

```
class Solution {
    public int search(int[] nums, int target) {
        return biSearch(nums, 0, nums.length - 1, target);
    }

    private int biSearch(int[] nums, int left, int right, int target) {
        if (left > right)
            return -1;
        while (left <= right) {
            int mid = left + right >> 1;
            if (nums[mid] == target)
                return mid;
```

```

        if (nums[left] == nums[mid])
            left++;
        else if (nums[left] < nums[mid]) {
            if (nums[left] <= target && target < nums[mid])
                right = mid - 1;
            else
                left = mid + 1;
        } else {
            if (nums[mid] < target && nums[right] >= target)
                left = mid + 1;
            else
                right = mid - 1;
        }
    }
    return -1;
}
}

```

81. 搜索旋转排序数组 II Medium

已知存在一个按非降序排列的整数数组 `nums`，数组中的值不必互不相同。

在传递给函数之前，`nums` 在预先未知的某个下标 `k` ($0 \leq k < \text{nums.length}$) 上进行了 **旋转**，使数组变为 `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]`（下标从 0 开始计数）。例如，`[0,1,2,4,4,4,5,6,6,7]` 在下标 5 处经旋转后可能变为 `[4,5,6,6,7,0,1,2,4,4]`。

给你 **旋转后** 的数组 `nums` 和一个整数 `target`，请你编写一个函数来判断给定的目标值是否存在于数组中。如果 `nums` 中存在这个目标值 `target`，则返回 `true`，否则返回 `false`。

示例 1：

```

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

```

示例 2：

```

输入: nums = [2,5,6,0,0,1,2], target = 3
输出: false

```

提示：

- $1 \leq \text{nums.length} \leq 5000$
- $-104 \leq \text{nums}[i] \leq 104$
- 题目数据保证 `nums` 在预先未知的某个下标上进行了旋转
- $-104 \leq \text{target} \leq 104$

进阶：

- 这是 [搜索旋转排序数组](#) 的延伸题目，本题中的 `nums` 可能包含重复元素。
- 这会影响到程序的时间复杂度吗？会有怎样的影响，为什么？

```
class Solution {  
    public boolean search(int[] nums, int target) {  
        int left = 0, right = nums.length - 1;  
        while (left <= right) {  
            int mid = (left + right) >> 1;  
            if (nums[mid] == target)  
                return true;  
            if (nums[left] == nums[mid])  
                left++;  
            else if (nums[mid] <= nums[right]) {  
                if (nums[mid] <= target && target <= nums[right])  
                    left = mid + 1;  
                else  
                    right = mid - 1;  
            } else if (nums[mid] >= nums[left]) {  
                if (nums[left] <= target && target <= nums[mid])  
                    right = mid - 1;  
                else  
                    left = mid + 1;  
            }  
        }  
        return false;  
    }  
}
```

对于数组中有重复元素的情况，二分查找时可能会有 $a[\text{left}] = a[\text{mid}] = a[\text{right}]$ ，此时无法判断区间 $[\text{left}, \text{mid}]$ 和区间 $[\text{mid}+1, \text{right}]$ 哪个是有序的。

对于这种情况，我们只能将当前二分区间的左边界加一，右边界减一，然后在新区间上继续二分查找。

5. 最长回文子串 Medium

给你一个字符串 `s`，找到 `s` 中最长的回文子串。

示例 1：

```
输入: s = "babad"  
输出: "bab"  
解释: "aba" 同样是符合题意的答案。
```

示例 2:

```
输入: s = "cbbd"
输出: "bb"
```

示例 3:

```
输入: s = "a"
输出: "a"
```

示例 4:

```
输入: s = "ac"
输出: "a"
```

提示:

- $1 \leq s.length \leq 1000$
- s 仅由数字和英文字母（大写和/或小写）组成

```
public class Solution {
    public String longestPalindrome(String s) {
        int len = s.length();
        boolean[][] dp = new boolean[len][len];
        int left = 0, right = 0;
        for (int i = 0; i < dp.length; i++) {
            dp[i][i] = true;
        }
        for (int j = 1; j < dp.length; j++) {
            for (int i = 0; i < j; i++) {
                dp[i][j] = (i + 2 >= j || dp[i + 1][j - 1]) && s.charAt(i) ==
s.charAt(j);
                if (dp[i][j] && j - i + 1 > right - left + 1) {
                    left = i;
                    right = j;
                }
            }
        }
        return s.substring(left, right + 1);
    }
}

//动态规划，建一个二维矩阵，矩阵的右上三角部分有意义需要进行计算，左下部分无实际意义。
//矩阵dp[i][j]的意思：起点为i，终点为j的子串（包括两个端点）是否是palindrome。
//本题计算过程中，两重循环处，先对列进行遍历，内循环为同一列情况下每一行的情况计算。
```

```
class Solution {
```

```

public String longestPalindrome(String s) {
    int left = 0, right = 0;
    for (int i = 0; i < s.length(); i++) {
        int oddLength = expandAroundCenter(s, i, i);
        int evenLength = expandAroundCenter(s, i, i + 1);
        int len = Math.max(oddLength, evenLength);
        if (len > right - left + 1) {
            left = i - (len - 1) / 2;
            right = i + len / 2;
        }
    }
    return s.substring(left, right + 1);
}

private int expandAroundCenter(String s, int left, int right) {
    while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
        left--;
        right++;
    }
    return right - left - 1;
}
}

//把字符串s转化为char数组可以优化时间。
//以上两种方法的时间复杂度都是平方阶，这种方法叫做中心扩散。
//本题还有一个时间复杂度为线性阶的解法Manacher算法，暂不收录。

```

```

class Solution {
    int left = 0, right = 0;
    int len = -1;

    public String longestPalindrome(String s) {
        char[] cs = s.toCharArray();
        for (int i = 0; i < cs.length - 1; i++) {
            centerDiffuse(cs, i, i);
            centerDiffuse(cs, i, i + 1);
        }
        return s.substring(left, right + 1);
    }

    private void centerDiffuse(char[] cs, int i, int j) {
        while (i >= 0 && j <= cs.length - 1) {
            if (cs[i] != cs[j])
                break;
            i--;
            j++;
        }
        if (j - i - 1 > len) {
            left = i + 1;
            len = j - i - 1;
        }
    }
}

```

```

        right = j - 1;
        len = j - i - 1;
    }
}
}

```

200. 岛屿数量 Medium

给你一个由 '1' (陆地) 和 '0' (水) 组成的二维网格，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。

示例 1:

```

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

```

示例 2:

```

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

```

提示:

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 300`
- `grid[i][j]` 的值为 '0' 或 '1'

```

class Solution {
    public int numIslands(char[][] grid) {
        boolean[][] visited = new boolean[grid.length][grid[0].length];
        int cnt = 0;

```

```

        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[i].length; j++) {
                if (grid[i][j] == '1' && !visited[i][j]) {
                    dfs(grid, visited, i, j);
                    cnt++;
                }
            }
        }
        return cnt;
    }

    private void dfs(char[][] grid, boolean[][] visited, int i, int j) {
        if (i < 0 || i >= grid.length || j < 0 || j >= grid[i].length || grid[i][j] == '0' || visited[i][j])
            return;
        visited[i][j] = true;
        dfs(grid, visited, i + 1, j);
        dfs(grid, visited, i - 1, j);
        dfs(grid, visited, i, j + 1);
        dfs(grid, visited, i, j - 1);
    }
}

```

除了`dfs`外，本题还有`bfs`、`并查集`两种解法。

300. 最长递增子序列 Medium

给你一个整数数组 `nums`，找到其中最长严格递增子序列的长度。

子序列是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，`[3,6,2,7]` 是数组 `[0,3,1,6,2,2,7]` 的子序列。

示例 1：

```

输入: nums = [10,9,2,5,3,7,101,18]
输出: 4
解释: 最长递增子序列是 [2,3,7,101]，因此长度为 4。

```

示例 2：

```

输入: nums = [0,1,0,3,2,3]
输出: 4

```

示例 3：

输入: nums = [7,7,7,7,7,7,7,7]
输出: 1

提示：

- `1 <= nums.length <= 2500`
 - `-104 <= nums[i] <= 104`

进阶：

- 你可以设计时间复杂度为 $O(n^2)$ 的解决方案吗？
 - 你能将算法的时间复杂度降低到 $O(n \log(n))$ 吗？

```
class Solution {
    public int lengthOfLIS(int[] nums) {
        int[] dp = new int[nums.length];
        int len = 1;
        for (int j = 0; j < nums.length; j++) {
            dp[j] = 1;
            for (int i = 0; i < j; i++) {
                if (nums[i] < nums[j])
                    dp[j] = Math.max(dp[j], dp[i] + 1);
            }
            len = Math.max(len, dp[j]);
        }
        return len;
    }
}
//动态规划，时间复杂度为平方阶。
```

```
class Solution {
    public int lengthOfLIS(int[] nums) {
        int len = nums.length;
        int[] subSequence = new int[len];
        int lastEleIndex = 0;
        subSequence[lastEleIndex] = nums[0];
        for (int i = 1; i < subSequence.length; i++) {
            if (nums[i] > subSequence[lastEleIndex])
                subSequence[++lastEleIndex] = nums[i];
            else if (nums[i] == subSequence[lastEleIndex])
                continue;
            else {
                int left = 0, right = lastEleIndex;
                boolean isEqual = false;
                while (left < right) {
                    int mid = (left + right) / 2;
                    if (subSequence[mid] == nums[i]) {
                        isEqual = true;
                        break;
                    } else if (subSequence[mid] < nums[i])
                        left = mid + 1;
                    else
                        right = mid;
                }
                if (!isEqual)
                    subSequence[++lastEleIndex] = nums[i];
            }
        }
        return lastEleIndex + 1;
    }
}
```

```

        isEqual = true;
        break;
    } else if (subSequence[mid] < nums[i])
        left = mid + 1;
    else if (subSequence[mid] > nums[i])
        right = mid;
    }
    if (!isEqual)
        subSequence[left] = nums[i];
}
}

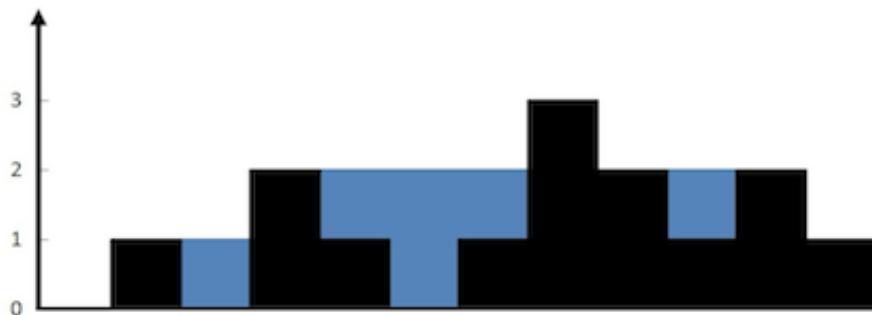
return ++lastEleIndex;
}
}

```

42. 接雨水 Hard

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

示例 1：



输入：height = [0,1,0,2,1,0,1,3,2,1,2,1]

输出：6

解释：上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。

示例 2：

输入：height = [4,2,0,3,2,5]

输出：9

提示：

- $n == \text{height.length}$
- $0 \leq n \leq 3 * 10^4$
- $0 \leq \text{height}[i] \leq 105$

```

class Solution {
    public int trap(int[] height) {
        int volume = 0;
        int leftBorder, rightBorder;
        for (int i = 1; i < height.length - 1; i++) {
            leftBorder = getMaxLeftHeight(height, i - 1);
            rightBorder = getMaxRightHeight(height, i + 1);
            volume += Math.max(0, Math.min(leftBorder, rightBorder) - height[i]);
        }
        return volume;
    }

    private int getMaxRightHeight(int[] height, int i) {
        int maxHeight = 0;
        while (i < height.length) {
            maxHeight = Math.max(maxHeight, height[i++]);
        }
        return maxHeight;
    }

    private int getMaxLeftHeight(int[] height, int i) {
        int maxHeight = 0;
        while (i >= 0) {
            maxHeight = Math.max(maxHeight, height[i--]);
        }
        return maxHeight;
    }
}

```

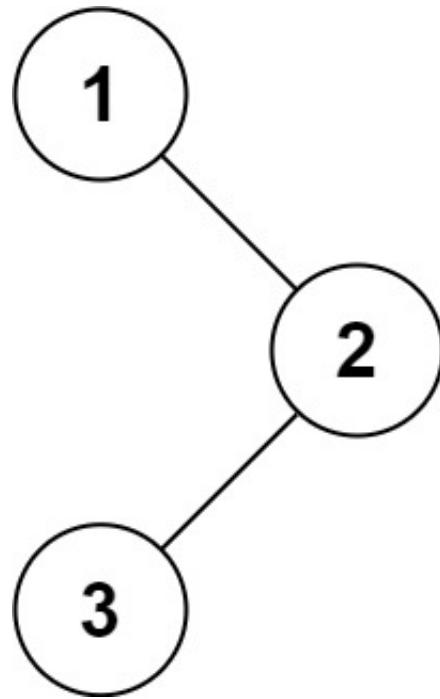
每个位置可以接雨水的容量取决于该柱子左右两边最大的柱子高度。这道题至少有四种解法：

- 1、暴力解法，具体实现如上。时间复杂度为平方阶，空间复杂度为常数。
- 2、对暴力算法进行优化，先对数组遍历两次得到每一个柱子左右最大柱子高度并保存在数组中。时间复杂度为线性阶，空间复杂度为线性阶，以空间换时间的方法。
- 3、继续优化，将空间复杂度降到常数阶。使用双指针，具体做法不表。
- 4、使用栈，具体做法不表。官方题解时间复杂度为线性阶，空间复杂度为线性阶。

94. 二叉树的中序遍历 Easy

给定一个二叉树的根节点 `root`，返回它的 中序 遍历。

示例 1：



输入: `root = [1,null,2,3]`
输出: `[1,3,2]`

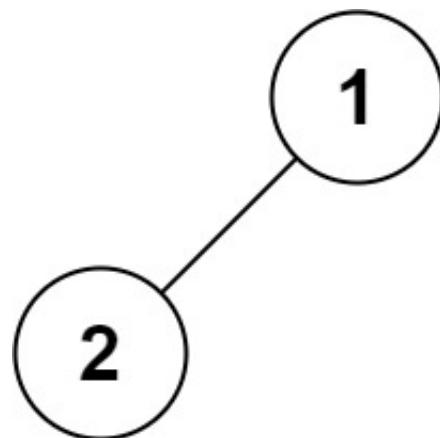
示例 2:

输入: `root = []`
输出: `[]`

示例 3:

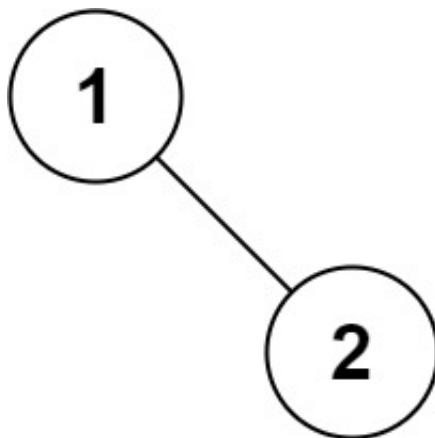
输入: `root = [1]`
输出: `[1]`

示例 4:



输入: `root = [1,2]`
输出: `[2,1]`

示例 5:



```
输入: root = [1,null,2]
输出: [1,2]
```

提示:

- 树中节点数目在范围 `[0, 100]` 内
- `-100 <= Node.val <= 100`

进阶: 递归算法很简单, 你可以通过迭代算法完成吗?

```
class Solution {
    List<Integer> res = new ArrayList<>();

    public List<Integer> inorderTraversal(TreeNode root) {
        if (root == null)
            return res;
        inorderTraversal(root.left);
        res.add(root.val);
        inorderTraversal(root.right);
        return res;
    }
}
```

```
class Solution {

    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();
        TreeNode cur = root;
        while (!stack.isEmpty() || cur != null) {
            while (cur != null) {
                stack.push(cur);
                cur = cur.left;
            }
        }
    }
}
```

```

        TreeNode pop = stack.pop();
        res.add(pop.val);
        cur = pop.right;
    }
    return res;
}
}

```

Morris遍历，先写这里，慢慢看吧。

```

class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<Integer>();
        TreeNode predecessor = null;

        while (root != null) {
            if (root.left != null) {
                // predecessor 节点就是当前 root 节点向左走一步，然后一直向右走至无法走为止
                predecessor = root.left;
                while (predecessor.right != null && predecessor.right != root) {
                    predecessor = predecessor.right;
                }

                // 让 predecessor 的右指针指向 root，继续遍历左子树
                if (predecessor.right == null) {
                    predecessor.right = root;
                    root = root.left;
                }
                // 说明左子树已经访问完了，我们需要断开链接
                else {
                    res.add(root.val);
                    predecessor.right = null;
                    root = root.right;
                }
            }
            // 如果没有左孩子，则直接访问右孩子
            else {
                res.add(root.val);
                root = root.right;
            }
        }
        return res;
    }
}

```

70. 爬楼梯 Easy

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？

注意：给定 n 是一个正整数。

示例 1：

```
输入: 2
输出: 2
解释: 有两种方法可以爬到楼顶。
1. 1 阶 + 1 阶
2. 2 阶
```

示例 2：

```
输入: 3
输出: 3
解释: 有三种方法可以爬到楼顶。
1. 1 阶 + 1 阶 + 1 阶
2. 1 阶 + 2 阶
3. 2 阶 + 1 阶
```

```
class Solution {
    public int climbStairs(int n) {
        if (n <= 3)
            return n;
        int a = 1, b = 2, temp;
        for (int i = 3; i <= n; i++) {
            temp = a + b;
            a = b;
            b = temp;
        }
        return b;
    }
}
```

```

class Solution {
    public int climbStairs(int n) {
        if (n <= 3)
            return n;
        int[] dp = new int[n + 1];
        dp[1] = 1;
        dp[2] = 2;
        for (int i = 3; i < dp.length; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }
        return dp[n];
    }
}

```

```

class Solution {
    HashMap<Integer, Integer> hashMap = new HashMap<>();

    public int climbStairs(int n) {
        if (n <= 3)
            return n;
        if (hashMap.containsKey(n))
            return hashMap.get(n);
        hashMap.put(n, climbStairs(n - 1) + climbStairs(n - 2));
        return hashMap.get(n);
    }
}

//这里hashMap一定要放到方法外面，否则时间复杂度为指数级。

```

23. 合并K个升序链表 Hard

给你一个链表数组，每个链表都已经按升序排列。

请你将所有链表合并到一个升序链表中，返回合并后的链表。

示例 1：

```
输入: lists = [[1,4,5],[1,3,4],[2,6]]
```

```
输出: [1,1,2,3,4,4,5,6]
```

解释: 链表数组如下:

```
[  
    1->4->5,  
    1->3->4,  
    2->6  
]
```

将它们合并到一个有序链表中得到。

```
1->1->2->3->4->4->5->6
```

示例 2:

```
输入: lists = []
```

```
输出: []
```

示例 3:

```
输入: lists = [[]]
```

```
输出: []
```

提示:

- `k == lists.length`
- `0 <= k <= 10^4`
- `0 <= lists[i].length <= 500`
- `-10^4 <= lists[i][j] <= 10^4`
- `lists[i]` 按升序排列
- `lists[i].length` 的总和不超过 `10^4`

```
class Solution {  
    public ListNode mergeKLists(ListNode[] lists) {  
        if (lists.length == 0)  
            return null;  
        return divideAndConquer(lists, 0, lists.length - 1);  
    }  
  
    private ListNode divideAndConquer(ListNode[] lists, int left, int right) {  
        if (left == right)  
            return lists[left];  
        int mid = left + right >> 1;  
        ListNode leftNode = divideAndConquer(lists, left, mid);  
        ListNode rightNode = divideAndConquer(lists, mid + 1, right);  
        return mergeLists(leftNode, rightNode);  
    }  
}
```

```

private ListNode mergeLists(ListNode leftNode, ListNode rightNode) {
    ListNode dummyHead = new ListNode();
    ListNode cur = dummyHead;
    while (leftNode != null && rightNode != null) {
        if (leftNode.val < rightNode.val) {
            cur.next = leftNode;
            leftNode = leftNode.next;
        } else {
            cur.next = rightNode;
            rightNode = rightNode.next;
        }
        cur = cur.next;
    }
    cur.next = leftNode == null ? rightNode : leftNode;
    return dummyHead.next;
}

```

//这道题采用分治方法，还可以采用优先队列的做法，试试手写堆不采用自带的PriorityQueue类。

```

class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        PriorityQueue<Integer> pq = new PriorityQueue<>(new Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
                return o1 - o2;
            }
        });
        for (ListNode list : lists) {
            while (list != null) {
                pq.offer(list.val);
                list = list.next;
            }
        }
        ListNode dummyHead = new ListNode();
        ListNode cur = dummyHead;
        while (!pq.isEmpty()) {
            cur.next = new ListNode(pq.poll());
            cur = cur.next;
        }
        return dummyHead.next;
    }
}

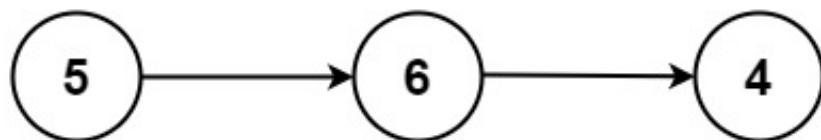
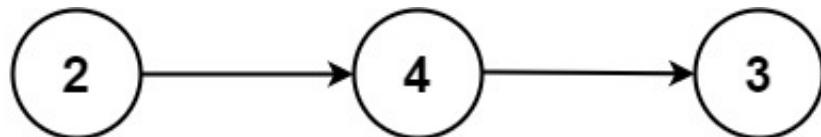
```

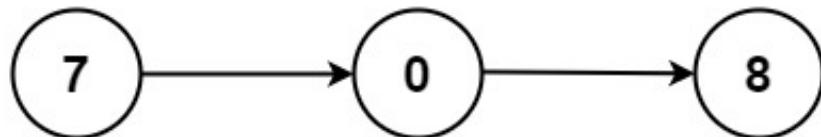
2. 两数相加 Medium

给你两个 **非空** 的链表，表示两个非负的整数。它们每位数字都是按照 **逆序** 的方式存储的，并且每个节点只能存储一位 数字。

请你将两个数相加，并以相同形式返回一个表示和的链表。你可以假设除了数字 0 之外，这两个数都不会以 0 开头。

示例 1：





输入: $11 = [2, 4, 3]$, $12 = [5, 6, 4]$

输出: $[7, 0, 8]$

解释: $342 + 465 = 807$.

示例 2：

输入: $11 = [0]$, $12 = [0]$

输出: $[0]$

示例 3：

输入: $11 = [9, 9, 9, 9, 9, 9, 9]$, $12 = [9, 9, 9, 9]$

输出: $[8, 9, 9, 9, 0, 0, 0, 1]$

提示：

- 每个链表中的节点数在范围 `[1, 100]` 内
- `0 <= Node.val <= 9`
- 题目数据保证列表表示的数字不含前导零

```

class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        int carry = 0;
        ListNode dummyHead = new ListNode();
        ListNode cur = dummyHead;
        while (l1 != null || l2 != null || carry != 0) {
            int num1 = l1 == null ? 0 : l1.val;
            int num2 = l2 == null ? 0 : l2.val;
            ListNode node = new ListNode((num1 + num2 + carry) % 10);
            carry = (num1 + num2 + carry) / 10;
            cur.next = node;
            cur = cur.next;
            l1 = l1 == null ? l1 : l1.next;
            l2 = l2 == null ? l2 : l2.next;
        }
        return dummyHead.next;
    }
}

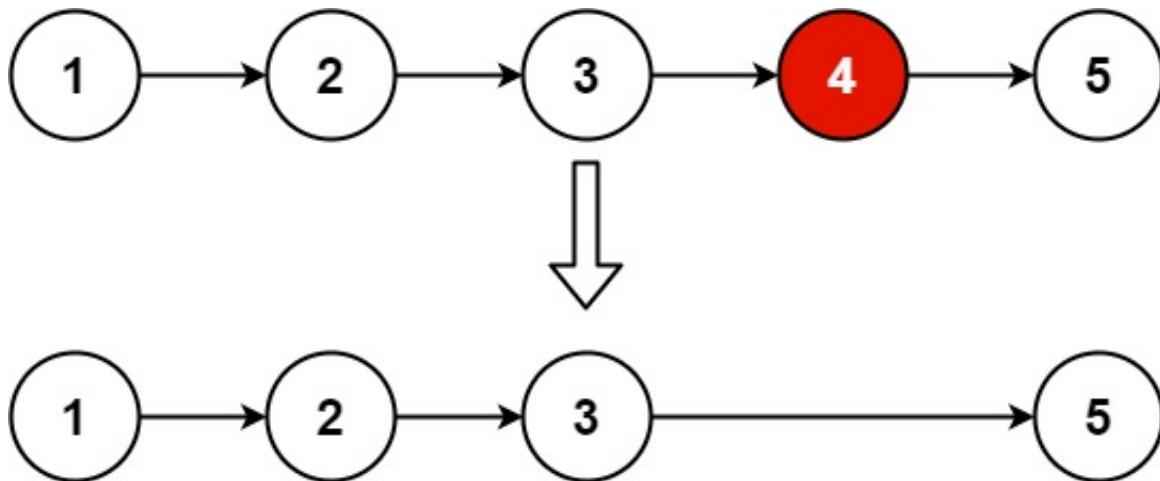
```

19. 删除链表的倒数第 N 个结点 Medium

给你一个链表，删除链表的倒数第 n 个结点，并且返回链表的头结点。

进阶：你能尝试使用一趟扫描实现吗？

示例 1：



输入: head = [1,2,3,4,5], n = 2

输出: [1,2,3,5]

示例 2：

```
输入: head = [1], n = 1
输出: []
```

示例 3:

```
输入: head = [1,2], n = 1
输出: [1]
```

提示:

- 链表中结点的数目为 `sz`
- `1 <= sz <= 30`
- `0 <= Node.val <= 100`
- `1 <= n <= sz`

```
class Solution {
    private int count = 0;

    public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode dummyHead = new ListNode(0, head);
        backwardTraverse(dummyHead, n);
        return dummyHead.next;
    }

    private void backwardTraverse(ListNode node, int n) {
        if (node.next != null)
            backwardTraverse(node.next, n);
        if (++count == n + 1)
            node.next = node.next.next;
    }
}
```

```
class Solution {

    public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode dummyHead = new ListNode(0, head);
        ListNode fast = head;
        ListNode slow = dummyHead;
        while (n-- > 0) {
            fast = fast.next;
        }
        while (fast != null) {
            fast = fast.next;
            slow = slow.next;
        }
        slow.next = slow.next.next;
    }
}
```

```
    return dummyHead.next;
}
}
```

方法一采用递归方式，递归到链表最后节点，开始计算当前节点是倒数第几个节点。

方法二采用快慢指针，快指针先走n步，然后快慢指针一起移动。

56. 合并区间 Medium

以数组 `intervals` 表示若干个区间的集合，其中单个区间为 `intervals[i] = [starti, endi]`。请你合并所有重叠的区间，并返回一个不重叠的区间数组，该数组需恰好覆盖输入中的所有区间。

示例 1：

```
输入: intervals = [[1,3],[2,6],[8,10],[15,18]]
输出: [[1,6],[8,10],[15,18]]
解释: 区间 [1,3] 和 [2,6] 重叠，将它们合并为 [1,6].
```

示例 2：

```
输入: intervals = [[1,4],[4,5]]
输出: [[1,5]]
解释: 区间 [1,4] 和 [4,5] 可被视为重叠区间。
```

提示：

- `1 <= intervals.length <= 104`
- `intervals[i].length == 2`
- `0 <= starti <= endi <= 104`

```
class Solution {
    public int[][] merge(int[][] intervals) {
        List<int[]> list = new ArrayList<>();
        Arrays.sort(intervals, new Comparator<int[]>() {
            @Override
            public int compare(int[] o1, int[] o2) {
                return o1[0] - o2[0];
            }
        });
        // Arrays.sort(intervals, Comparator.comparingInt(o -> o[0]));
        // Arrays.sort(intervals, (o1, o2) -> o1[0] - o2[0]);
        int leftBound = intervals[0][0];
        int rightBound = intervals[0][1];
        for (int i = 1; i < intervals.length; i++) {
            if (intervals[i][0] <= rightBound) {
                rightBound = Math.max(rightBound, intervals[i][1]);
            } else {
                list.add(new int[]{leftBound, rightBound});
                leftBound = intervals[i][0];
                rightBound = intervals[i][1];
            }
        }
        list.add(new int[]{leftBound, rightBound});
        return list.toArray(new int[list.size()][2]);
    }
}
```

```

        if (rightBound >= intervals[i][0])
            rightBound = Math.max(rightBound, intervals[i][1]);
        else {
            list.add(new int[] { leftBound, rightBound });
            leftBound = intervals[i][0];
            rightBound = intervals[i][1];
        }
    }
    list.add(new int[] { leftBound, rightBound });

    int count = 0, size = list.size();
    int[][] res = new int[size][2];
    for (int[] item : list) {
        res[count++] = item;
    }
    return res;
}
}

```

```

class Solution {
    public int[][] merge(int[][] intervals) {
        Arrays.sort(intervals, new Comparator<int[]>() {
            @Override
            public int compare(int[] o1, int[] o2) {
                if (o1[0] == o2[0])
                    return o1[1] - o2[1];
                return o1[0] - o2[0];
            }
        });
        int left = intervals[0][0];
        int right = intervals[0][1];
        List<int[]> res = new ArrayList<>();
        for (int i = 1; i < intervals.length; i++) {
            if (right < intervals[i][0]) {
                res.add(new int[]{left, right});
                left = intervals[i][0];
            }
            right = Math.max(right, intervals[i][1]);
        }
        res.add(new int[]{left, right});
        return res.toArray(new int[res.size()][]);
    }
}

```

104. 二叉树的最大深度 Easy

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

说明: 叶子节点是指没有子节点的节点。

示例:

给定二叉树 [3,9,20,null,null,15,7]，

```
3
/ \
9  20
/   \
15   7
```

返回它的最大深度 3。

```
class Solution {
    public int maxDepth(TreeNode root) {
        return root == null ? 0 : Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;
    }
}
```

```
class Solution {
    public int maxDepth(TreeNode root) {
        int depth = 0;
        if (root == null)
            return depth;
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        while (!queue.isEmpty()) {
            int size = queue.size();
            while (size-- > 0) {
                TreeNode poll = queue.poll();
                if (poll.left != null)
                    queue.offer(poll.left);
                if (poll.right != null)
                    queue.offer(poll.right);
            }
            depth++;
        }
        return depth;
    }
}
```

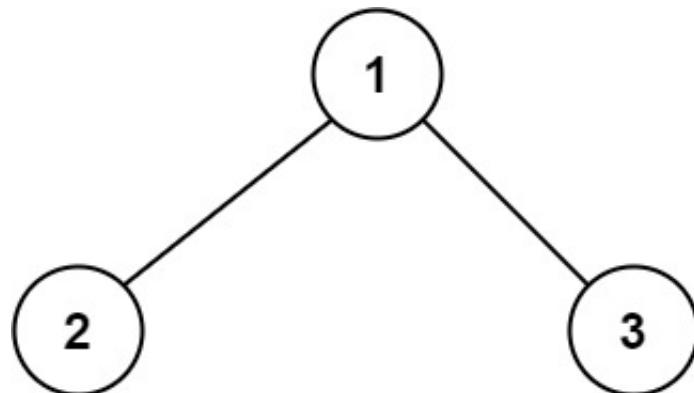
bfs用于求n叉树深度

124. 二叉树中的最大路径和 Hard

路径 被定义为一条从树中任意节点出发，沿父节点-子节点连接，达到任意节点的序列。同一个节点在一条路径序列中 至多出现一次 。该路径 至少包含一个 节点，且不一定经过根节点。

路径和 是路径中各节点值的总和。给你一个二叉树的根节点 `root`，返回其 最大路径和 。

示例 1：

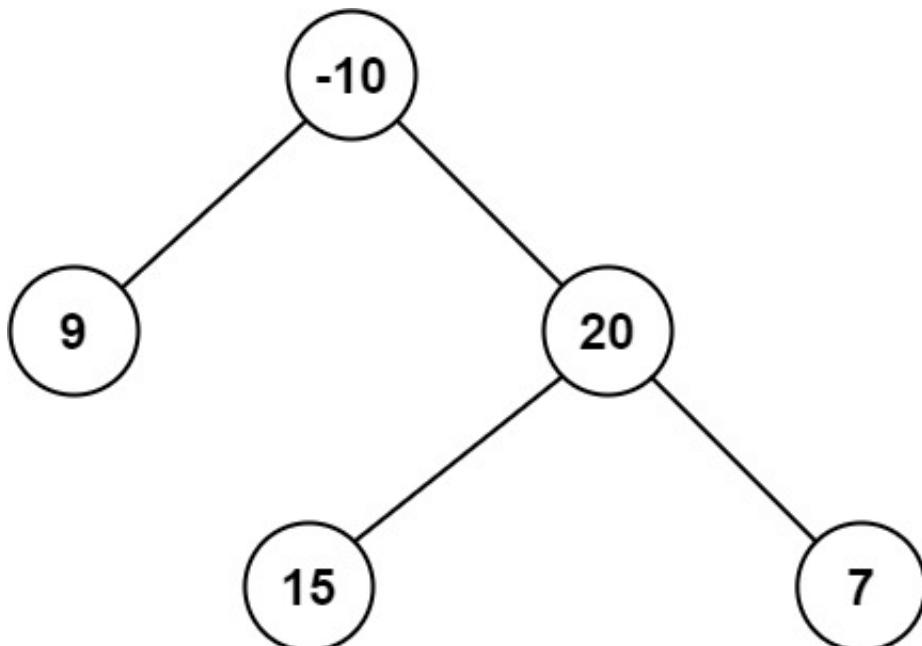


输入: `root = [1,2,3]`

输出: 6

解释: 最优路径是 $2 \rightarrow 1 \rightarrow 3$ ，路径和为 $2 + 1 + 3 = 6$

示例 2：



输入: `root = [-10,9,20,null,null,15,7]`

输出: 42

解释: 最优路径是 $15 \rightarrow 20 \rightarrow 7$ ，路径和为 $15 + 20 + 7 = 42$

提示:

- 树中节点数目范围是 `[1, 3 * 104]`
- `-1000 <= Node.val <= 1000`

```
class Solution {  
    private int pathSum = Integer.MIN_VALUE;  
  
    public int maxPathSum(TreeNode root) {  
        nodeGain(root);  
        return pathSum;  
    }  
  
    private int nodeGain(TreeNode root) {  
        if (root == null)  
            return 0;  
        int leftSonGain = Math.max(0, nodeGain(root.left));  
        int rightSonGain = Math.max(0, nodeGain(root.right));  
        pathSum = Math.max(pathSum, leftSonGain + root.val + rightSonGain);  
        return root.val + Math.max(leftSonGain, rightSonGain);  
    }  
}
```

31. 下一个排列 Medium

实现获取 **下一个排列** 的函数，算法需要将给定数字序列重新排列成字典序中下一个更大的排列（即，组合出下一个更大的整数）。

如果不存在下一个更大的排列，则将数字重新排列成最小的排列（即升序排列）。

必须原地修改，只允许使用额外常数空间。

示例 1:

```
输入: nums = [1,2,3]  
输出: [1,3,2]
```

示例 2:

```
输入: nums = [3,2,1]  
输出: [1,2,3]
```

示例 3:

```
输入: nums = [1,1,5]
输出: [1,5,1]
```

示例 4:

```
输入: nums = [1]
输出: [1]
```

提示:

- $1 \leq \text{nums.length} \leq 100$
- $0 \leq \text{nums}[i] \leq 100$

```
class Solution {
    public void nextPermutation(int[] nums) {
        int length = nums.length;
        int i = length - 2;
        while (i >= 0 && nums[i] >= nums[i + 1]) {
            i--;
        }
        if (i >= 0) {
            int j = length - 1;
            while (j > i && nums[i] >= nums[j]) {
                j--;
            }
            swap(nums, i, j);
        }
        reverse(nums, i + 1);
    }

    private void reverse(int[] nums, int starter) {
        int left = starter;
        int right = nums.length - 1;
        while (left < right) {
            swap(nums, left++, right--);
        }
    }

    private void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}
```

题目改一下，求上一个排列？

```

class Solution {
    public void nextPermutation(int[] nums) {
        for (int i = nums.length - 2; i >= 0; i--) {
            if (nums[i] > nums[i + 1]) {
                for (int j = i + 1; j < nums.length; j++) {
                    if (nums[j] < nums[i] && (j == nums.length - 1 || nums[j + 1] >= nums[i])) {
                        swap(nums, i, j);
                        reverse(nums, i + 1);
                        return;
                    }
                }
            }
        }
        reverse(nums, 0);
    }

    private void reverse(int[] nums, int starter) {
        int left = starter;
        int right = nums.length - 1;
        while (left < right) {
            swap(nums, left++, right--);
        }
    }

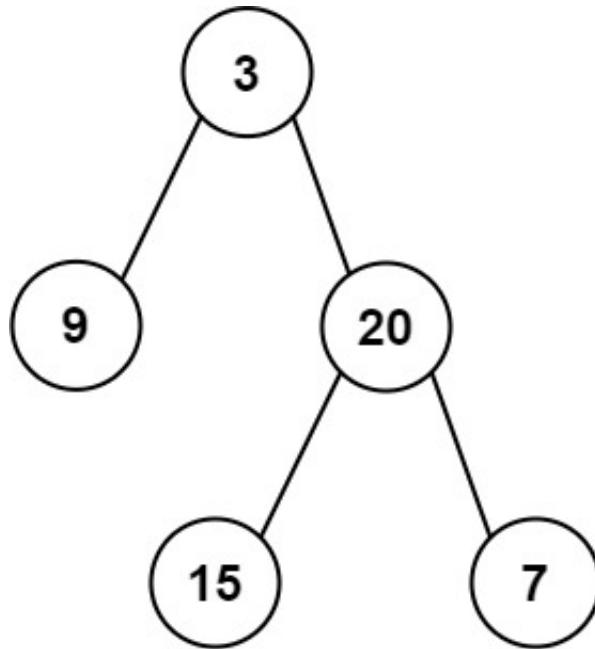
    private void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}

```

105. 从前序与中序遍历序列构造二叉树 Medium

给定一棵树的前序遍历 `preorder` 与中序遍历 `inorder`。请构造二叉树并返回其根节点。

示例 1:



Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
 Output: [3,9,20,null,null,15,7]

示例 2:

Input: preorder = [-1], inorder = [-1]
 Output: [-1]

提示:

- `1 <= preorder.length <= 3000`
- `inorder.length == preorder.length`
- `-3000 <= preorder[i], inorder[i] <= 3000`
- `preorder` 和 `inorder` 均无重复元素
- `inorder` 均出现在 `preorder`
- `preorder` 保证为二叉树的前序遍历序列
- `inorder` 保证为二叉树的中序遍历序列

```

class Solution {
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        int preLength = preorder.length;
        int inLength = inorder.length;
        int count = -1;
        if (preLength == 1 && inLength == 1)
            return new TreeNode(preorder[0]);
        if (preLength <= 0 || inLength <= 0)
            return null;

        TreeNode root = new TreeNode(preorder[0]);
        while (inorder[++count] != root.val)
    }
}
  
```

```

        ;
    int newLeftLen = count;
    int newRightLen = preLength - 1 - newLeftLen;
    int[] newPreLeft = new int[newLeftLen];
    int[] newPreRight = new int[newRightLen];
    int[] newInLeft = new int[newLeftLen];
    int[] newInRight = new int[newRightLen];

    System.arraycopy(preorder, 1, newPreLeft, 0, newLeftLen);
    System.arraycopy(preorder, newLeftLen + 1, newPreRight, 0, newRightLen);
    System.arraycopy(inorder, 0, newInLeft, 0, newLeftLen);
    System.arraycopy(inorder, newLeftLen + 1, newInRight, 0, newRightLen);

    root.left = buildTree(newPreLeft, newInLeft);
    root.right = buildTree(newPreRight, newInRight);
    return root;
}
}

```

上面的代码，注意⚠️：新的数组的长度，`arraycopy`方法的起始点，不需要用`HashMap`、循环来确定，想一下，利用一些固定的数组长度关系，就可以了。

上面的代码写得不好，可以做到简洁明了一点。时间复杂度为平方阶，可以用`HashMap`先扫描一遍`inorder`数组得到每一个数字在节点的具体下标，这样`preorder`中寻找每一个数字在`inorder`中的下标就不用再遍历了，时间复杂度降为线性阶。对于递归时左右边界的计算要仔细。

此题还可以使用迭代方法，暂不收录，因为不懂。

```

class Solution {

    private HashMap<Integer, Integer> hashMap = new HashMap<Integer, Integer>();

    public TreeNode buildTree(int[] preorder, int[] inorder) {
        int preLength = preorder.length;
        int inLength = inorder.length;

        for (int i = 0; i < inorder.length; i++) {
            hashMap.put(inorder[i], i);
        }

        return buildTree(preorder, 0, preLength - 1, inorder, 0, inLength - 1);
    }

    private TreeNode buildTree(int[] preorder, int preLeft, int preRight, int[]
inorder, int inLeft, int inRight) {
        int preLength = preRight - preLeft + 1;
        int inLength = inRight - inLeft + 1;
        if (preLength == 1 && inLength == 1)
            return new TreeNode(preorder[preLeft]);
        if (preLength <= 0 || inLength <= 0)

```

```

        return null;

TreeNode root = new TreeNode(preorder[preLeft]);
int prePivot = preLeft;
int inPivot = hashMap.get(root.val);

root.left = buildTree(preorder, prePivot + 1, preLeft + inPivot - inLeft,
inorder, inLeft, inPivot - 1);
root.right = buildTree(preorder, preLeft + inPivot - inLeft + 1, preRight,
inorder, inPivot + 1, inRight);

return root;
}
}

```

方法二：迭代

思路

迭代法是一种非常巧妙的实现方法。

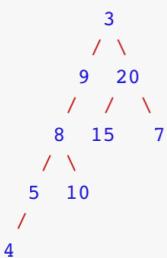
对于前序遍历中的任意两个连续节点 u 和 v ，根据前序遍历的流程，我们可以知道 u 和 v 只有两种可能的关系：

- v 是 u 的左儿子。这是因为在遍历到 u 之后，下一个遍历的节点就是 u 的左儿子，即 v ；
- u 没有左儿子，并且 v 是 u 的某个祖先节点（或者 u 本身）的右儿子。如果 u 没有左儿子，那么下一个遍历的节点就是 u 的右儿子。如果 u 没有右儿子，我们就会向上回溯，直到遇到第一个有右儿子（且 u 不在它的右儿子的子树中）的节点 u_a ，那么 v 就是 u_a 的右儿子。

第二种关系看上去有些复杂。我们举一个例子来说明其正确性，并在例子中给出我们的迭代算法。

例子

我们以树



为例，它的前序遍历和中序遍历分别为

```

preorder = [3, 9, 8, 5, 4, 10, 20, 15, 7]
inorder = [4, 5, 8, 10, 9, 3, 15, 20, 7]

```

我们用一个栈 `stack` 来维护「当前节点的所有还没有考虑过右儿子的祖先节点」，栈顶就是当前节点。也就是说，只有在栈中的节点才可能连接一个新的右儿子。同时，我们用一个指针 `index` 指向中序遍历的某个位置，初始值为 `0`。`index` 对应的节点是「当前节点不断往左走达到的最终节点」，这也是符合中序遍历的，它的作用在下面的过程中会有所体现。

首先我们将根节点 `3` 入栈，再初始化 `index` 所指向的节点为 `4`，随后对于前序遍历中的每个节点，我们依次判断它是栈顶节点的左儿子，还是栈中某个节点的右儿子。

- 我们遍历 `9`，`9` 一定是栈顶节点 `3` 的左儿子。我们使用反证法。假设 `9` 是 `3` 的右儿子，那么

3 没有左儿子，`index` 应该恰好指向 3，但实际上为 4，因此产生了矛盾。所以我们将 9 作为 3 的左儿子，并将 9 入栈。

- `stack = [3, 9]`
- `index -> inorder[0] = 4`

- 我们遍历 8，5 和 4。同理可得它们都是上一个节点（栈顶节点）的左儿子，所以它们会依次入栈。

- `stack = [3, 9, 8, 5, 4]`
- `index -> inorder[0] = 4`

- 我们遍历 10，这时情况就不一样了。我们发现 `index` 恰好指向当前的栈顶节点 4，也就是说 4 没有左儿子，那么 10 必须为栈中某个节点的右儿子。那么如何找到这个节点呢？栈中的节点的顺序和它们在前序遍历中出现的顺序是一致的，而且每一个节点的右儿子都还没有被遍历过，那么这些节点的顺序和它们在中序遍历中出现的顺序一定是相反的。

这是因为栈中的任意两个相邻的节点，前者都是后者的某个祖先。并且我们知道，栈中的任意一个节点的右儿子还没有被遍历过，说明后者一定是前者左儿子的子树中的节点，那么后者就先于前者出现在中序遍历中。

因此我们可以把 `index` 不断向右移动，并与栈顶节点进行比较。如果 `index` 对应的元素恰好等于栈顶节点，那么说明我们在中序遍历中找到了栈顶节点，所以将 `index` 增加 1 并弹出栈顶节点，直到 `index` 对应的元素不等于栈顶节点。按照这样的过程，我们弹出的最后一个节点 `x` 就是 10 是 `x` 的右儿子。

回到我们的例子，我们会依次从栈顶弹出 4，5 和 8，并且将 `index` 向右移动了三次。我们将 10 作为最后弹出的节点 8 的右儿子，并将 10 入栈。

- `stack = [3, 9, 10]`
- `index -> inorder[3] = 10`

- 我们遍历 20。同理，`index` 恰好指向当前栈顶节点 10，那么我们会依次从栈顶弹出 10，9 和 3，并且将 `index` 向右移动了三次。我们将 20 作为最后弹出的节点 3 的右儿子，并将 20 入栈。

- `stack = [20]`
- `index -> inorder[6] = 15`

- 我们遍历 15，将 15 作为栈顶节点 20 的左儿子，并将 15 入栈。

- `stack = [20, 15]`
- `index -> inorder[6] = 15`

- 我们遍历 7。`index` 恰好指向当前栈顶节点 15，那么我们会依次从栈顶弹出 15 和 20，并且将 `index` 向右移动了两次。我们将 7 作为最后弹出的节点 20 的右儿子，并将 7 入栈。

- `stack = [7]`
- `index -> inorder[8] = 7`

此时遍历结束，我们就构造出了正确的二叉树。

算法

我们归纳出上述例子中的算法流程：

- 我们用一个栈和一个指针辅助进行二叉树的构造。初始时栈中存放了根节点（前序遍历的第一个节点），指针指向中序遍历的第一个节点；
- 我们依次枚举前序遍历中除了第一个节点以外的每个节点。如果 `index` 恰好指向栈顶节点，那么我们不断地弹出栈顶节点并向右移动 `index`，并将当前节点作为最后一个弹出的节点的右儿子；如果 `index` 和栈顶节点不同，我们将当前节点作为栈顶节点的左儿子；

- 无论是哪一种情况，我们最后都将当前的节点入栈。

最后得到的二叉树即为答案。

```
C++ | Java | Python3 | Golang

class Solution {
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        if (preorder == null || preorder.length == 0) {
            return null;
        }
        TreeNode root = new TreeNode(preorder[0]);
        Deque<TreeNode> stack = new LinkedList<TreeNode>();
        stack.push(root);
        int inorderIndex = 0;
        for (int i = 1; i < preorder.length; i++) {
            int preorderVal = preorder[i];
            TreeNode node = stack.peek();
            if (node.val != inorder[inorderIndex]) {
                node.left = new TreeNode(preorderVal);
                stack.push(node.left);
            } else {
                while (!stack.isEmpty() && stack.peek().val == inorder[inorderIndex]) {
                    node = stack.pop();
                    inorderIndex++;
                }
                node.right = new TreeNode(preorderVal);
                stack.push(node.right);
            }
        }
        return root;
    }
}
```

复杂度分析

- 时间复杂度： $O(n)$ ，其中 n 是树中的节点个数。
- 空间复杂度： $O(n)$ ，除去返回的答案需要的 $O(n)$ 空间之外，我们还需要使用 $O(h)$ （其中 h 是树的高度）的空间存储栈。这里 $h < n$ ，所以（在最坏情况下）总空间复杂度为 $O(n)$ 。

239. 滑动窗口最大值 Hard

给你一个整数数组 `nums`，有一个大小为 `k` 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 `k` 个数字。滑动窗口每次只向右移动一位。

返回滑动窗口中的最大值。

示例 1：

输入: nums = [1,3,-1,-3,5,3,6,7], k = 3

输出: [3,3,5,5,6,7]

解释:

滑动窗口的位置	最大值
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

示例 2:

输入: nums = [1], k = 1

输出: [1]

示例 3:

输入: nums = [1,-1], k = 1

输出: [1,-1]

示例 4:

输入: nums = [9,11], k = 2

输出: [11]

示例 5:

输入: nums = [4,-2], k = 2

输出: [4]

提示:

- `1 <= nums.length <= 105`
- `-104 <= nums[i] <= 104`
- `1 <= k <= nums.length`

```
class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        int len = nums.length;
        int[] res = new int[len - k + 1];
        PriorityQueue<int[]> pq = new PriorityQueue<>(new Comparator<int[]>() {
            @Override
            public int compare(int[] o1, int[] o2) {
```

```

        if (o1[0] == o2[0])
            return o2[1] - o1[1];
        return o2[0] - o1[0];
    }
});

for (int i = 0; i < nums.length; i++) {
    while (!pq.isEmpty() && i - pq.peek()[1] >= k)
        pq.poll();
    pq.offer(new int[]{nums[i], i});
    if (i >= k - 1)
        res[i - k + 1] = pq.peek()[0];
}
return res;
}
}

```

上面优先队列时间复杂度为 $O(n \log n)$ ，空间复杂度为 $O(n)$ 。下面采用单调队列的方法，时间复杂度为 $O(n)$ ，空间复杂度为 $O(k)$ 。

```

class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        int len = nums.length;
        int[] res = new int[len - k + 1];
        Deque<Integer> deque = new LinkedList<>();
        for (int i = 0; i < nums.length; i++) {
            while (!deque.isEmpty() && nums[i] >= nums[deque.peekLast()])
                deque.pollLast();
            deque.offerLast(i);
            if (i - deque.peekFirst() >= k)
                deque.pollFirst();
            if (i + 1 - k >= 0)
                res[i + 1 - k] = nums[deque.peekFirst()];
        }
        return res;
    }
}

```

155. 最小栈 Easy

设计一个支持 `push`，`pop`，`top` 操作，并能在常数时间内检索到最小元素的栈。

- `push(x)` —— 将元素 x 推入栈中。
- `pop()` —— 删除栈顶的元素。
- `top()` —— 获取栈顶元素。
- `getMin()` —— 检索栈中的最小元素。

示例:

输入:

```
["MinStack","push","push","push","getMin","pop","top","getMin"]
[],[-2],[0],[-3],[],[],[],[]]
```

输出:

```
[null,null,null,null,-3,null,0,-2]
```

解释:

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();    --> 返回 -3.
minStack.pop();
minStack.top();      --> 返回 0.
minStack.getMin();    --> 返回 -2.
```

提示:

- `pop`、`top` 和 `getMin` 操作总是在 **非空栈** 上调用。

```
class MinStack {
    private Stack<value> stack;
    private int min;

    class value {
        int val;
        int min;

        public value(int val, int min) {
            this.val = val;
            this.min = min;
        }

        public value() {
        }
    }

    public MinStack() {
        stack = new Stack<>();
    }

    public void push(int val) {
        if (stack.isEmpty())
            stack.push(new value(val, val));
        else
```

```

        stack.push(new Value(val, Math.min(val, stack.peek().min)));
    }

    public void pop() {
        stack.pop();
    }

    public int top() {
        return stack.peek().val;
    }

    public int getMin() {
        return stack.peek().min;
    }
}

```

76. 最小覆盖子串 Hard

给你一个字符串 s 、一个字符串 t 。返回 s 中涵盖 t 所有字符的最小子串。如果 s 中不存在涵盖 t 所有字符的子串，则返回空字符串 "") 。

注意：

- 对于 t 中重复字符，我们寻找的子字符串中该字符数量必须不少于 t 中该字符数量。
- 如果 s 中存在这样的子串，我们保证它是唯一的答案。

示例 1：

输入: $s = \text{"ADOBECODEBANC"}$, $t = \text{"ABC"}$
 输出: "BANC"

示例 2：

输入: $s = \text{"a"}$, $t = \text{"a"}$
 输出: "a"

示例 3：

输入: $s = \text{"a"}$, $t = \text{"aa"}$
 输出: "")
 解释: t 中两个字符 ' a ' 均应包含在 s 的子串中，
 因此没有符合条件的子字符串，返回空字符串。

提示：

- $1 \leq s.length, t.length \leq 105$
- s 和 t 由英文字母组成

进阶：你能设计一个在 $O(n)$ 时间内解决此问题的算法吗？

```
class Solution {  
    public String minWindow(String s, String t) {  
        int[][] count = new int[200][2];  
        for (char ch : t.toCharArray()) {  
            count[ch][1]++;  
        }  
        char[] cs = s.toCharArray();  
        int len = s.length() + 1;  
        int L = 0, R = -1;  
        for (int left = 0, right = 0; right < cs.length; right++) {  
            count[cs[right]][0]++;  
            while (check(count) && left <= right) {  
                if (len > right - left + 1) {  
                    len = right - left + 1;  
                    L = left;  
                    R = right;  
                }  
                count[cs[left++]][0]--;  
            }  
        }  
        return s.substring(L, R + 1);  
    }  
  
    private boolean check(int[][] count) {  
        for (int i = 0; i < count.length; i++) {  
            if (count[i][1] == 0)  
                continue;  
            if (count[i][0] < count[i][1])  
                return false;  
        }  
        return true;  
    }  
}
```

改用 `HashMap` 写。

4. 寻找两个正序数组的中位数 Hard

给定两个大小分别为 m 和 n 的正序（从小到大）数组 nums1 和 nums2 。请你找出并返回这两个正序数组的中位数。

算法的时间复杂度应该为 $O(\log(m+n))$ 。

示例 1:

输入: $\text{nums1} = [1, 3]$, $\text{nums2} = [2]$

输出: 2.00000

解释: 合并数组 = [1, 2, 3]，中位数 2

示例 2:

输入: $\text{nums1} = [1, 2]$, $\text{nums2} = [3, 4]$

输出: 2.50000

解释: 合并数组 = [1, 2, 3, 4]，中位数 $(2 + 3) / 2 = 2.5$

示例 3:

输入: $\text{nums1} = [0, 0]$, $\text{nums2} = [0, 0]$

输出: 0.00000

示例 4:

输入: $\text{nums1} = []$, $\text{nums2} = [1]$

输出: 1.00000

示例 5:

输入: $\text{nums1} = [2]$, $\text{nums2} = []$

输出: 2.00000

提示:

- $\text{nums1.length} == m$
- $\text{nums2.length} == n$
- $0 \leq m \leq 1000$
- $0 \leq n \leq 1000$
- $1 \leq m + n \leq 2000$
- $-10^6 \leq \text{nums1}[i], \text{nums2}[i] \leq 10^6$

```
class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int length1 = nums1.length, length2 = nums2.length;
```

```

        int totalLength = length1 + length2;
        if (totalLength % 2 == 1) {
            int midIndex = totalLength / 2;
            double median = getKthElement(nums1, nums2, midIndex + 1);
            return median;
        } else {
            int midIndex1 = totalLength / 2 - 1, midIndex2 = totalLength / 2;
            double median = (getKthElement(nums1, nums2, midIndex1 + 1) +
                getKthElement(nums1, nums2, midIndex2 + 1)) / 2.0;
            return median;
        }
    }

    public int getKthElement(int[] nums1, int[] nums2, int k) {
        /* 主要思路：要找到第 k (k>1) 小的元素，那么就取 pivot1 = nums1[k/2-1] 和 pivot2 = nums2[k/2-1] 进行比较
         * 这里的 "/" 表示整除
         * nums1 中小于等于 pivot1 的元素有 nums1[0 .. k/2-2] 共计 k/2-1 个
         * nums2 中小于等于 pivot2 的元素有 nums2[0 .. k/2-2] 共计 k/2-1 个
         * 取 pivot = min(pivot1, pivot2)，两个数组中小于等于 pivot 的元素共计不会超过 (k/2-1)
         * + (k/2-1) <= k-2 个
         * 这样 pivot 本身最大也只能是第 k-1 小的元素
         * 如果 pivot = pivot1，那么 nums1[0 .. k/2-1] 都不可能是第 k 小的元素。把这些元素全部
         * "删除"，剩下的作为新的 nums1 数组
         * 如果 pivot = pivot2，那么 nums2[0 .. k/2-1] 都不可能是第 k 小的元素。把这些元素全部
         * "删除"，剩下的作为新的 nums2 数组
         * 由于我们 "删除" 了一些元素（这些元素都比第 k 小的元素要小），因此需要修改 k 的值，减去删
         * 除的数的个数
        */
        int length1 = nums1.length, length2 = nums2.length;
        int index1 = 0, index2 = 0;

        while (true) {
            // 边界情况
            if (index1 == length1)
                return nums2[index2 + k - 1];
            if (index2 == length2)
                return nums1[index1 + k - 1];
            if (k == 1)
                return Math.min(nums1[index1], nums2[index2]);

            // 正常情况
            int half = k / 2;
            int newIndex1 = Math.min(index1 + half, length1) - 1;
            int newIndex2 = Math.min(index2 + half, length2) - 1;
            int pivot1 = nums1[newIndex1], pivot2 = nums2[newIndex2];
            if (pivot1 <= pivot2) {
                k -= (newIndex1 - index1 + 1);

```

```

        index1 = newIndex1 + 1;
    } else {
        k -= (newIndex2 - index2 + 1);
        index2 = newIndex2 + 1;
    }
}
}
}

```

直接背吧。这道题直接使用暴力法，把两个数组合到一起，经过测试，在执行用时和内存消耗上都是完全可以接受的。

72. 编辑距离 Hard

给你两个单词 `word1` 和 `word2`，请你计算出将 `word1` 转换成 `word2` 所使用的最少操作数。

你可以对一个单词进行如下三种操作：

- 插入一个字符
- 删除一个字符
- 替换一个字符

示例 1：

```

输入: word1 = "horse", word2 = "ros"
输出: 3
解释:
horse -> rorse (将 'h' 替换为 'r')
rorse -> rose (删除 'r')
rose -> ros (删除 'e')

```

示例 2：

```

输入: word1 = "intention", word2 = "execution"
输出: 5
解释:
intention -> inention (删除 't')
inention -> enention (将 'i' 替换为 'e')
enention -> exention (将 'n' 替换为 'x')
exention -> exection (将 'n' 替换为 'c')
exection -> execution (插入 'u')

```

提示：

- $0 \leq \text{word1.length}, \text{word2.length} \leq 500$
- `word1` 和 `word2` 由小写英文字母组成

```

class Solution {
    public int minDistance(String word1, String word2) {
        char[] cs1 = word1.toCharArray();
        char[] cs2 = word2.toCharArray();
        int insertCost = 1;
        int deleteCost = 1;
        int replaceCost = 1;
        int[][] dp = new int[cs1.length + 1][cs2.length + 1];
        for (int i = 0; i < dp.length; i++) {
            for (int j = 0; j < dp[i].length; j++) {
                if (i == 0)
                    dp[i][j] = j * insertCost;
                if (j == 0)
                    dp[i][j] = i * deleteCost;
            }
        }
        for (int i = 1; i < dp.length; i++) {
            for (int j = 1; j < dp[i].length; j++) {
                if (cs1[i - 1] == cs2[j - 1])
                    dp[i][j] = dp[i - 1][j - 1];
                else
                    dp[i][j] = Math.min(dp[i - 1][j - 1] + replaceCost, Math.min(dp[i - 1][j] + deleteCost, dp[i][j - 1] + insertCost));
            }
        }
        return dp[cs1.length][cs2.length];
    }
}

```

定义 $dp[i][j]$ 的含义为：word1的前*i*个字符和word2的前*j*个字符的编辑距离。意思就是word1的前*i*个字符，变成word2的前*j*个字符，最少需要这么多步。二维的dp数组可以优化到一维，暂不收录。

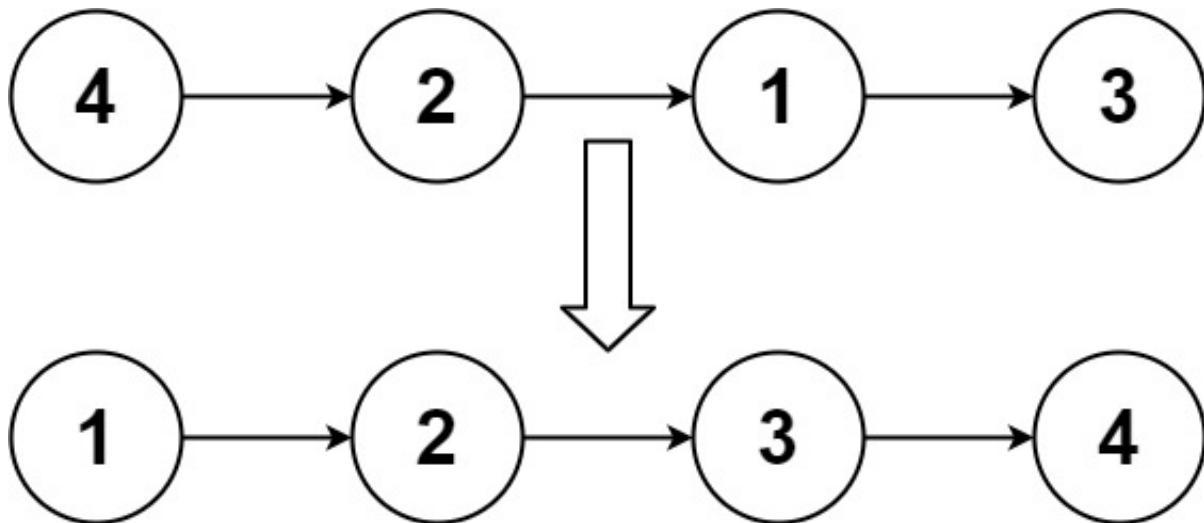
148. 排序链表 Medium

给你链表的头结点 `head`，请将其按 升序 排列并返回 排序后的链表 。

进阶：

- 你可以在 $O(n \log n)$ 时间复杂度和常数级空间复杂度下，对链表进行排序吗？

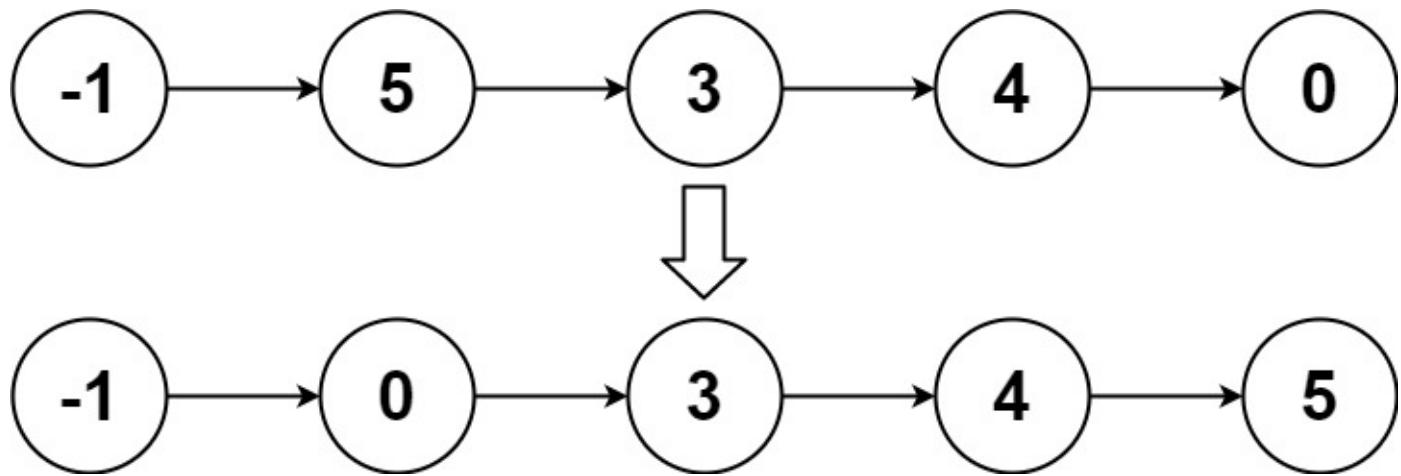
示例 1：



输入: head = [4,2,1,3]

输出: [1,2,3,4]

示例 2:



输入: head = [-1,5,3,4,0]

输出: [-1,0,3,4,5]

示例 3:

输入: head = []

输出: []

提示:

- 链表中节点的数目在范围 `[0, 5 * 104]` 内
- `-105 <= Node.val <= 105`

```

class Solution {
    public ListNode sortList(ListNode head) {
        if (head == null || head.next == null)

```

```

        return head;
    ListNode fast = head, slow = head, pre = head;
    while (fast != null && fast.next != null) {
        pre = slow;
        slow = slow.next;
        fast = fast.next.next;
    }
    pre.next = null;
    ListNode left = sortList(head);
    ListNode right = sortList(slow);
    return merge(left, right);
}

private ListNode merge(ListNode node1, ListNode node2) {
    ListNode dummyHead = new ListNode();
    ListNode cursor = dummyHead;
    while (node1 != null && node2 != null) {
        if (node1.val < node2.val) {
            cursor.next = node1;
            node1 = node1.next;
        } else {
            cursor.next = node2;
            node2 = node2.next;
        }
        cursor = cursor.next;
    }
    cursor.next = node1 == null ? node2 : node1;
    return dummyHead.next;
}
}

```

时间复杂度: $O(n \log n)$, 其中 n 是链表的长度。

空间复杂度: $O(\log n)$, 其中 n 是链表的长度。空间复杂度主要取决于递归调用的栈空间

该解法中有一个点不懂, 如果将分的部分写成下面样子会报错 `StackOverflowError`。一个未解之谜。

```

ListNode secondHalf = slow.next;
slow.next = null;
ListNode leftHalf = sort(head);
ListNode rightHalf = sort(secondHalf);

```

在这道题的一个评论看到了解答。

简单描述分治思想:

把一个复杂的问题分成两个或更多的相同或相似的子问题, 直到最后子问题可以简单的直接求解, 原问题的解即子问题的解的合并。

上面代码中最小的子问题（可以简单的直接求解的问题）为链表中包含元素的数量 ≤ 1 ，即 `(if(head==null || head.next==null) return head;)` 设为条件 a。

分析处理链表长度为 2 的情况，下面分别按照 `fast = head.next` 与 `fast = head` 进行，假设链表中包含两个元素 1, 2，不满足条件 a 因而无法进行直接求解。

1、`fast = head` 时，`while` 循环执行一次，`slow` 指向 2，`head2 = slow.next = null`，将链表分为 1, 2 与 `null` 两个部分，未缩小问题规模，进而产生死循环

2、`fast = head.next` 时 `fast.next != null` 不满足，`while` 循环无法执行。将链表分为 1 和 2 两个部分，缩小了问题规模。

如果你特别任性，一定要写 `fast = head` 也是可以的，只需要调整对于最小子问题的定义即可，将 `head` 的长度 = 2 时也作为最小子问题的一种进行处理：

```
public ListNode sortList(ListNode head) {
    if (head == null || head.next == null) return head;
    if (head.next != null && head.next.next == null) {
        // 链表长度为 2 时
        ListNode next = head.next;
        head.next = null;
        if (next.val < head.val) {
            next.next = head;
            return next;
        } else {
            head.next = next;
            return head;
        }
    }
    ListNode slow = head;
    ListNode fast = head.next;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }
    ListNode head2 = slow.next;
    slow.next = null;
    return merge(sortList(head), sortList(head2));
}
```

这道题有至少三种解法：

- 1、自顶向下的归并排序
- 2、自底向上的归并排序
- 3、快速排序版本

```
class Solution {
    public ListNode sortList(ListNode head) {
        if (head == null || head.next == null)
            return head;
        int len = getLength(head);
```

```

ListNode dummyHead = new ListNode(0, head);

for (int mergeSize = 1; mergeSize <= len; mergeSize = 2 * mergeSize) {
    ListNode pre = dummyHead;
    ListNode cur = dummyHead.next;
    while (cur != null) {
        ListNode node1 = cur;
        for (int i = 1; i < mergeSize && cur != null && cur.next != null; i++)
{
            cur = cur.next;
}
        ListNode node2 = cur.next;
        cur.next = null;
        cur = node2;

        for (int i = 1; i < mergeSize && cur != null && cur.next != null; i++)
{
            cur = cur.next;
}
        ListNode next = null;
        if (cur != null) {
            next = cur.next;
            cur.next = null;
}
        pre.next = merge(node1, node2);
        while (pre.next != null)
            pre = pre.next;
        cur = next;
}
    }
    return dummyHead.next;
}

private ListNode merge(ListNode node1, ListNode node2) {
    ListNode dummyHead = new ListNode();
    ListNode cursor = dummyHead;
    while (node1 != null && node2 != null) {
        if (node1.val < node2.val) {
            cursor.next = node1;
            node1 = node1.next;
}
        else {
            cursor.next = node2;
            node2 = node2.next;
}
        cursor = cursor.next;
}
    cursor.next = node1 == null ? node2 : node1;
    return dummyHead.next;
}

```

```

    }

    private int getLength(ListNode head) {
        return head == null ? 0 : getLength(head.next) + 1;
    }
}

```

```

class Solution {
    public ListNode sortList(ListNode head) {
        if (head == null || head.next == null)
            return head;
        ListNode dummyHead = new ListNode(0, head);
        quickSort(dummyHead, null);
        return dummyHead.next;
    }

    private void quickSort(ListNode dummyHead, ListNode tail) {
        if (dummyHead == tail || dummyHead.next == tail)
            return;
        ListNode smallerEleList = new ListNode();
        ListNode cursor = smallerEleList;
        ListNode pivot = dummyHead.next;
        ListNode comparedNode = pivot;
        while (comparedNode.next != tail) {
            if (pivot.val > comparedNode.next.val) {
                cursor.next = comparedNode.next;
                cursor = cursor.next;
                comparedNode.next = comparedNode.next.next;
            } else
                comparedNode = comparedNode.next;
        }
        // dummyHead.next = smallerEleList.next;
        // 下面两行代码不能交换位置否则可能会报空指针异常。
        // 因为可能会有所有比较元素都大于pivot的情况,
        // 这时先将dummyHead的指向改到新链的节点上会导致dummyHead后面为空
        cursor.next = pivot;
        dummyHead.next = smallerEleList.next;

        quickSort(dummyHead, pivot);
        quickSort(pivot, tail);
    }
}

```

543. 二叉树的直径 Easy

给定一棵二叉树，你需要计算它的直径长度。一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过也可能不穿过根结点。

示例：

给定二叉树



返回 3, 它的长度是路径 [4,2,1,3] 或者 [5,2,1,3]。

注意：两结点之间的路径长度是以它们之间边的数目表示。

```
class Solution {
    public int diameterOfBinaryTree(TreeNode root) {
        if (root == null)
            return 0;
        int left = height(root.left);
        int right = height(root.right);
        return Math.max(Math.max(diameterOfBinaryTree(root.left),
diameterOfBinaryTree(root.right)), left + right);
    }

    private int height(TreeNode root) {
        if (root == null)
            return 0;
        return Math.max(height(root.left), height(root.right)) + 1;
    }
}
```

不知道上面的方法时间复杂度怎么计算。可以优化出时间复杂度为 $O(N)$ 的方法，其中 N 为二叉树的节点数，即遍历一棵二叉树的时间复杂度，每个结点只被访问一次。

```
class Solution {
    private int diameter = 0;

    public int diameterOfBinaryTree(TreeNode root) {
        dfs(root);
        return diameter;
    }

    private int dfs(TreeNode root) {
```

```

    if (root == null)
        return 0;
    int left = dfs(root.left);
    int right = dfs(root.right);
    diameter = Math.max(diameter, left + right);
    return 1 + Math.max(left, right);
}
}

```

复杂度分析：

时间复杂度：O(N)，其中 N 为二叉树的节点数，即遍历一棵二叉树的时间复杂度，每个结点只被访问一次。

空间复杂度：O(Height)，其中 Height 为二叉树的高度。由于递归函数在递归过程中需要为每一层递归函数分配栈空间，所以这里需要额外的空间且该空间取决于递归的深度，而递归的深度显然为二叉树的高度，并且每次递归调用的函数里又只用了常数个变量，所以所需空间复杂度为 O(Height)。

169. 多数元素 Easy

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 $\lfloor n/2 \rfloor$ 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

```

进阶：

- 尝试设计时间复杂度为 $O(n)$ 、空间复杂度为 $O(1)$ 的算法解决此问题。

```

class Solution {
    public int majorityElement(int[] nums) {
        Arrays.sort(nums);
        return nums[nums.length / 2];
    }
}

```

```
//Boyer-Moore 投票算法
class Solution {
    public int majorityElement(int[] nums) {
        int majority = nums[0];
        int count = 0;
        for (int num : nums) {
            if (count == 0)
                majority = num;
            count += majority == num ? 1 : -1;
        }
        return majority;
    }
}
```

98. 验证二叉搜索树 Medium

给定一个二叉树，判断其是否是一个有效的二叉搜索树。

假设一个二叉搜索树具有如下特征：

- 节点的左子树只包含**小于**当前节点的数。
- 节点的右子树只包含**大于**当前节点的数。
- 所有左子树和右子树自身必须也是二叉搜索树。

示例 1:

```
输入：
2
/
1   3
输出: true
```

示例 2:

```
输入：
5
/
1   4
  / \
 3   6
输出: false
解释: 输入为: [5,1,4,null,null,3,6]。
根节点的值为 5，但是其右子节点值为 4。
```

```

class Solution {
    public boolean isValidBST(TreeNode root) {
        return isValidBST(root, Long.MIN_VALUE, Long.MAX_VALUE);
    }

    private boolean isValidBST(TreeNode root, long minValue, long maxValue) {
        if (root == null)
            return true;
        if (root.val <= minValue || root.val >= maxValue)
            return false;
        return isValidBST(root.left, minValue, root.val) && isValidBST(root.right,
root.val, maxValue);
    }
}

```

显然，一个二叉搜索树中序遍历得到的序列是一个严格递增的序列。

```

class Solution {
    private long pre = Long.MIN_VALUE;

    public boolean isValidBST(TreeNode root) {
        if (root == null)
            return true;
        if (!isValidBST(root.left))
            return false;
        if (pre >= root.val)
            return false;
        pre = root.val;
        return isValidBST(root.right);
    }
}

```

```

class Solution {
    public boolean isValidBST(TreeNode root) {
        Stack<TreeNode> stack = new Stack<>();
        TreeNode cur = root;
        long pre = Long.MIN_VALUE;
        while (cur != null || !stack.isEmpty()) {
            while (cur != null) {
                stack.push(cur);
                cur = cur.left;
            }
            TreeNode pop = stack.pop();
            if (pop.val <= pre)
                return false;
            pre = pop.val;
            cur = pop.right;
        }
    }
}

```

```
    }
    return true;
}
}
```

101. 对称二叉树 Easy

给定一个二叉树，检查它是否是镜像对称的。

例如，二叉树 `[1,2,2,3,4,4,3]` 是对称的。

```
1
/
2   2
/ \ / \
3 4 4 3
```

但是下面这个 `[1,2,2,null,3,null,3]` 则不是镜像对称的：

```
1
/
2   2
\   \
3   3
```

进阶：

你可以运用递归和迭代两种方法解决这个问题吗？

```
class Solution {
    public boolean isSymmetric(TreeNode root) {
        if (root == null)
            return true;
        return isSymmetric(root.left, root.right);
    }

    private boolean isSymmetric(TreeNode left, TreeNode right) {
        if (left == null && right == null)
            return true;
        else if (left == null || right == null)
            return false;
        else if (left.val != right.val)
            return false;
    }
}
```

```

        return isSymmetric(left.left, right.right) && isSymmetric(left.right,
right.left);
    }
}

```

```

class Solution {

    public boolean isSymmetric(TreeNode root) {
        if (root == null)
            return true;
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root.left);
        queue.offer(root.right);
        while (!queue.isEmpty()) {
            TreeNode left = queue.poll();
            TreeNode right = queue.poll();
            if (left == null && right == null)
                continue;
            if (left == null || right == null)
                return false;
            if (left.val != right.val)
                return false;
            queue.offer(left.left);
            queue.offer(right.right);
            queue.offer(left.right);
            queue.offer(right.left);
        }
        return true;
    }
}

```

226. 翻转二叉树 Easy

翻转一棵二叉树。

示例：

输入：

```

4
/
2   \
/ \   / \
1   3 6   9

```

输出：

```
    4
   /   \
  7     2
 / \   / \
9   6  3   1
```

备注：

这个问题是受到 [Max Howell](#) 的 [原问题](#) 启发的：

谷歌：我们90%的工程师使用您编写的软件(Homebrew)，但是您却无法在面试时在白板上写出翻转二叉树这道题，这太糟糕了。

Google: 90% of our engineers use the software you wrote (Homebrew), but you can't invert a binary tree on a whiteboard so fuck off.

```
class Solution {
    public TreeNode invertTree(TreeNode root) {
        if (root == null)
            return root;
        swapChildren(root);
        invertTree(root.left);
        invertTree(root.right);
        return root;
    }

    private void swapChildren(TreeNode root) {
        TreeNode temp = root.left;
        root.left = root.right;
        root.right = temp;
    }
}
```

```
class Solution {
    public TreeNode invertTree(TreeNode root) {
        if (root == null)
            return root;
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        while (!queue.isEmpty()) {
            TreeNode poll = queue.poll();
            if (poll == null)
                continue;
            swapChildren(poll);
            queue.offer(poll.left);
            queue.offer(poll.right);
        }
    }
}
```

```

    }
    return root;
}

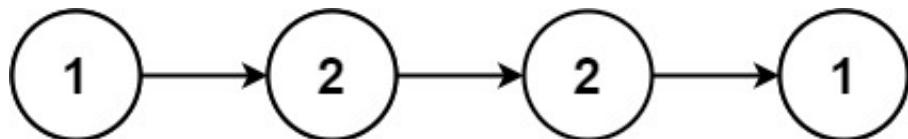
private void swapChildren(TreeNode root) {
    if (root == null)
        return;
    TreeNode temp = root.left;
    root.left = root.right;
    root.right = temp;
}
}

```

234. 回文链表 Easy

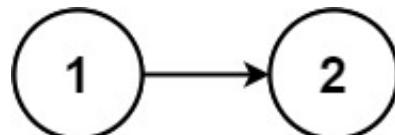
给你一个单链表的头节点 `head`，请你判断该链表是否为回文链表。如果是，返回 `true`；否则，返回 `false`。

示例 1：



输入： `head = [1,2,2,1]`
输出： `true`

示例 2：



输入： `head = [1,2]`
输出： `false`

提示：

- 链表中节点数目在范围 `[1, 105]` 内
- `0 <= Node.val <= 9`

进阶：你能否用 $O(n)$ 时间复杂度和 $O(1)$ 空间复杂度解决此题？

```

class Solution {
    public boolean isPalindrome(ListNode head) {

```

```

    if (head == null || head.next == null)
        return true;

    ListNode slow = head, fast = head;
    boolean res = false;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }
    ListNode newHead = reverseList(slow);
    res = isPalindrome(head, newHead);
    reverseList(newHead);
    return res;
}

private boolean isPalindrome(ListNode head1, ListNode head2) {
    ListNode h1 = head1, h2 = head2;
    while (h1 != null && h2 != null) {
        if (h1.val != h2.val)
            return false;
        h1 = h1.next;
        h2 = h2.next;
    }
    return true;
}

private ListNode reverseList(ListNode head) {
    ListNode pre = null;
    ListNode node = head;
    ListNode next = head;
    while (node != null) {
        next = node.next;
        node.next = pre;
        pre = node;
        node = next;
    }
    return pre;
}

}

```

可以在使用快慢指针找出中点时，反转前半部分，这样时间应该会更快一点。

```

class Solution {
    public boolean isPalindrome(ListNode head) {
        if (head == null || head.next == null)
            return true;
        ListNode pre = null;
        ListNode slow = head, fast = head;
        while (fast != null && fast.next != null) {

```

```

        fast = fast.next.next;
        ListNode next = slow.next;
        slow.next = pre;
        pre = slow;
        slow = next;
    }
    if (fast == null)
        return isPalindrome(pre, slow);
    else
        return isPalindrome(pre, slow.next);
}

private boolean isPalindrome(ListNode pre, ListNode next) {
    while (pre != null) {
        if (pre.val != next.val)
            return false;
        pre = pre.next;
        next = next.next;
    }
    return true;
}
}

```

78. 子集 Medium

给你一个整数数组 `nums`，数组中的元素 **互不相同**。返回该数组所有可能的子集（幂集）。

解集 **不能** 包含重复的子集。你可以按 **任意顺序** 返回解集。

示例 1：

```

输入: nums = [1,2,3]
输出: [[], [1], [2], [1,2], [3], [1,3], [2,3], [1,2,3]]

```

示例 2：

```

输入: nums = [0]
输出: [[], [0]]

```

提示：

- `1 <= nums.length <= 10`
- `-10 <= nums[i] <= 10`
- `nums` 中的所有元素 **互不相同**

```

class Solution {
    private List<List<Integer>> res = new ArrayList<>();
    private List<Integer> item = new ArrayList<>();

    public List<List<Integer>> subsets(int[] nums) {
        backtracking(nums, 0);
        return res;
    }

    private void backtracking(int[] nums, int curDigit) {
        res.add(new ArrayList<>(item));
        for (int i = curDigit; i < nums.length; i++) {
            item.add(nums[i]);
            backtracking(nums, i + 1);
            item.remove(item.size() - 1);
        }
    }
}

```

90. 子集 II Medium

给你一个整数数组 `nums`，其中可能包含重复元素，请你返回该数组所有可能的子集（幂集）。

解集 **不能** 包含重复的子集。返回的解集中，子集可以按 **任意顺序** 排列。

示例 1：

```

输入: nums = [1,2,2]
输出: [[], [1], [1,2], [1,2,2], [2], [2,2]]

```

示例 2：

```

输入: nums = [0]
输出: [[], [0]]

```

提示：

- `1 <= nums.length <= 10`
- `-10 <= nums[i] <= 10`

```

class Solution {
    private List<List<Integer>> res;

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

```

```

        res = new ArrayList<>();
        Arrays.sort(nums);
        backtracking(nums, 0, nums.length - 1, new ArrayList<>(), new
boolean[nums.length]);
        return res;
    }

    private void backtracking(int[] nums, int curDigit, int totalDigit,
ArrayList<Integer> item, boolean[] visited) {
        res.add(new ArrayList<>(item));
        for (int i = curDigit; i <= totalDigit; i++) {
            if (visited[i] || i > 0 && !visited[i - 1] && nums[i] == nums[i - 1])
                continue;
            visited[i] = true;
            item.add(nums[i]);
            backtracking(nums, i + 1, totalDigit, item, visited);
            item.remove(item.size() - 1);
            visited[i] = false;
        }
    }
}

```

32. 最长有效括号 Hard

给你一个只包含 '(' 和 ')' 的字符串，找出最长有效（格式正确且连续）括号子串的长度。

示例 1：

输入： s = "(()"

输出： 2

解释： 最长有效括号子串是 "(())"

示例 2：

输入： s = ")()())"

输出： 4

解释： 最长有效括号子串是 "())()

示例 3：

输入： s = ""

输出： 0

提示：

- $0 \leq s.length \leq 3 * 10^4$
- $s[i]$ 为 '(' 或 ')'

```

class Solution {
    public int longestValidParentheses(String s) {
        Stack<Integer> stack = new Stack<>();
        stack.push(-1);
        int len = 0;
        char[] cs = s.toCharArray();
        for (int i = 0; i < cs.length; i++) {
            if (cs[i] == '(')
                stack.push(i);
            else {
                stack.pop();
                if (stack.isEmpty())
                    stack.push(i);
                else
                    len = Math.max(len, i - stack.peek());
            }
        }
        return len;
    }
}

```

上面使用栈，具体做法是我们始终保持栈底元素为当前已经遍历过的元素中「最后一个没有被匹配的右括号的下标」。

下面使用动态规划。我们定义 $dp[i]$ 表示以下标 i 字符结尾的最长有效括号的长度。我们将 dp 数组全部初始化为 0。显然有效的子串一定以 ')' 结尾，因此我们可以知道以 ')' 结尾的子串对应的 dp 值必定为 0，我们只需要求解 ')' 在 dp 数组中对应位置的值。我们从前往后遍历字符串求解 dp 值，我们每两个字符检查一次。

```

class Solution {
    public int longestValidParentheses(String s) {
        int[] dp = new int[s.length()];
        char[] cs = s.toCharArray();
        int len = 0;
        for (int i = 1; i < cs.length; i++) {
            if (cs[i] == '(')
                continue;
            else if (cs[i - 1] == '(')
                dp[i] = (i - 2 >= 0 ? dp[i - 2] : 0) + 2;
            else if (i - dp[i - 1] > 0 && cs[i - dp[i - 1] - 1] == '(')
                dp[i] = 2 + dp[i - 1] + ((i - dp[i - 1] - 2) >= 0 ? dp[i - dp[i - 1] - 2] : 0);
            len = Math.max(len, dp[i]);
        }
    }
}

```

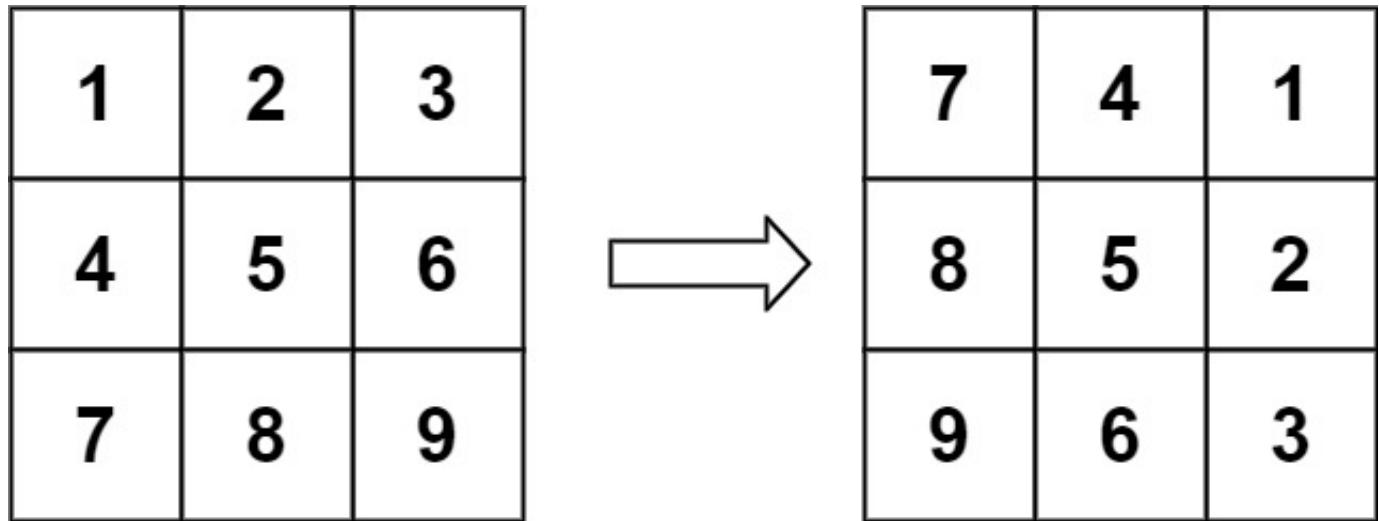
```
    return len;
}
}
```

48. 旋转图像 Medium

给定一个 $n \times n$ 的二维矩阵 `matrix` 表示一个图像。请你将图像顺时针旋转 90 度。

你必须在 [原地](#) 旋转图像，这意味着你需要直接修改输入的二维矩阵。请不要 使用另一个矩阵来旋转图像。

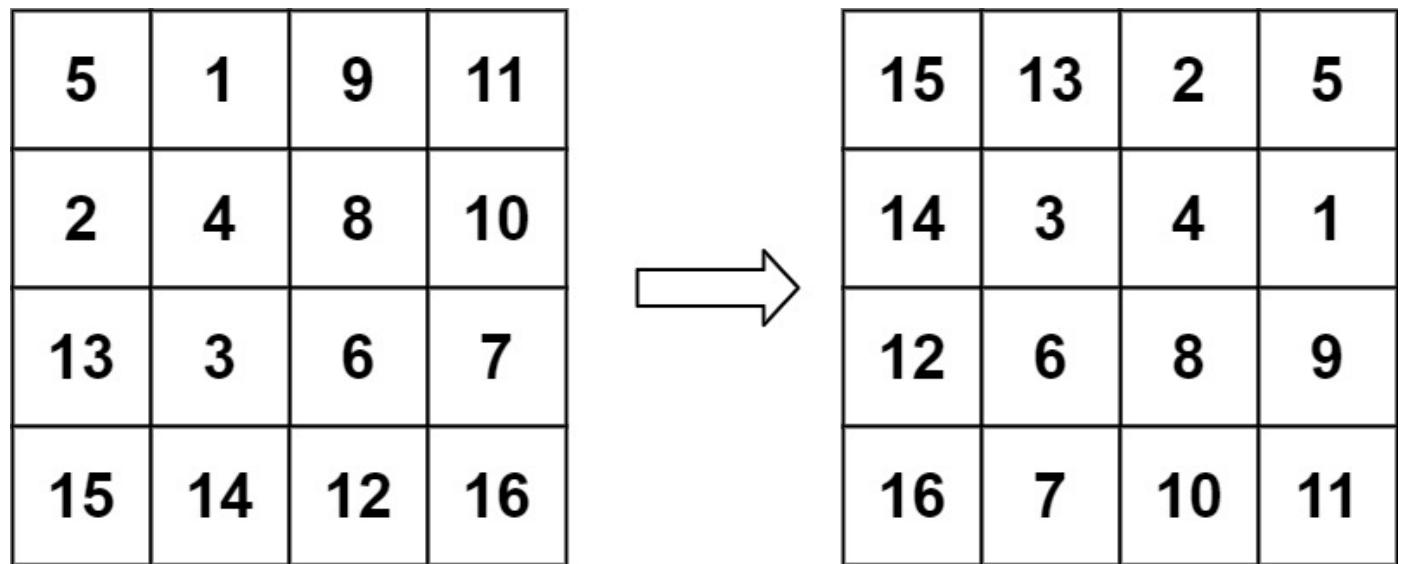
示例 1：



输入： `matrix = [[1,2,3],[4,5,6],[7,8,9]]`

输出： `[[7,4,1],[8,5,2],[9,6,3]]`

示例 2：



```
输入: matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]
输出: [[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]
```

示例 3:

```
输入: matrix = [[1]]
输出: [[1]]
```

示例 4:

```
输入: matrix = [[1,2],[3,4]]
输出: [[3,1],[4,2]]
```

提示:

- `matrix.length == n`
- `matrix[i].length == n`
- `1 <= n <= 20`
- `-1000 <= matrix[i][j] <= 1000`

```
class Solution {
    public void rotate(int[][] matrix) {
        for (int i = 0; i < matrix.length - 1; i++) {
            for (int j = i + 1; j < matrix[i].length; j++) {
                swap(matrix, i, j);
            }
        }
        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[i].length / 2; j++) {
                int temp = matrix[i][j];
                matrix[i][j] = matrix[i][matrix[i].length - j - 1];
                matrix[i][matrix[i].length - j - 1] = temp;
            }
        }
    }

    private void swap(int[][] matrix, int i, int j) {
        int temp = matrix[i][j];
        matrix[i][j] = matrix[j][i];
        matrix[j][i] = temp;
    }
}
```

先转置transpose, 再左右翻转。这题觉得好无聊。当然还有另一种相对容易想到的方法。

```
class Solution {
```

```

public void rotate(int[][] matrix) {
    int n = matrix.length;
    for (int i = 0; i < n / 2; i++) {
        for (int j = 0; j < (n + 1) / 2; j++) {
            int temp = matrix[i][j];
            matrix[i][j] = matrix[n - 1 - j][i];
            matrix[n - 1 - j][i] = matrix[n - 1 - i][n - 1 - j];
            matrix[n - 1 - i][n - 1 - j] = matrix[j][n - 1 - i];
            matrix[j][n - 1 - i] = temp;
        }
    }
}

```

322. 零钱兑换 Medium

给你一个整数数组 `coins`，表示不同面额的硬币；以及一个整数 `amount`，表示总金额。

计算并返回可以凑成总金额所需的 **最少的硬币个数**。如果没有任何一种硬币组合能组成总金额，返回 `-1`。

你可以认为每种硬币的数量是无限的。

示例 1:

```

输入: coins = [1, 2, 5], amount = 11
输出: 3
解释: 11 = 5 + 5 + 1

```

示例 2:

```

输入: coins = [2], amount = 3
输出: -1

```

示例 3:

```

输入: coins = [1], amount = 0
输出: 0

```

示例 4:

```

输入: coins = [1], amount = 1
输出: 1

```

示例 5:

```
输入: coins = [1], amount = 2
输出: 2
```

提示:

- $1 \leq \text{coins.length} \leq 12$
- $1 \leq \text{coins}[i] \leq 231 - 1$
- $0 \leq \text{amount} \leq 104$

```
class Solution {
    private static final int INF = 100_000;

    public int coinChange(int[] coins, int amount) {
        int[][] dp = new int[coins.length + 1][amount + 1];
        for (int i = 0; i < dp.length; i++) {
            for (int j = 0; j < dp[i].length; j++) {
                if (j == 0)
                    dp[i][j] = 0;
                else
                    dp[i][j] = INF;
            }
        }
        for (int i = 1; i <= coins.length; i++) {
            for (int j = 1; j <= amount; j++) {
                dp[i][j] = dp[i - 1][j];
                for (int k = 0; j - k * coins[i - 1] >= 0; k++) {
                    dp[i][j] = Math.min(dp[i][j], dp[i][j - k * coins[i - 1]] + k);
                }
            }
        }
        return dp[coins.length][amount] >= INF ? -1 : dp[coins.length][amount];
    }
}
```

二维的dp，上面↑有一个三重循环的dp，那个是基本思想，需要好好掌握，再上升到这个两重循环的二维，再上升到一维。

```
class Solution {
    public int coinChange(int[] coins, int amount) {
        int[] dp = new int[amount + 1];
        Arrays.fill(dp, Integer.MAX_VALUE >> 1);
        dp[0] = 0;
        for (int coin : coins) {
            for (int i = 0; i < amount + 1; i++) {
                if (i >= coin)
                    dp[i] = Math.min(dp[i - coin] + 1, dp[i]);
            }
        }
    }
}
```

```

    }
    return dp[amount] >= Integer.MAX_VALUE >> 1 ? -1 : dp[amount];
}
}

```

完全背包的遍历顺序可以变换。

```

class Solution {
    public int coinChange(int[] coins, int amount) {
        int[] dp = new int[amount + 1];
        Arrays.fill(dp, amount + 1);
        dp[0] = 0;
        for (int coin : coins) {
            for (int i = coin; i <= amount; i++) {
                dp[i] = Math.min(dp[i], dp[i - coin] + 1);
            }
        }
        return dp[amount] > amount ? -1 : dp[amount];
    }
}

```

```

class Solution {
    public int coinChange(int[] coins, int amount) {
        int[] dp = new int[amount + 1];
        Arrays.fill(dp, amount + 1);
        dp[0] = 0;
        for (int i = 1; i <= amount; i++) {
            for (int coin : coins) {
                if (i - coin >= 0)
                    dp[i] = Math.min(dp[i], dp[i - coin] + 1);
            }
        }
        return dp[amount] > amount ? -1 : dp[amount];
    }
}

```

518. 零钱兑换 II Medium

给你一个整数数组 `coins` 表示不同面额的硬币，另给一个整数 `amount` 表示总金额。

请你计算并返回可以凑成总金额的硬币组合数。如果任何硬币组合都无法凑出总金额，返回 `0`。

假设每一种面额的硬币有无限个。题目数据保证结果符合 32 位带符号整数。

示例 1：

```
输入: amount = 5, coins = [1, 2, 5]
```

```
输出: 4
```

```
解释: 有四种方式可以凑成总金额:
```

```
5=5
```

```
5=2+2+1
```

```
5=2+1+1+1
```

```
5=1+1+1+1+1
```

示例 2:

```
输入: amount = 3, coins = [2]
```

```
输出: 0
```

```
解释: 只用面额 2 的硬币不能凑成总金额 3。
```

示例 3:

```
输入: amount = 10, coins = [10]
```

```
输出: 1
```

提示:

- $1 \leq \text{coins.length} \leq 300$
- $1 \leq \text{coins}[i] \leq 5000$
- `coins` 中的所有值 互不相同
- $0 \leq \text{amount} \leq 5000$

```
class Solution {  
    public int change(int amount, int[] coins) {  
        return backtracking(coins, amount, 0);  
    }  
  
    private int backtracking(int[] coins, int amount, int index) {  
        if (amount <= 0)  
            return amount == 0 ? 1 : 0;  
        int count = 0;  
        for (int i = index; i < coins.length; i++) {  
            count += backtracking(coins, amount - coins[i], i);  
        }  
        return count;  
    }  
}
```

这个方法部分测试用例会超时。

```

class Solution {
    public int change(int amount, int[] coins) {
        int[] dp = new int[amount + 1];
        dp[0] = 1;
        for (int coin : coins) {
            for (int i = coin; i <= amount; i++) {
                dp[i] += dp[i - coin];
            }
        }
        return dp[amount];
    }
}

```

`dp[x]` 表示金额之和等于 `x` 的硬币组合数。

因为外层循环是遍历数组 `coins` 的值，内层循环是遍历不同的金额之和，在计算 `dp[i]` 的值时，可以确保金额之和等于 `i` 的硬币面额的顺序，由于顺序确定，因此不会重复计算不同的排列。

34. 在排序数组中查找元素的第一个和最后一个位置 Medium

给定一个按照升序排列的整数数组 `nums`，和一个目标值 `target`。找出给定目标值在数组中的开始位置和结束位置。

如果数组中不存在目标值 `target`，返回 `[-1, -1]`。

进阶：

- 你可以设计并实现时间复杂度为 $O(\log n)$ 的算法解决此问题吗？

示例 1：

```

输入: nums = [5,7,7,8,8,10], target = 8
输出: [3,4]

```

示例 2：

```

输入: nums = [5,7,7,8,8,10], target = 6
输出: [-1,-1]

```

示例 3：

```

输入: nums = [], target = 0
输出: [-1,-1]

```

提示：

- $0 \leq \text{nums.length} \leq 105$

- $-109 \leq \text{nums}[i] \leq 109$
- `nums` 是一个非递减数组
- $-109 \leq \text{target} \leq 109$

```

class Solution {
    public int[] searchRange(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        //这个等号可不可以去掉了，如果测试数组只有一个元素，去掉的话会报错。
        while (left <= right) {
            int mid = (left + right) / 2;
            if (nums[mid] < target)
                left = mid + 1;
            else if (nums[mid] > target)
                right = mid - 1;
            else {
                int leftBound = mid;
                int rightBound = mid;
                while (leftBound - 1 >= left && nums[leftBound - 1] == nums[mid])
                    leftBound--;
                while (rightBound + 1 <= right && nums[rightBound + 1] == nums[mid])
                    rightBound++;
                return new int[]{leftBound, rightBound};
            }
        }
        return new int[]{-1, -1};
    }
}

```

这个方法的时间复杂度可能退化到线性阶。

```

class Solution {
    public int[] searchRange(int[] nums, int target) {
        int leftBound = searchLeft(nums, target);
        int rightBound = searchRight(nums, target);
        if (leftBound > rightBound || leftBound < 0 || rightBound >= nums.length ||
            nums[leftBound] != target || nums[rightBound] != target)
            return new int[]{-1, -1};
        return new int[]{leftBound, rightBound};
    }

    private int searchLeft(int[] nums, int target) {
        int left = -1, right = nums.length;
        while (left + 1 != right) {
            int mid = (left + right) / 2;
            if (nums[mid] == target)
                right = mid;
            else if (nums[mid] < target)
                left = mid;
            else if (nums[mid] > target)
                right = mid;
        }
        return left;
    }
}

```

```

        else if (nums[mid] < target)
            left = mid;
        else if (nums[mid] > target)
            right = mid;
    }
    return right;
}

private int searchRight(int[] nums, int target) {
    int left = -1, right = nums.length;
    while (left + 1 != right) {
        int mid = (left + right) / 2;
        if (nums[mid] == target)
            left = mid;
        else if (nums[mid] < target)
            left = mid;
        else if (nums[mid] > target)
            right = mid;
    }
    return left;
}

}

```

二分真的讨厌。用这个模板改写了第一个解法。

```

class Solution {
    public int[] searchRange(int[] nums, int target) {
        int left = -1, right = nums.length;
        while (left + 1 != right) {
            int mid = (left + right) / 2;
            if (nums[mid] < target)
                left = mid;
            else if (nums[mid] > target)
                right = mid;
            else {
                int leftBound = mid;
                int rightBound = mid;
                while (leftBound - 1 >= Math.max(0, left) && nums[leftBound - 1] ==
nums[mid])
                    leftBound--;
                while (rightBound + 1 <= Math.min(nums.length - 1, right) &&
nums[rightBound + 1] == nums[mid])
                    rightBound++;
                return new int[]{leftBound, rightBound};
            }
        }
        return new int[]{-1, -1};
    }
}

```

```
}
```

一个巧妙思，让这道题的难度下降了一些。

```
class Solution {
    public int[] searchRange(int[] nums, int target) {
        if (nums.length == 0)
            return new int[]{-1, -1};
        int left = leftBound(nums, target);
        int right = leftBound(nums, target + 1) - 1;
        if (left >= nums.length || nums[left] != target)
            return new int[]{-1, -1};
        return new int[]{left, right};
    }

    private int leftBound(int[] nums, int target) {
        int left = 0, right = nums.length;
        while (left < right) {
            int mid = (left + right) >> 1;
            if (nums[mid] >= target)
                right = mid;
            else
                left = mid + 1;
        }
        return left;
    }
}
```

二分真的好烦啊，还是不会。

```
class Solution {
    public int[] searchRange(int[] nums, int target) {
        int[] res = new int[2];
        int begin = beginPosition(nums, target);
        int end = endPosition(nums, target);
        if (begin < 0 || begin >= nums.length || nums[begin] != target)
            begin = -1;
        if (end < 0 || end >= nums.length || nums[end] != target)
            end = -1;
        res[0] = begin;
        res[1] = end;
        return res;
    }

    private int beginPosition(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        while (left < right) {
            int mid = (left + right) >> 1;

```

```

        if (nums[mid] == target)
            right = mid;
        else if (nums[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return right;
}

private int endPosition(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left < right) {
        int mid = (left + right + 1) >> 1;
        if (nums[mid] == target)
            left = mid;
        else if (nums[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return left;
}
}

```

22. 括号生成 Medium

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。

有效括号组合需满足：左括号必须以正确的顺序闭合。

示例 1：

```

输入: n = 3
输出: [ "((()))" , "(()())" , "()(())" , "()((())" , "()()()" ]

```

示例 2：

```

输入: n = 1
输出: [ "( )" ]

```

提示：

- `1 <= n <= 8`

```

class Solution {
    public List<String> generateParenthesis(int n) {
        ArrayList<String> res = new ArrayList<String>();
        dfs(n, 0, 0, "", res);
        return res;
    }

    private void dfs(int n, int left, int right, String str, ArrayList<String> res) {
        if (left > n || right > n || left < right)
            return;
        if (left == right && left == n) {
            res.add(str);
            return;
        }
        dfs(n, left + 1, right, str + "(", res);
        dfs(n, left, right + 1, str + ")", res);
    }
}

```

```

class Solution {
    private List<String> res;

    public List<String> generateParenthesis(int n) {
        res = new ArrayList<>();
        dfs(0, 0, n, new StringBuffer());
        return res;
    }

    private void dfs(int leftParenthesis, int rightParenthesis, int n, StringBuffer sb)
    {
        if (leftParenthesis > n || leftParenthesis < rightParenthesis)
            return;
        if (leftParenthesis == rightParenthesis && leftParenthesis == n) {
            res.add(sb.toString());
            return;
        }
        dfs(leftParenthesis + 1, rightParenthesis, n, sb.append("("));
        sb.deleteCharAt(sb.length() - 1);
        dfs(leftParenthesis, rightParenthesis + 1, n, sb.append(")");
        sb.deleteCharAt(sb.length() - 1);
    }
}

```

此题还有bfs、dp两种做法，暂未收录。

64. 最小路径和 Medium

给定一个包含非负整数的 $m \times n$ 网格 `grid`，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

说明：每次只能向下或者向右移动一步。

示例 1：

1	3	1
1	5	1
4	2	1

输入：`grid = [[1,3,1],[1,5,1],[4,2,1]]`

输出：7

解释：因为路径 $1 \rightarrow 3 \rightarrow 1 \rightarrow 1 \rightarrow 1$ 的总和最小。

示例 2：

输入：`grid = [[1,2,3],[4,5,6]]`

输出：12

提示：

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 200`
- `0 <= grid[i][j] <= 100`

```
class Solution {
    private int minPath = Integer.MAX_VALUE;

    public int minPathSum(int[][] grid) {
        backtracking(grid, 0, 0, 0);
        return minPath;
    }

    private void backtracking(int[][] grid, int i, int j, int path) {
```

```

    if (i < 0 || i >= grid.length || j < 0 || j >= grid[i].length || path >
minPath)
        return;

    if (i == grid.length - 1 && j == grid[i].length - 1) {
        minPath = Math.min(minPath, path + grid[i][j]);
        return;
    }
    if (i == grid.length - 1)
        backtracking(grid, i, j + 1, path + grid[i][j]);
    else if (j == grid[i].length - 1)
        backtracking(grid, i + 1, j, path + grid[i][j]);
    else {
        backtracking(grid, i, j + 1, path + grid[i][j]);
        backtracking(grid, i + 1, j, path + grid[i][j]);
    }
}
}

```

这个方法在测试用例很大的时候超时。用动态规划做。

```

class Solution {
    public int minPathSum(int[][][] grid) {
        int m = grid.length;
        int n = grid[0].length;
        int[][] path = new int[m][n];
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (i == 0 && j == 0)
                    path[i][j] = grid[0][0];
                else if (i == 0)
                    path[i][j] = path[i][j - 1] + grid[i][j];
                else if (j == 0)
                    path[i][j] = path[i - 1][j] + grid[i][j];
                else
                    path[i][j] = Math.min(path[i - 1][j], path[i][j - 1]) + grid[i][j];
            }
        }
        return path[m - 1][n - 1];
    }
}

```

136. 只出现一次的数字 Easy

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

说明：

你的算法应该具有线性时间复杂度。你可以不使用额外空间来实现吗？

示例 1：

```
输入: [2,2,1]
输出: 1
```

示例 2：

```
输入: [4,1,2,1,2]
输出: 4
```

```
class Solution {
    public int singleNumber(int[] nums) {
        int res = 0;
        for (int num : nums) {
            res ^= num;
        }
        return res;
    }
}
```

不需要额外空间的方法，就往位运算上想。

39. 组合总和 Medium

给定一个无重复元素的正整数数组 `candidates` 和一个正整数 `target`，找出 `candidates` 中所有可以使数字和为目标数 `target` 的唯一组合。

`candidates` 中的数字可以无限制重复被选取。如果至少一个所选数字数量不同，则两种组合是唯一的。

对于给定的输入，保证和为 `target` 的唯一组合数少于 150 个。

示例 1：

```
输入: candidates = [2,3,6,7], target = 7
输出: [[7],[2,2,3]]
```

示例 2:

```
输入: candidates = [2,3,5], target = 8
输出: [[2,2,2,2],[2,3,3],[3,5]]
```

示例 3:

```
输入: candidates = [2], target = 1
输出: []
```

示例 4:

```
输入: candidates = [1], target = 1
输出: [[1]]
```

示例 5:

```
输入: candidates = [1], target = 2
输出: [[1,1]]
```

提示:

- $1 \leq \text{candidates.length} \leq 30$
- $1 \leq \text{candidates}[i] \leq 200$
- `candidate` 中的每个元素都是独一无二的。
- $1 \leq \text{target} \leq 500$

```
class Solution {
    private List<List<Integer>> res;

    public List<List<Integer>> combinationSum(int[] candidates, int target) {
        res = new ArrayList<>();
        Arrays.sort(candidates);
        backtracking(candidates, 0, candidates.length - 1, 0, new ArrayList<>(),
target);
        return res;
    }

    private void backtracking(int[] candidates, int curDigit, int totalDigit, int sum,
List<Integer> item, int target) {
        if (sum > target)
            return;
        if (sum == target) {
            res.add(new ArrayList<>(item));
            return;
        }
    }
}
```

```

        for (int i = curDigit; i <= totalDigit; i++) {
            item.add(candidates[i]);
            dfs(candidates, i, totalDigit, sum + candidates[i], item, target);
            item.remove(item.size() - 1);
        }
    }
}

```

283. 移动零 Easy

给定一个数组 `nums`，编写一个函数将所有 `0` 移动到数组的末尾，同时保持非零元素的相对顺序。

示例：

```

输入: [0,1,0,3,12]
输出: [1,3,12,0,0]

```

说明：

1. 必须在原数组上操作，不能拷贝额外的数组。
2. 尽量减少操作次数。

```

class Solution {
    public void moveZeroes(int[] nums) {
        int cursor = 0;
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] != 0)
                nums[cursor++] = nums[i];
        }
        while (cursor < nums.length)
            nums[cursor++] = 0;
    }
}

```

这么些应该是不合题意的。

```

class Solution {
    public void moveZeroes(int[] nums) {
        for (int left = 0, right = 0; right < nums.length; right++) {
            if (nums[right] != 0)
                swap(nums, left++, right);
        }
    }
}

```

```
private void swap(int[] nums, int i, int j) {  
    int temp = nums[i];  
    nums[i] = nums[j];  
    nums[j] = temp;  
}  
}
```

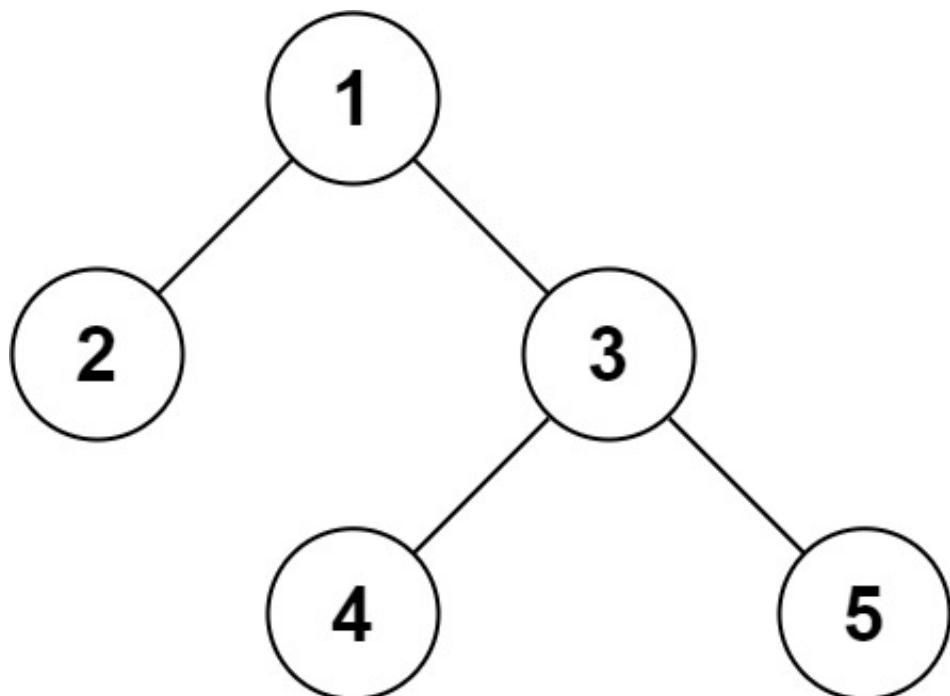
297. 二叉树的序列化与反序列化 Hard

序列化是将一个数据结构或者对象转换为连续的比特位的操作，进而可以将转换后的数据存储在一个文件或者内存中，同时也可以通过网络传输到另一个计算机环境，采取相反方式重构得到原数据。

请设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。

提示：输入输出格式与 LeetCode 目前使用的方式一致，详情请参阅 [LeetCode 序列化二叉树的格式](#)。你并非必须采取这种方式，你也可以采用其他的方法解决这个问题。

示例 1：



输入: root = [1,2,3,null,null,4,5]

输出: [1,2,3,null,null,4,5]

示例 2：

输入: root = []

输出: []

示例 3:

```
输入: root = [1]
输出: [1]
```

示例 4:

```
输入: root = [1,2]
输出: [1,2]
```

提示:

- 树中结点数在范围 `[0, 104]` 内
- `-1000 <= Node.val <= 1000`

```
public class Codec {

    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        if (root == null)
            return "null";
        String left = serialize(root.left);
        String right = serialize(root.right);
        return root.val + "," + left + "," + right;
    }

    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        if (data == "")
            return null;
        Queue<String> queue = new LinkedList<>(Arrays.asList(data.split(",")));
        return deserialize(queue);
    }

    private TreeNode deserialize(Queue<String> queue) {
        String poll = queue.poll();
        if ("null".equals(poll))
            return null;
        TreeNode root = new TreeNode(Integer.parseInt(poll));
        root.left = deserialize(queue);
        root.right = deserialize(queue);
        return root;
    }
}
```

240. 搜索二维矩阵 II Medium

编写一个高效的算法来搜索 $*m * x *n *$ 矩阵 `matrix` 中的一个目标值 `target`。该矩阵具有以下特性：

- 每行的元素从左到右升序排列。
- 每列的元素从上到下升序排列。

示例 1：

1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

```
输入: matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],  
[18,21,23,26,30]], target = 5  
输出: true
```

示例 2：

1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

```
输入: matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],  
[18,21,23,26,30]], target = 20
```

```
输出: false
```

提示:

- `m == matrix.length`
- `n == matrix[i].length`
- `1 <= n, m <= 300`
- `-109 <= matrix[i][j] <= 109`
- 每行的所有元素从左到右升序排列
- 每列的所有元素从上到下升序排列
- `-109 <= target <= 109`

```

class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        for (int i = 0; i < matrix.length; i++) {
            for (int j = matrix[i].length - 1; j >= 0; j--) {
                if (matrix[i][j] == target)
                    return true;
                else if (matrix[i][j] < target)
                    break;
            }
        }
        return false;
    }
}

```

可以优化。

```

class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        int i = 0, j = matrix[0].length - 1;
        while (i < matrix.length && j >= 0) {
            if (matrix[i][j] == target)
                return true;
            else if (matrix[i][j] < target)
                i++;
            else if (matrix[i][j] > target)
                j--;
        }
        return false;
    }
}

```

62. 不同路径 Medium

一个机器人位于一个 $m \times n$ 网格的左上角（起始点在下图中标记为“Start”）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

问总共有多少条不同的路径？

示例 1：



输入: m = 3, n = 7

输出: 28

示例 2:

输入: m = 3, n = 2

输出: 3

解释:

从左上角开始, 总共有 3 条路径可以到达右下角。

1. 向右 -> 向下 -> 向下
2. 向下 -> 向下 -> 向右
3. 向下 -> 向右 -> 向下

示例 3:

输入: m = 7, n = 3

输出: 28

示例 4:

输入: m = 3, n = 3

输出: 6

提示:

- $1 \leq m, n \leq 100$
- 题目数据保证答案小于等于 $2 * 10^9$

```
class Solution {
    private int pathNum = 0;

    public int uniquePaths(int m, int n) {
        backtracking(m, n, 1, 1);
        return pathNum;
    }
}
```

```

private void backtracking(int m, int n, int i, int j) {
    if (i == m && j == n) {
        pathNum++;
        return;
    }
    if (i < 1 || i > m || j < 1 || j > n)
        return;
    if (i == m)
        backtracking(m, n, i, j + 1);
    else if (j == n)
        backtracking(m, n, i + 1, j);
    else {
        backtracking(m, n, i + 1, j);
        backtracking(m, n, i, j + 1);
    }
}
}

```

超时。

```

class Solution {
    public int uniquePaths(int m, int n) {
        int[][] dp = new int[m + 1][n + 1];
        dp[1][1] = 1;
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (i == 1 && j == 1)
                    continue;
                dp[i][j] = dp[i][j - 1] + dp[i - 1][j];
            }
        }
        return dp[m][n];
    }
}

```

128. 最长连续序列 Medium

给定一个未排序的整数数组 `nums`，找出数字连续的最长序列（不要求序列元素在原数组中连续）的长度。

请你设计并实现时间复杂度为 $O(n)$ 的算法解决此问题。

示例 1：

输入: nums = [100,4,200,1,3,2]
输出: 4
解释: 最长数字连续序列是 [1, 2, 3, 4]。它的长度为 4。

示例 2:

输入: nums = [0,3,7,2,5,8,4,6,0,1]
输出: 9

提示:

- $0 \leq \text{nums.length} \leq 105$
- $-109 \leq \text{nums}[i] \leq 109$

```
class Solution {
    public int longestConsecutive(int[] nums) {
        HashSet<Integer> hashSet = new HashSet<>();
        for (int num : nums) {
            hashSet.add(num);
        }
        int consecutive = 0;
        for (int num : nums) {
            if (!hashSet.contains(num - 1)) {
                int temp = num;
                int cnt = 0;
                while (hashSet.contains(temp)) {
                    hashSet.remove(temp++);
                    cnt++;
                }
                consecutive = Math.max(consecutive, cnt);
            }
        }
        return consecutive;
    }
}
```

221. 最大正方形 Medium

在一个由 '0' 和 '1' 组成的二维矩阵内，找到只包含 '1' 的最大正方形，并返回其面积。

示例 1:

1	0	1	0	0
1	0	1	1	1
1	1	1	1	1
1	0	0	1	0

```
输入: matrix = [["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],
["1","0","0","1","0"]]
输出: 4
```

示例 2:

0	1
1	0

```
输入: matrix = [["0","1"],["1","0"]]
输出: 1
```

示例 3:

```
输入: matrix = [[ "0 "]]
输出: 0
```

提示:

- `m == matrix.length`
- `n == matrix[i].length`
- `1 <= m, n <= 300`
- `matrix[i][j]` 为 `'0'` 或 `'1'`

```

class Solution {
    private int maxSquare = 0;

    public int maximalSquare(char[][] matrix) {
        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[i].length; j++) {
                if (matrix[i][j] == '0')
                    continue;
                int sideLen = 1;
                while (isSquare(i, j, sideLen, matrix))
                    maxSquare = Math.max(maxSquare, sideLen++);
            }
        }
        return maxSquare * maxSquare;
    }

    private boolean isSquare(int x, int y, int sideLen, char[][] matrix) {
        if (x + sideLen > matrix.length || y + sideLen > matrix[0].length)
            return false;
        for (int i = x; i < x + sideLen; i++) {
            for (int j = y; j < y + sideLen; j++) {
                if (matrix[i][j] == '0')
                    return false;
            }
        }
        return true;
    }
}

```

暴力方法时间复杂度高，但是怎么计算暴力方法的时间复杂度了，不知道。

时间复杂度： $O(mn \cdot \min(m, n)^2)$ ，其中 m 和 n 是矩阵的行数和列数。

需要遍历整个矩阵寻找每个1，遍历矩阵的时间复杂度是 $O(mn)$ 。

对于每个可能的正方形，其边长不超过 m 和 n 中的最小值，需要遍历该正方形中的每个元素判断是不是只包含 1，遍历正方形时间复杂度是 $O(\min(m, n)^2)$ 。

总时间复杂度是 $O(mn \cdot \min(m, n)^2)$

动态规划可以优化空间复杂度。

```

class Solution {
    public int maximalSquare(char[][] matrix) {
        int[][] dp = new int[matrix.length][matrix[0].length];
        int square = 0;
        for (int i = 0; i < dp.length; i++) {
            for (int j = 0; j < dp[i].length; j++) {

```

```

        if (matrix[i][j] == '0')
            continue;
        if (i == 0 || j == 0)
            dp[i][j] = 1;
        else
            dp[i][j] = Math.min(dp[i - 1][j - 1], Math.min(dp[i - 1][j], dp[i]
[j - 1])) + 1;
        square = Math.max(square, dp[i][j]);
    }
}
return square * square;
}
}

```

198. 打家劫舍 Medium

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你 不触动警报装置的情况下，一夜之内能够偷窃到的最高金额。

示例 1：

```

输入: [1,2,3,1]
输出: 4
解释: 偷窃 1 号房屋 (金额 = 1) , 然后偷窃 3 号房屋 (金额 = 3)。
偷窃到的最高金额 = 1 + 3 = 4 。

```

示例 2：

```

输入: [2,7,9,3,1]
输出: 12
解释: 偷窃 1 号房屋 (金额 = 2), 偷窃 3 号房屋 (金额 = 9), 接着偷窃 5 号房屋 (金额 = 1)。
偷窃到的最高金额 = 2 + 9 + 1 = 12 。

```

提示：

- $1 \leq \text{nums.length} \leq 100$
- $0 \leq \text{nums}[i] \leq 400$

```

class Solution {
public int rob(int[] nums) {
    int[][] dp = new int[nums.length][2];

```

```

dp[0][0] = 0;
dp[0][1] = nums[0];
int max = nums[0];
for (int i = 1; i < dp.length; i++) {
    dp[i][0] = Math.max(dp[i - 1][1], dp[i - 1][0]);
    dp[i][1] = dp[i - 1][0] + nums[i];
    max = Math.max(max, Math.max(dp[i][1], dp[i][0]));
}
return max;
}
}

```

`dp[i][0]` 表示第 `i` 间房不偷窃后的总金额burglary。

`dp[i][1]` 表示第 `i` 间房偷窃后的总金额。

```

class Solution {
    public int rob(int[] nums) {
        if (nums.length == 1)
            return nums[0];
        else if (nums.length == 2)
            return Math.max(nums[0], nums[1]);
        int[] dp = new int[nums.length];
        dp[0] = nums[0];
        dp[1] = Math.max(nums[0], nums[1]);
        for (int i = 2; i < nums.length; i++) {
            dp[i] = Math.max(dp[i - 1], dp[i - 2] + nums[i]);
        }
        return dp[nums.length - 1];
    }
}

```

`dp[i]` 表示前 `i` 间房屋能偷窃到的最高总金额。

这道题要求在输出可以偷窃的最大金额的同时，还输出得到最大金额的路径。

213. 打家劫舍 II Medium

你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。这个地方所有的房屋都 **围成一圈**，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你 **在不触动警报装置的情况下**，今晚能够偷窃到的最高金额。

示例 1：

输入: nums = [2,3,2]

输出: 3

解释: 你不能先偷窃 1 号房屋 (金额 = 2) , 然后偷窃 3 号房屋 (金额 = 2) , 因为他们是相邻的。

示例 2:

输入: nums = [1,2,3,1]

输出: 4

解释: 你可以先偷窃 1 号房屋 (金额 = 1) , 然后偷窃 3 号房屋 (金额 = 3) 。

偷窃到的最高金额 = 1 + 3 = 4 。

示例 3:

输入: nums = [0]

输出: 0

提示:

- $1 \leq \text{nums.length} \leq 100$
- $0 \leq \text{nums}[i] \leq 1000$

```
class Solution {
    public int rob(int[] nums) {
        if (nums.length == 1)
            return nums[0];
        if (nums.length == 2)
            return Math.max(nums[0], nums[1]);
        return Math.max(rob(nums, 0, nums.length - 2), rob(nums, 1, nums.length - 1));
    }

    private int rob(int[] nums, int start, int end) {
        int[][] dp = new int[end - start + 1][2];
        dp[0][0] = 0;
        dp[0][1] = nums[start];
        for (int i = 1; i <= end - start; i++) {
            dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1]);
            dp[i][1] = dp[i - 1][0] + nums[start + i];
        }
        return Math.max(dp[end - start][0], dp[end - start][1]);
    }
}
```

这样构造 `dp[][]` 比较好理解, 构造一维的 `dp[]` 数组不会。

```
class Solution {
    public int rob(int[] nums) {
```

```

int length = nums.length;
if (length == 1)
    return nums[0];
if (length == 2)
    return Math.max(nums[0], nums[1]);

int maxMoney = 0;
int[] dp = new int[nums.length - 1];
dp[0] = nums[0];
dp[1] = Math.max(nums[0], nums[1]);
for (int i = 2; i < dp.length; i++) {
    dp[i] = Math.max(dp[i - 2] + nums[i], dp[i - 1]);
}
maxMoney = dp[dp.length - 1];
dp[0] = nums[1];
dp[1] = Math.max(nums[1], nums[2]);
for (int i = 2; i < dp.length; i++) {
    dp[i] = Math.max(dp[i - 2] + nums[i + 1], dp[i - 1]);
}
return Math.max(maxMoney, dp[dp.length - 1]);
}
}

```

```

class Solution {
    public int rob(int[] nums) {
        int length = nums.length;
        if (length == 1) {
            return nums[0];
        } else if (length == 2) {
            return Math.max(nums[0], nums[1]);
        }
        return Math.max(robRange(nums, 0, length - 2), robRange(nums, 1, length - 1));
    }

    public int robRange(int[] nums, int start, int end) {
        int first = nums[start], second = Math.max(nums[start], nums[start + 1]);
        for (int i = start + 2; i <= end; i++) {
            int temp = second;
            second = Math.max(first + nums[i], second);
            first = temp;
        }
        return second;
    }
}

```

337. 打家劫舍 III Medium

在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为“根”。除了“根”之外，每栋房子有且只有一个“父”房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

示例 1:

输入: [3,2,3,null,3,null,1]

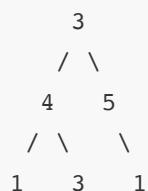


输出: 7

解释: 小偷一晚能够盗取的最高金额 = 3 + 3 + 1 = 7.

示例 2:

输入: [3,4,5,1,3,null,1]



输出: 9

解释: 小偷一晚能够盗取的最高金额 = 4 + 5 = 9.

```
class Solution {

    public int rob(TreeNode root) {
        return Math.max(burglary(root, true), burglary(root, false));
    }

    private int burglary(TreeNode root, boolean stolen) {
        if (root == null)
            return 0;
        if (stolen)
            return burglary(root.left, false) + burglary(root.right, false) + root.val;
        return Math.max(burglary(root.left, true), burglary(root.left, false))
            + Math.max(burglary(root.right, true), burglary(root.right, false));
    }
}
```

```
    }
}
```

这个方法简单好理解，但是存在大量重复计算会超时。

```
import java.util.HashMap;

class Solution {
    private HashMap<TreeNode, Integer> burglary = new HashMap<>();
    private HashMap<TreeNode, Integer> unBurglary = new HashMap<>();

    public int rob(TreeNode root) {
        return Math.max(rob(root, true), rob(root, false));
    }

    private int rob(TreeNode root, boolean stolen) {
        if (root == null)
            return 0;
        if (!burglary.containsKey(root)) {
            int val = root.val + rob(root.left, false) + rob(root.right, false);
            burglary.put(root, val);
        }
        if (!unBurglary.containsKey(root)) {
            int val = Math.max(rob(root.left, false), rob(root.left, true))
                    + Math.max(rob(root.right, false), rob(root.right, true));
            unBurglary.put(root, val);
        }
        return stolen ? burglary.get(root) : unBurglary.get(root);
    }
}
```

空间复杂度可以进一步优化，省略哈希表，运行时间上也可以更快。

```
class Solution {
    public int rob(TreeNode root) {
        int[] state = dfs(root);
        return Math.max(state[0], state[1]);
    }

    private int[] dfs(TreeNode root) {
        if (root == null)
            return new int[]{0, 0};
        int[] left = dfs(root.left);
        int[] right = dfs(root.right);
        int burglary = root.val + left[1] + right[1];
        int unBurglary = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);
        return new int[]{burglary, unBurglary};
    }
}
```

```
}
```

79. 单词搜索 Medium

给定一个 $m \times n$ 二维字符网格 `board` 和一个字符串单词 `word`。如果 `word` 存在于网格中，返回 `true`；否则，返回 `false`。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

示例 1：

A	B	C	E
S	F	C	S
A	D	E	E

```
输入: board = [[ "A", "B", "C", "E"], [ "S", "F", "C", "S"], [ "A", "D", "E", "E"]], word = "ABCED"
输出: true
```

示例 2：

A	B	C	E
S	F	C	S
A	D	E	E

```
输入: board = [[ "A", "B", "C", "E"], [ "S", "F", "C", "S"], [ "A", "D", "E", "E"]], word = "SEE"
输出: true
```

示例 3：

A	B	C	E
S	F	C	S
A	D	E	E

```
输入: board = [[ "A", "B", "C", "E"], [ "S", "F", "C", "S"], [ "A", "D", "E", "E"]], word = "ABCB"
输出: false
```

提示:

- `m == board.length`
- `n = board[i].length`
- `1 <= m, n <= 6`
- `1 <= word.length <= 15`
- `board` 和 `word` 仅由大小写英文字母组成

进阶：你可以使用搜索剪枝的技术来优化解决方案，使其在 `board` 更大的情况下可以更快解决问题？

```
class Solution {
    public boolean exist(char[][] board, String word) {
        boolean[][] visited = new boolean[board.length][board[0].length];
        char[] cs = word.toCharArray();
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[i].length; j++) {
                if (!visited[i][j] && board[i][j] == word.charAt(0))
                    if (dfs(board, i, j, cs, 0, visited))
                        return true;
            }
        }
        return false;
    }

    private boolean dfs(char[][] board, int i, int j, char[] cs, int index, boolean[][] visited) {
        if (i < 0 || i >= board.length || j < 0 || j >= board[0].length || visited[i][j] || board[i][j] != cs[index])
            return false;
        if (index == cs.length - 1)
            return true;
        visited[i][j] = true;
        if (dfs(board, i + 1, j, cs, index + 1, visited) ||
            dfs(board, i - 1, j, cs, index + 1, visited) ||
            dfs(board, i, j + 1, cs, index + 1, visited) ||
            dfs(board, i, j - 1, cs, index + 1, visited))
            return true;
        visited[i][j] = false;
        return false;
    }
}
```

```

    if (index == cs.length - 1)
        return true;
    visited[i][j] = true;
    boolean flag = dfs(board, i + 1, j, cs, index + 1, visited)
        || dfs(board, i - 1, j, cs, index + 1, visited)
        || dfs(board, i, j + 1, cs, index + 1, visited)
        || dfs(board, i, j - 1, cs, index + 1, visited);
    visited[i][j] = false;
    return flag;
}
}

```

152. 乘积最大子数组 Medium

给你一个整数数组 `nums`，请你找出数组中乘积最大的连续子数组（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。

示例 1：

输入： [2,3,-2,4]
 输出： 6
 解释： 子数组 [2,3] 有最大乘积 6。

示例 2：

输入： [-2,0,-1]
 输出： 0
 解释： 结果不能为 2，因为 [-2,-1] 不是子数组。

```

class Solution {
    public int maxProduct(int[] nums) {
        int[] min = new int[nums.length];
        int[] max = new int[nums.length];
        min[0] = nums[0];
        max[0] = nums[0];
        int res = nums[0];
        for (int i = 1; i < nums.length; i++) {
            if (nums[i] > 0) {
                max[i] = Math.max(nums[i], max[i - 1] * nums[i]);
                min[i] = Math.min(nums[i], min[i - 1] * nums[i]);
            } else {
                max[i] = Math.max(nums[i], min[i - 1] * nums[i]);
                min[i] = Math.min(nums[i], max[i - 1] * nums[i]);
            }
        }
        return res;
    }
}

```

```
        res = Math.max(res, Math.max(min[i], max[i]));
    }
    return res;
}
}
```

```
class Solution {
    public int maxProduct(int[] nums) {
        if (nums.length == 0)
            return nums[0];
        int max = 1, min = 1;
        int res = nums[0];
        for (int num : nums) {
            int temp = max;
            max = Math.max(num, Math.max(num * min, num * max));
            min = Math.min(num, Math.min(num * min, num * temp));
            res = Math.max(res, Math.max(max, min));
        }
        return res;
    }
}
```

739. 每日温度 Medium

请根据每日 气温 列表 `temperatures`，请计算在每一天需要等几天才会有更高的温度。如果气温在这之后都不会升高，请在该位置用 `0` 来代替。

示例 1:

```
输入: temperatures = [73,74,75,71,69,72,76,73]
输出: [1,1,4,2,1,1,0,0]
```

示例 2:

```
输入: temperatures = [30,40,50,60]
输出: [1,1,1,0]
```

示例 3:

```
输入: temperatures = [30,60,90]
输出: [1,1,0]
```

提示:

- $1 \leq \text{temperatures.length} \leq 105$
- $30 \leq \text{temperatures}[i] \leq 100$

```
class Solution {  
    public int[] dailyTemperatures(int[] temperatures) {  
        int[] res = new int[temperatures.length];  
        Stack<Integer> stack = new Stack<>();  
        for (int i = 0; i < temperatures.length; i++) {  
            while (!stack.isEmpty() && temperatures[i] > temperatures[stack.peek()])  
                res[stack.peek()] = i - stack.pop();  
            stack.push(i);  
        }  
        return res;  
    }  
}
```

560. 和为 K 的子数组 Medium

给你一个整数数组 `nums` 和一个整数 `k`，请你统计并返回该数组中和为 `k` 的连续子数组的个数。

示例 1：

```
输入: nums = [1,1,1], k = 2  
输出: 2
```

示例 2：

```
输入: nums = [1,2,3], k = 3  
输出: 2
```

提示：

- $1 \leq \text{nums.length} \leq 2 * 10^4$
- $-1000 \leq \text{nums}[i] \leq 1000$
- $-10^7 \leq k \leq 10^7$

```
class Solution {  
  
    public int subarraySum(int[] nums, int k) {  
        HashMap<Integer, Integer> prefixSum = new HashMap<>();  
        prefixSum.put(0, 1);  
        int sum = 0, count = 0;
```

```

        for (int i = 0; i < nums.length; i++) {
            sum += nums[i];
            // 下面两步操作不能交换顺序，针对测试用例[1],0的情况。
            count += prefixSum.getOrDefault(sum - k, 0);
            prefixSum.put(sum, prefixSum.getOrDefault(sum, 0) + 1);
            // 本体针对数组操作，不需要进行回溯复原，leetcode437需要进行回溯复原。
            // 这个区别是什么？
        }
        return count;
    }
}

```

```

class Solution {
    HashMap<Integer, Integer> prefixSum = new HashMap<>();

    public int subarraySum(int[] nums, int k) {
        prefixSum.put(0, 1);
        return dfs(nums, 0, 0, k);
    }

    private int dfs(int[] nums, int curDigit, int sum, int k) {
        if (curDigit >= nums.length)
            return 0;
        sum = sum + nums[curDigit];
        //下面的sum-k可以改为k-sum吗
        int count = prefixSum.getOrDefault(sum - k, 0);
        prefixSum.put(sum, prefixSum.getOrDefault(sum, 0) + 1);
        count += dfs(nums, curDigit + 1, sum, k);
        return count;
    }
}

```

```

class Solution {
    private HashMap<Integer, Integer> prefixSum;

    public int subarraySum(int[] nums, int k) {
        prefixSum = new HashMap<>();
        prefixSum.put(0, 1);
        int sum = 0;
        int cnt = 0;
        for (int num : nums) {
            sum += num;
            cnt += prefixSum.getOrDefault(sum - k, 0);
            prefixSum.put(sum, prefixSum.getOrDefault(sum, 0) + 1);
        }
        return cnt;
    }
}

```

287. 寻找重复数 Medium

给定一个包含 $n + 1$ 个整数的数组 `nums`，其数字都在 1 到 n 之间（包括 1 和 n ），可知至少存在一个重复的整数。

假设 `nums` 只有一个重复的整数，找出这个重复的数。

你设计的解决方案必须不修改数组 `nums` 且只用常量级 $O(1)$ 的额外空间。

示例 1：

```
输入: nums = [1,3,4,2,2]
输出: 2
```

示例 2：

```
输入: nums = [3,1,3,4,2]
输出: 3
```

示例 3：

```
输入: nums = [1,1]
输出: 1
```

示例 4：

```
输入: nums = [1,1,2]
输出: 1
```

提示：

- $1 \leq n \leq 105$
- $\text{nums.length} == n + 1$
- $1 \leq \text{nums}[i] \leq n$
- `nums` 中只有一个整数出现两次或多次，其余整数均只出现一次

进阶：

- 如何证明 `nums` 中至少存在一个重复的数字？
- 你可以设计一个线性级时间复杂度 $O(n)$ 的解决方案吗？

```
class Solution {
    public int findDuplicate(int[] nums) {
```

```

int slow = 0, fast = 0;
do {
    slow = nums[slow];
    fast = nums[nums[fast]];
} while (slow != fast);
slow = 0;
while (slow != fast) {
    slow = nums[slow];
    fast = nums[fast];
}
return slow;
}
}

```

394. 字符串解码 Medium

给定一个经过编码的字符串，返回它解码后的字符串。

编码规则为: `k[encoded_string]`，表示其中方括号内部的 *encoded_string* 正好重复 *k* 次。注意 *k* 保证为正整数。

你可以认为输入字符串总是有效的；输入字符串中没有额外的空格，且输入的方括号总是符合格式要求的。

此外，你可以认为原始数据不包含数字，所有的数字只表示重复的次数 *k*，例如不会出现像 `3a` 或 `2[4]` 的输入。

示例 1：

```

输入: s = "3[a]2[bc]"
输出: "aaabcbc"

```

示例 2：

```

输入: s = "3[a2[c]]"
输出: "accaccacc"

```

示例 3：

```

输入: s = "2[abc]3[cd]ef"
输出: "abcabcccdcdcef"

```

示例 4：

```

输入: s = "abc3[cd]xyz"
输出: "abcccdcdxyz"

```

```

class Solution {
    public String decodeString(String s) {
        int num = 0;
        StringBuffer sb = new StringBuffer();
        Stack<Integer> numStack = new Stack<>();
        Stack<StringBuffer> sbStack = new Stack<>();
        for (char ch : s.toCharArray()) {
            if (ch == '[') {
                sbStack.push(sb);
                numStack.push(num);
                sb = new StringBuffer();
                num = 0;
            } else if (ch == ']') {
                StringBuffer popSb = sbStack.pop();
                int popNum = numStack.pop();
                for (int i = 0; i < popNum; i++) {
                    popSb.append(sb);
                }
                sb = popSb;
            } else if (Character.isDigit(ch)) {
                num = num * 10 + Integer.valueOf(ch - '0');
            } else if (Character.isLetter(ch)) {
                sb.append(ch);
            }
        }
        return sb.toString();
    }
}

```

208. 实现 Trie (前缀树) Medium

Trie (发音类似 "try") 或者说 前缀树 是一种树形数据结构，用于高效地存储和检索字符串数据集中的键。这一数据结构有相当多的应用情景，例如自动补完和拼写检查。

请你实现 Trie 类：

- `Trie()` 初始化前缀树对象。
- `void insert(String word)` 向前缀树中插入字符串 `word`。
- `boolean search(String word)` 如果字符串 `word` 在前缀树中，返回 `true` (即，在检索之前已经插入)；否则，返回 `false`。
- `boolean startsWith(String prefix)` 如果之前已经插入的字符串 `word` 的前缀之一为 `prefix`，返回 `true`；否则，返回 `false`。

示例：

输入

```
["Trie", "insert", "search", "search", "startsWith", "insert", "search"]
[], ["apple"], ["apple"], ["app"], ["app"], ["app"], ["app"]]
```

输出

```
[null, null, true, false, true, null, true]
```

解释

```
Trie trie = new Trie();
trie.insert("apple");
trie.search("apple");    // 返回 True
trie.search("app");      // 返回 False
trie.startsWith("app"); // 返回 True
trie.insert("app");
trie.search("app");      // 返回 True
```

提示:

- $1 \leq \text{word.length}, \text{prefix.length} \leq 2000$
- `word` 和 `prefix` 仅由小写英文字母组成
- `insert`、`search` 和 `startsWith` 调用次数 总计 不超过 $3 * 10^4$ 次

```
class Trie {
    private node root;

    public Trie() {
        root = new node();
    }

    public void insert(String word) {
        node cur = root;
        char[] cs = word.toCharArray();
        for (char ch : cs) {
            if (cur.next[ch - 'a'] == null)
                cur.next[ch - 'a'] = new node();
            cur = cur.next[ch - 'a'];
        }
        cur.isEnd = true;
    }

    public boolean search(String word) {
        node cur = root;
        char[] cs = word.toCharArray();
        for (char ch : cs) {
            if (cur.next[ch - 'a'] == null)
                return false;
            cur = cur.next[ch - 'a'];
        }
    }
}
```

```

        return cur.isEnd;
    }

    public boolean startsWith(String prefix) {
        node cur = root;
        char[] cs = prefix.toCharArray();
        for (char ch : cs) {
            if (cur.next[ch - 'a'] == null)
                return false;
            cur = cur.next[ch - 'a'];
        }
        return true;
    }

    class node {
        boolean isEnd;
        node[] next;

        public node() {
            this.next = new node[26];
        }
    }
}

```

55. 跳跃游戏 Medium

给定一个非负整数数组 `nums`，你最初位于数组的第一个下标。

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下标。

示例 1:

```

输入: nums = [2,3,1,1,4]
输出: true
解释: 可以先跳 1 步，从下标 0 到达下标 1，然后再从下标 1 跳 3 步到达最后一个下标。

```

示例 2:

```

输入: nums = [3,2,1,0,4]
输出: false
解释: 无论怎样，总会到达下标为 3 的位置。但该下标的最大跳跃长度是 0，所以永远不可能到达最后一个下标。

```

```

class Solution {
    public boolean canJump(int[] nums) {

```

```

boolean[ ] reachable = new boolean[nums.length];
int end = nums.length - 1;
reachable[end] = true;
for (int i = nums.length - 2; i >= 0; i--) {
    for (int step = 1; step <= nums[i] && !reachable[i] && i + step <= end;
step++) {
        if (reachable[i + step])
            reachable[i] = true;
    }
}
return reachable[0];
}
}

```

这个方法时间复杂度最差到平方阶，可以优化一下到线性阶。官方题解采用贪心方法，从第一个开始计算。

```

class Solution {
    public boolean canJump(int[ ] nums) {
        int end = nums.length - 1;
        int lastIndex = end;
        for (int i = nums.length - 2; i >= 0; i--) {
            if (i + nums[i] >= lastIndex)
                lastIndex = i;
        }
        return 0 + nums[0] >= lastIndex;
    }
}

```

```

class Solution {
    public boolean canJump(int[ ] nums) {
        int index = 0;
        int left = 0;
        while (index <= left) {
            left = Math.max(left, index + nums[index++]);
            if (left >= nums.length - 1)
                return true;
        }
        return false;
    }
}

```

207. 课程表 Medium

你这个学期必须选修 `numCourses` 门课程，记为 `0` 到 `numCourses - 1`。

在选修某些课程之前需要一些先修课程。先修课程按数组 `prerequisites` 给出，其中 `prerequisites[i] = [ai, bi]`，表示如果要学习课程 `ai` 则 必须 先学习课程 `bi`。

- 例如，先修课程对 `[0, 1]` 表示：想要学习课程 `0`，你需要先完成课程 `1`。

请你判断是否可能完成所有课程的学习？如果可以，返回 `true`；否则，返回 `false`。

示例 1：

输入：`numCourses = 2, prerequisites = [[1,0]]`

输出：`true`

解释：总共有 2 门课程。学习课程 1 之前，你需要完成课程 0。这是可能的。

示例 2：

输入：`numCourses = 2, prerequisites = [[1,0],[0,1]]`

输出：`false`

解释：总共有 2 门课程。学习课程 1 之前，你需要先完成课程 0；并且学习课程 0 之前，你还应先完成课程 1。这是不可能的。

提示：

- `1 <= numCourses <= 105`
- `0 <= prerequisites.length <= 5000`
- `prerequisites[i].length == 2`
- `0 <= ai, bi < numCourses`
- `prerequisites[i]` 中的所有课程对 互不相同

```
class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        int[] indeg = new int[numCourses];
        HashMap<Integer, List<Integer>> hashMap = new HashMap<>();
        Queue<Integer> queue = new LinkedList<>();
        int cnt = 0;
        for (int[] prerequisite : prerequisites) {
            indeg[prerequisite[0]]++;
            hashMap.putIfAbsent(prerequisite[1], new ArrayList<>());
            hashMap.get(prerequisite[1]).add(prerequisite[0]);
        }
        for (int i = 0; i < indeg.length; i++) {
            if (indeg[i] != 0)
                continue;
            queue.offer(i);
            cnt++;
        }
        while (!queue.isEmpty()) {
            int cur = queue.poll();
            for (int next : hashMap.get(cur)) {
                indeg[next]--;
                if (indeg[next] == 0)
                    queue.offer(next);
            }
        }
        return cnt == numCourses;
    }
}
```

```

    }
    while (!queue.isEmpty()) {
        int poll = queue.poll();
        List<Integer> courses = hashMap.get(poll);
        if (courses == null)
            continue;
        for (int course : courses) {
            indeg[course]--;
            if (indeg[course] == 0) {
                queue.offer(course);
                cnt++;
            }
        }
    }
    return cnt == numCourses;
}
}

```

210. 课程表 II Medium

现在你总共有 n 门课需要选，记为 0 到 $n-1$ 。

在选修某些课程之前需要一些先修课程。例如，想要学习课程 0 ，你需要先完成课程 1 ，我们用一个匹配来表示他们: $[0,1]$

给定课程总量以及它们的先决条件，返回你为了学完所有课程所安排的学习顺序。

可能会有多个正确的顺序，你只要返回一种就可以了。如果不可能完成所有课程，返回一个空数组。

示例 1:

输入: $2, [[1,0]]$

输出: $[0,1]$

解释: 总共有 2 门课程。要学习课程 1，你需要先完成课程 0。因此，正确的课程顺序为 $[0,1]$ 。

示例 2:

输入: $4, [[1,0],[2,0],[3,1],[3,2]]$

输出: $[0,1,2,3]$ or $[0,2,1,3]$

解释: 总共有 4 门课程。要学习课程 3，你应该先完成课程 1 和课程 2。并且课程 1 和课程 2 都应该排在课程 0 之后。

因此，一个正确的课程顺序是 $[0,1,2,3]$ 。另一个正确的排序是 $[0,2,1,3]$ 。

说明:

1. 输入的先决条件是由**边缘列表**表示的图形，而不是邻接矩阵。详情请参见[图的表示法](#)。

2. 你可以假定输入的先决条件中没有重复的边。

提示:

1. 这个问题相当于查找一个循环是否存在与有向图中。如果存在循环，则不存在拓扑排序，因此不可能选取所有课程进行学习。
2. [通过 DFS 进行拓扑排序](#) - 一个关于Coursera的精彩视频教程（21分钟），介绍拓扑排序的基本概念。
3. 拓扑排序也可以通过[BFS](#) 完成。

```
class Solution {  
    public int[] findOrder(int numCourses, int[][] prerequisites) {  
        ArrayList<List<Integer>> courses = new ArrayList<List<Integer>>();  
        int[] res = new int[numCourses];  
        int[] indeg = new int[numCourses];  
        for (int i = 0; i < numCourses; i++) {  
            courses.add(new ArrayList<>());  
        }  
        for (int[] course : prerequisites) {  
            courses.get(course[1]).add(course[0]);  
            indeg[course[0]]++;  
        }  
        Queue<Integer> queue = new LinkedList<Integer>();  
        int count = 0;  
        for (int i = 0; i < indeg.length; i++) {  
            if (indeg[i] == 0) {  
                queue.offer(i);  
                res[count++] = i;  
            }  
        }  
        while (!queue.isEmpty()) {  
            Integer b = queue.poll();  
            for (int a : courses.get(b)) {  
                indeg[a]--;  
                if (indeg[a] == 0) {  
                    queue.offer(a);  
                    res[count++] = a;  
                }  
            }  
        }  
        return count == numCourses ? res : new int[]{};  
    }  
}
```

494. 目标和 Medium

给你一个整数数组 `nums` 和一个整数 `target`。

向数组中的每个整数前添加 `'+'` 或 `'-'`，然后串联起所有整数，可以构造一个 **表达式**：

- 例如，`nums = [2, 1]`，可以在 `2` 之前添加 `'+'`，在 `1` 之前添加 `'-'`，然后串联起来得到表达式 `"+2-1"`。

返回可以通过上述方法构造的、运算结果等于 `target` 的不同 **表达式** 的数目。

示例 1：

输入: `nums = [1,1,1,1,1], target = 3`

输出: 5

解释: 一共有 5 种方法让最终目标和为 3。

`-1 + 1 + 1 + 1 + 1 = 3`

`+1 - 1 + 1 + 1 + 1 = 3`

`+1 + 1 - 1 + 1 + 1 = 3`

`+1 + 1 + 1 - 1 + 1 = 3`

`+1 + 1 + 1 + 1 - 1 = 3`

示例 2：

输入: `nums = [1], target = 1`

输出: 1

提示:

- `1 <= nums.length <= 20`
- `0 <= nums[i] <= 1000`
- `0 <= sum(nums[i]) <= 1000`
- `-1000 <= target <= 1000`

前缀和不适合在这里使用，因为存在数组开头和末尾元素组成的答案，使用前缀和无法遍历到这种组合。

```
class Solution {
    public int findTargetSumWays(int[] nums, int target) {
        return dfs(nums, 0, nums[0], target) + dfs(nums, 0, -nums[0], target);
    }

    private int dfs(int[] nums, int index, int sum, int target) {
        if (index == nums.length - 1)
            return sum == target ? 1 : 0;
        return dfs(nums, index + 1, sum + nums[index + 1], target)
            + dfs(nums, index + 1, sum - nums[index + 1], target);
    }
}
```

dfs时间复杂度高。

```
class Solution {
    public int findTargetSumWays(int[] nums, int target) {
        int sum = Arrays.stream(nums).sum();
        if (target + sum < 0 || ((sum + target) & 1) != 0)
            return 0;
        int k = (target + sum) >> 1;
        int[][] dp = new int[nums.length + 1][k + 1];
        dp[0][0] = 1;
        for (int i = 0; i < nums.length; i++) {
            for (int j = 0; j <= k; j++) {
                if (j >= nums[i])
                    dp[i + 1][j] = dp[i][j] + dp[i][j - nums[i]];
                else
                    dp[i + 1][j] = dp[i][j];
            }
        }
        return dp[nums.length][k];
    }
}
```

```
class Solution {
    public int findTargetSumWays(int[] nums, int target) {
        int sum = Arrays.stream(nums).sum();
        int diff;
        //这里 diff = sum - target 也可以AC, 但是为什么了?
        if ((diff = sum + target) < 0 || (diff & 1) != 0)
            return 0;
        int k = diff >> 1;
        int[][] dp = new int[nums.length + 1][k + 1];
        dp[0][0] = 1;
        //这里循环的终止条件注意一下, 0-1背包的循环顺序。
        for (int i = 1; i < dp.length; i++) {
            for (int j = 0; j < dp[i].length; j++) {
                if (j < nums[i - 1])
                    dp[i][j] = dp[i - 1][j];
                else
                    dp[i][j] = dp[i - 1][j - nums[i - 1]] + dp[i - 1][j];
            }
        }
        return dp[nums.length][k];
    }
}
```

二维dp数组改为一维dp数组。

647. 回文子串 Medium

给你一个字符串 `s`，请你统计并返回这个字符串中 回文子串 的数目。

回文字符串 是正着读和倒过来读一样的字符串。

子字符串 是字符串中的由连续字符组成的一个序列。

具有不同开始位置或结束位置的子串，即使是由相同的字符组成，也会被视作不同的子串。

示例 1：

```
输入: s = "abc"
输出: 3
解释: 三个回文子串: "a", "b", "c"
```

示例 2：

```
输入: s = "aaa"
输出: 6
解释: 6个回文子串: "a", "a", "a", "aa", "aa", "aaa"
```

提示：

- `1 <= s.length <= 1000`
- `s` 由小写英文字母组成

```
class Solution {
    public int countSubstrings(String s) {
        char[] charArray = s.toCharArray();
        int count = charArray.length;
        for (int i = 0; i < charArray.length - 1; i++) {
            for (int j = i + 1; j < charArray.length; j++) {
                count += isPalindromic(charArray, i, j);
            }
        }
        return count;
    }

    private int isPalindromic(char[] charArray, int left, int right) {
        while (left <= right) {
            if (charArray[left++] != charArray[right--])
                return 0;
        }
        return 1;
    }
}
```

时间复杂度达到立方阶。

```
class Solution {
    public int countSubstrings(String s) {
        char[] charArray = s.toCharArray();
        boolean[][] dp = new boolean[s.length()][s.length()];
        int count = 0;
        for (int j = 0; j < charArray.length; j++) {
            for (int i = 0; i <= j; i++) {
                if (charArray[i] == charArray[j] && (i + 2 >= j || dp[i + 1][j - 1])) {
                    dp[i][j] = true;
                    count++;
                }
            }
        }
        return count;
    }
}
```

时间复杂度到达平方阶，`dp[i][j]`表示起点为*i*，终点为*j*（包括两端点）的字符串是否是回文串。

```
class Solution {
    public int countSubstrings(String s) {
        char[] cs = s.toCharArray();
        int cnt = 0;
        for (int i = 0; i < cs.length - 1; i++) {
            cnt += isPalindrome(cs, i, i);
            cnt += isPalindrome(cs, i, i + 1);
        }
        return cnt;
    }

    private int isPalindrome(char[] cs, int i, int j) {
        int cnt = 0;
        while (i >= 0 && j <= cs.length - 1) {
            if (cs[i--] != cs[j++])
                break;
            cnt++;
        }
        return cnt;
    }
}
```

17. 电话号码的字母组合 Medium

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。答案可以按 **任意顺序** 返回。

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 1：

```
输入: digits = "23"
输出: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]
```

示例 2：

```
输入: digits = ""
输出: []
```

示例 3：

```
输入: digits = "2"
输出: ["a", "b", "c"]
```

提示：

- `0 <= digits.length <= 4`
- `digits[i]` 是范围 `['2', '9']` 的一个数字。

```
class Solution {

    String[] mapping = { "", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv",
    "wxyz" };
    List<String> res = new ArrayList<>();

    public List<String> letterCombinations(String digits) {
        if (digits.length() == 0)
            return res;
    }
}
```

```

        char[] cs = digits.toCharArray();
        backtracking(cs, 0, cs.length - 1, new StringBuffer());
        return res;
    }

    private void backtracking(char[] cs, int index, int end, StringBuffer sb) {
        if (index > end) {
            res.add(sb.toString());
            return;
        }
        char[] item = mapping[cs[index] - '0'].toCharArray();
        for (int i = 0; i < item.length; i++) {
            sb.append(item[i]);
            backtracking(cs, index + 1, end, sb);
            sb.deleteCharAt(index);
        }
    }
}

```

这道题不难，回溯也容易想到，但是基本功不扎实，如何把按键字符对应到字符串不清楚，可以使用 `String[]` 数组，也可以用 `HashMap`。

调用回溯函数传参数的时候，使用 `StringBuffer` 比 `String` 快。

```

class Solution {
    List<String> res = new ArrayList<>();
    HashMap<Character, String> hashMap = new HashMap<>();

    public List<String> letterCombinations(String digits) {
        if (digits.length() == 0)
            return res;
        hashMap.put('2', "abc");
        hashMap.put('3', "def");
        hashMap.put('4', "ghi");
        hashMap.put('5', "jkl");
        hashMap.put('6', "mno");
        hashMap.put('7', "pqrs");
        hashMap.put('8', "tuv");
        hashMap.put('9', "wxyz");
        char[] cs = digits.toCharArray();
        dfs(cs, 0, digits.length() - 1, new StringBuffer());
        return res;
    }

    private void dfs(char[] cs, int curDigit, int totalDigit, StringBuffer sb) {
        if (curDigit > totalDigit) {
            res.add(sb.toString());
            return;
        }

```

```

    }
    for (char ch : hashMap.get(cs[curDigit]).toCharArray()) {
        sb.append(ch);
        dfs(cs, curDigit + 1, totalDigit, sb);
        sb.deleteCharAt(sb.length() - 1);
    }
}
}

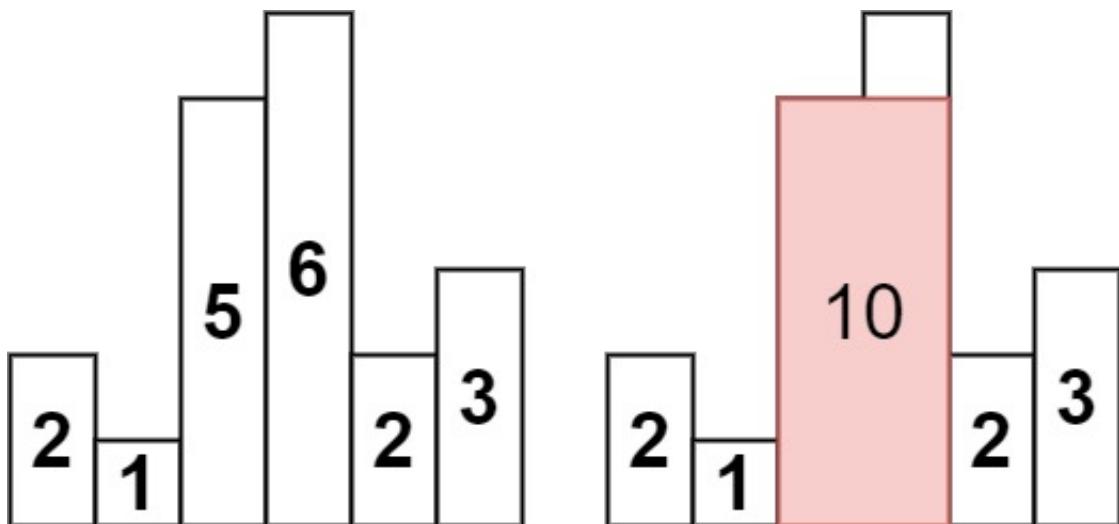
```

84. 柱状图中最大的矩形 Hard

给定 n 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1。

求在该柱状图中，能够勾勒出来的矩形的最大面积。

示例 1：

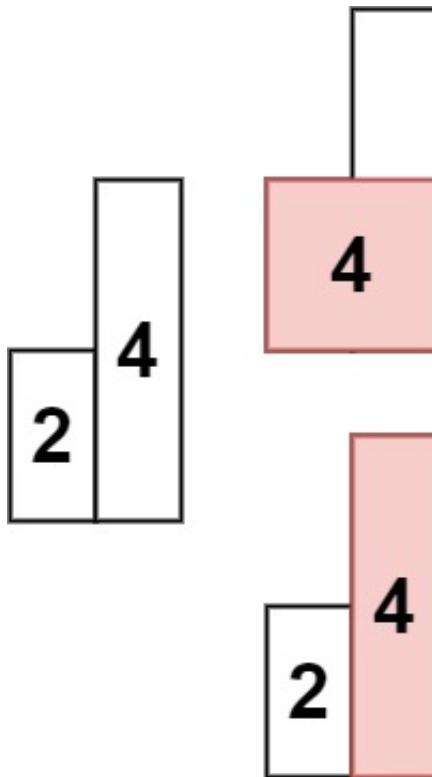


输入: heights = [2,1,5,6,2,3]

输出: 10

解释: 最大的矩形为图中红色区域，面积为 10

示例 2：



输入: heights = [2,4]

输出: 4

提示:

- `1 <= heights.length <=105`
- `0 <= heights[i] <= 104`

```
class Solution {
    public int largestRectangleArea(int[] heights) {
        Stack<Integer> stack = new Stack<>();
        stack.push(-1);
        int area = 0;
        for (int i = 0; i < heights.length; i++) {
            while (stack.peek() != -1 && heights[stack.peek()] >= heights[i]) {
                int left, right;
                int pop = stack.pop();
                left = stack.peek();
                right = i;
                area = Math.max(area, (right - left - 1) * heights[pop]);
            }
            stack.push(i);
        }
        while (stack.peek() != -1) {
            int left;
            int pop = stack.pop();
            left = stack.peek();
```

```

        area = Math.max(area, (heights.length - 1 - left) * heights[pop]);
    }
    return area;
}
}

```

方法二：单调栈 + 常数优化。在方法一中，我们首先从左往右对数组进行遍历，借助单调栈求出了每根柱子的左边界，随后从右往左对数组进行遍历，借助单调栈求出了每根柱子的右边界。那么我们是否可以只遍历一次就求出答案呢？

85. 最大矩形 Hard

给定一个仅包含 0 和 1、大小为 `rows x cols` 的二维二进制矩阵，找出只包含 1 的最大矩形，并返回其面积。

示例 1：

1	0	1	0	0
1	0	1	1	1
1	1	1	1	1
1	0	0	1	0

输入：`matrix = [["1", "0", "1", "0", "0"], ["1", "0", "1", "1", "1"], ["1", "1", "1", "1", "1"], ["1", "0", "0", "1", "0"]]`

输出：6

解释：最大矩形如上图所示。

示例 2：

输入：`matrix = []`

输出：0

示例 3：

```
输入: matrix = [[ "0" ]]
输出: 0
```

示例 4:

```
输入: matrix = [[ "1" ]]
输出: 1
```

示例 5:

```
输入: matrix = [[ "0", "0" ]]
输出: 0
```

提示:

- `rows == matrix.length`
- `cols == matrix[0].length`
- `0 <= row, cols <= 200`
- `matrix[i][j]` 为 '`0`' 或 '`1`'

```
class Solution {
    public int maximalRectangle(char[][] matrix) {
        if (matrix.length == 0)
            return 0;
        int[] heights = new int[matrix[0].length];
        int area = 0;
        for (int i = 0; i < matrix.length; i++) {
            heights = getHeights(matrix[i], heights);
            area = Math.max(area, largestRectangleArea(heights));
        }
        return area;
    }

    private int[] getHeights(char[] matrix, int[] heights) {
        for (int i = 0; i < heights.length; i++) {
            heights[i] = matrix[i] == '0' ? 0 : heights[i] + 1;
        }
        return heights;
    }

    private int largestRectangleArea(int[] heights) {
        Stack<Integer> stack = new Stack<>();
        stack.push(-1);
        int area = 0;
        for (int i = 0; i < heights.length; i++) {
            while (stack.peek() != -1 && heights[stack.peek()] >= heights[i])
```

```

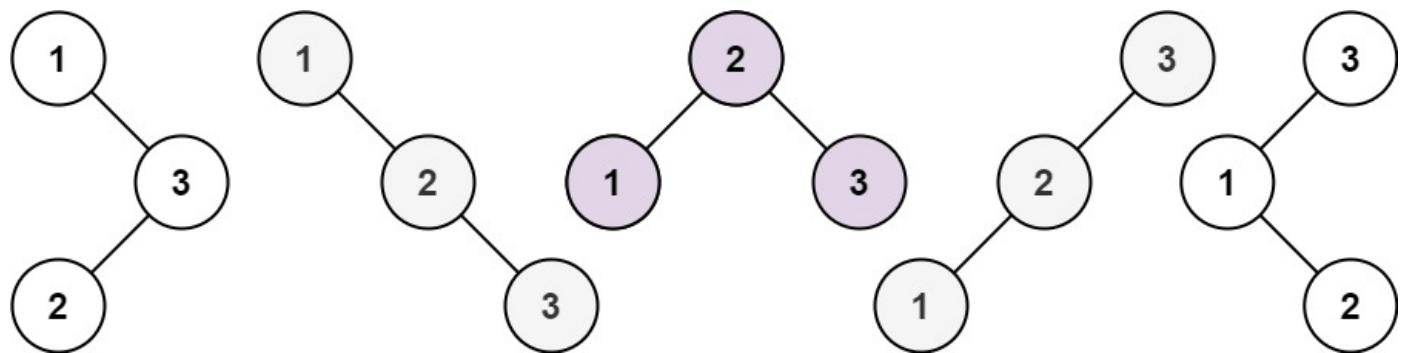
        area = Math.max(area, heights[stack.pop()] * (i - stack.peek() - 1));
        stack.push(i);
    }
    while (stack.peek() != -1) {
        area = Math.max(area, heights[stack.pop()] * (heights.length - stack.peek()
- 1));
    }
    return area;
}
}

```

96. 不同的二叉搜索树 Medium

给你一个整数 n ，求恰由 n 个节点组成且节点值从 1 到 n 互不相同的二叉搜索树有多少种？返回满足题意的二叉搜索树的种数。

示例 1：



输入： $n = 3$
输出： 5

示例 2：

输入： $n = 1$
输出： 1

提示：

- $1 \leq n \leq 19$

```

class Solution {
    public int numTrees(int n) {
        int[] dp = new int[n + 1];
        dp[0] = 1;
        for (int i = 1; i <= n; i++) {
            for (int j = 0; j <= i - 1; j++) {
                dp[i] += dp[j] * dp[i - 1 - j];
            }
        }
        return dp[n];
    }
}

```

给定一个有序序列 $1 \dots n$ ，为了构建出一棵二叉搜索树，我们可以遍历每个数字 i ，将该数字作为树根，将 $1 \dots (i-1)$ 序列作为左子树，将 $(i+1) \dots n$ 序列作为右子树。接着我们可以按照同样的方式递归构建左子树和右子树。

在上述构建的过程中，由于根的值不同，因此我们能保证每棵二叉搜索树是唯一的。

由此可见，原问题可以分解成规模较小的两个子问题，且子问题的解可以复用。因此，我们可以想到使用动态规划来求解本题。

238. 除自身以外数组的乘积 Medium

给你一个长度为 n 的整数数组 `nums`，其中 $n > 1$ ，返回输出数组 `output`，其中 `output[i]` 等于 `nums` 中除 `nums[i]` 之外其余各元素的乘积。

示例:

```

输入: [1,2,3,4]
输出: [24,12,8,6]

```

提示: 题目数据保证数组之中任意元素的全部前缀元素和后缀（甚至是整个数组）的乘积都在 32 位整数范围内。

说明: 请不要使用除法，且在 $O(n)$ 时间复杂度内完成此题。

进阶:

你可以在常数空间复杂度内完成这个题目吗？（出于对空间复杂度分析的目的，输出数组不被视为额外空间。）

```

class Solution {
    public int[] productExceptSelf(int[] nums) {
        int[] res = new int[nums.length];
        int prefix = 1;
        for (int i = 0; i < res.length; i++) {
            res[i] = prefix;
            prefix *= nums[i];
        }
        int suffix = 1;
        for (int i = res.length - 1; i >= 0; i--) {
            res[i] *= suffix;
            suffix *= nums[i];
        }
        return res;
    }
}

```

```

        prefix *= nums[i];
    }
    int suffix = 1;
    for (int i = nums.length - 1; i >= 0; i--) {
        res[i] *= suffix;
        suffix *= nums[i];
    }
    return res;
}
}

```

可以优化，将两次遍历优化为一次，但是运行时间竟然反而增加了。

```

class Solution {
    public int[] productExceptSelf(int[] nums) {
        int[] res = new int[nums.length];
        Arrays.fill(res, 1);
        int prefix = 1, suffix = 1;
        for (int i = 0; i < nums.length; i++) {
            res[i] *= prefix;
            prefix *= nums[i];
            res[nums.length - 1 - i] *= suffix;
            suffix *= nums[nums.length - 1 - i];
        }
        return res;
    }
}

```

617. 合并二叉树 Easy

给定两个二叉树，想象当你将它们中的一个覆盖到另一个上时，两个二叉树的一些节点便会重叠。

你需要将他们合并为一个新的二叉树。合并的规则是如果两个节点重叠，那么将他们的值相加作为节点合并后的数值，否则不为 NULL 的节点将直接作为新二叉树的节点。

示例 1:

输入：

Tree 1	Tree 2
1 / \br/> 3 2	2 / \br/> 1 3 \ \br/> 4 7

输出：

合并后的树：

```
    3
   / \
  4   5
 / \   \
5   4   7
```

注意: 合并必须从两个树的根节点开始。

```
class Solution {
    public TreeNode mergeTrees(TreeNode root1, TreeNode root2) {
        if (root1 == null)
            return root2;
        if (root2 == null)
            return root1;
        root1.val = root1.val + root2.val;
        root1.left = mergeTrees(root1.left, root2.left);
        root1.right = mergeTrees(root1.right, root2.right);
        return root1;
    }
}
```

```
class Solution {
    public TreeNode mergeTrees(TreeNode root1, TreeNode root2) {
        if (root1 == null)
            return root2;
        if (root2 == null)
            return root1;
        TreeNode root = new TreeNode(root1.val + root2.val);
        root.left = mergeTrees(root1.left, root2.left);
        root.right = mergeTrees(root1.right, root2.right);
        return root;
    }
}
```

279. 完全平方数 Medium

给定正整数 n ，找到若干个完全平方数（比如 `1, 4, 9, 16, ...`）使得它们的和等于 n 。你需要让组成和的完全平方数的个数最少。

给你一个整数 `n`，返回和为 `n` 的完全平方数的 **最少数量**。

完全平方数 是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的积。例如，`1`、`4`、`9` 和 `16` 都是完全平方数，而 `3` 和 `11` 不是。

示例 1:

```
输入: n = 12
输出: 3
解释: 12 = 4 + 4 + 4
```

示例 2:

```
输入: n = 13
输出: 2
解释: 13 = 4 + 9
```

提示:

- $1 \leq n \leq 104$

```
class Solution {
    public int numSquares(int n) {
        int[] dp = new int[n + 1];
        Arrays.fill(dp, n);
        dp[0] = 0;
        for (int i = 1; i * i <= n; i++) {
            for (int j = i * i; j <= n; j++) {
                dp[j] = Math.min(dp[j], dp[j - i * i] + 1);
            }
        }
        return dp[n];
    }
}
```

448. 找到所有数组中消失的数字 Easy

给你一个含 n 个整数的数组 nums ，其中 $\text{nums}[i]$ 在区间 $[1, n]$ 内。请你找出所有在 $[1, n]$ 范围内但没有出现在 nums 中的数字，并以数组的形式返回结果。

示例 1:

```
输入: nums = [4,3,2,7,8,2,3,1]
输出: [5,6]
```

示例 2:

```
输入: nums = [1,1]
输出: [2]
```

提示:

- $n == \text{nums.length}$
- $1 \leq n \leq 105$
- $1 \leq \text{nums}[i] \leq n$

进阶: 你能在不使用额外空间且时间复杂度为 $O(n)$ 的情况下解决这个问题吗? 你可以假定返回的数组不算在额外空间内。

```
class Solution {
    public List<Integer> findDisappearedNumbers(int[] nums) {
        List<Integer> res = new ArrayList<>();
        for (int num : nums) {
            int i = (num - 1) % nums.length;
            nums[i] += nums.length;
        }
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] <= nums.length)
                res.add(i + 1);
        }
        return res;
    }
}
```

309. 最佳买卖股票时机含冷冻期 Medium

给定一个整数数组，其中第 i 个元素代表了第 i 天的股票价格。

设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票）：

- 你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。
- 卖出股票后，你无法在第二天买入股票(即冷冻期为 1 天)。

示例:

```
输入: [1,2,3,0,2]
输出: 3
解释: 对应的交易状态为: [买入, 卖出, 冷冻期, 买入, 卖出]
```

```

class Solution {
    public int maxProfit(int[] prices) {
        int[][] dp = new int[prices.length][3];
        dp[0][0] = -prices[0];
        for (int i = 1; i < dp.length; i++) {
            dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][2] - prices[i]);
            dp[i][1] = dp[i - 1][0] + prices[i];
            dp[i][2] = Math.max(dp[i - 1][1], dp[i - 1][2]);
        }
        return Math.max(dp[prices.length - 1][1], dp[prices.length - 1][2]);
    }
}

```

`dp[i][0]` 第 i 天结束后持有股票的状态

`dp[i][1]` 第 i 天结束后不持有股票、冷冻状态

`dp[i][2]` 第 i 天结束后不持有股票、非冷冻状态

这道题目的关键在于，画状态转移图。那么该问题就会迎刃而解了。

49. 字母异位词分组 Medium

给你一个字符串数组，请你将 **字母异位词** 组合在一起。可以按任意顺序返回结果列表。

字母异位词 是由重新排列源单词的字母得到的一个新单词，所有源单词中的字母都恰好只用一次。

示例 1:

```

输入: strs = ["eat", "tea", "tan", "ate", "nat", "bat"]
输出: [["bat"], ["nat", "tan"], ["ate", "eat", "tea"]]

```

示例 2:

```

输入: strs = []
输出: [[]]

```

示例 3:

```

输入: strs = ["a"]
输出: [["a"]]

```

提示:

- $1 \leq \text{strs.length} \leq 104$
- $0 \leq \text{strs}[i].length \leq 100$

- `strs[i]` 仅包含小写字母

`strSorted` 函数可以使用自带的 api 处理, `String key = new String(str.toCharArray());`

```
class Solution {

    public List<List<String>> groupAnagrams(String[] strs) {
        HashMap<String, List<String>> hashMap = new HashMap<>();
        for (String str : strs) {
            char[] cs = str.toCharArray();
            Arrays.sort(cs);
            String key = new String(cs);
            if (!hashMap.containsKey(key))
                hashMap.put(key, new ArrayList<>());
            hashMap.get(key).add(str);
        }
        return new ArrayList<>(hashMap.values());
    }
}
```

416. 分割等和子集 Hard

给你一个 只包含正整数 的 非空 数组 `nums`。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

示例 1:

```
输入: nums = [1,5,11,5]
输出: true
解释: 数组可以分割成 [1, 5, 5] 和 [11] 。
```

示例 2:

```
输入: nums = [1,2,3,5]
输出: false
解释: 数组不能分割成两个元素和相等的子集。
```

提示:

- `1 <= nums.length <= 200`
- `1 <= nums[i] <= 100`

```
class Solution {
```

```

public boolean canPartition(int[] nums) {
    int sum = Arrays.stream(nums).sum();
    if ((sum & 1) != 0)
        return false;
    int k = sum >> 1;
    boolean[][] dp = new boolean[nums.length + 1][k + 1];
    dp[0][0] = true;
    for (int i = 1; i < dp.length; i++) {
        for (int j = 1; j <= k; j++) {
            if (j >= nums[i - 1])
                dp[i][j] = dp[i - 1][j - nums[i - 1]] || dp[i - 1][j];
            else
                dp[i][j] = dp[i - 1][j];
        }
    }
    return dp[nums.length][k];
}

```

```

class Solution {
    public boolean canPartition(int[] nums) {
        int sum = Arrays.stream(nums).sum();
        if ((sum & 1) != 0)
            return false;
        int k = sum >> 1;
        boolean[] dp = new boolean[k + 1];
        dp[0] = true;
        for (int num : nums) {
            for (int i = k; i >= num; i--) {
                dp[i] |= dp[i - num];
            }
        }
        return dp[k];
    }
}

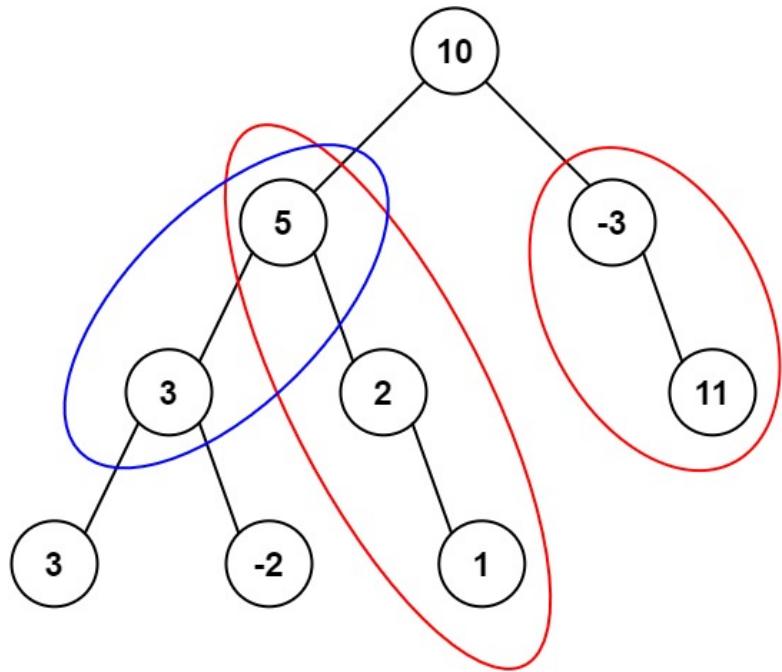
```

437. 路径总和 III Medium

给定一个二叉树的根节点 `root`，和一个整数 `targetSum`，求该二叉树里节点值之和等于 `targetSum` 的路径的数目。

路径 不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）。

示例 1：



输入: `root = [10,5,-3,3,2,null,11,3,-2,null,1], targetSum = 8`

输出: 3

解释: 和等于 8 的路径有 3 条, 如图所示。

示例 2:

输入: `root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22`

输出: 3

提示:

- 二叉树的节点个数的范围是 `[0,1000]`
- `-109 <= Node.val <= 109`
- `-1000 <= targetSum <= 1000`

```

class Solution {
    private int count = 0;

    public int pathSum(TreeNode root, int targetSum) {
        dfs(root, 0, targetSum);
        return count;
    }

    private void dfs(TreeNode root, int sum, int targetSum) {
        if (root == null)
            return;
        if (sum + root.val == targetSum)
            count++;
        dfs(root.left, root.val + sum, targetSum);
        dfs(root.right, root.val + sum, targetSum);
    }
}

```

```

        dfs(root.left, 0, targetSum);
        dfs(root.right, root.val + sum, targetSum);
        dfs(root.right, 0, targetSum);
    }
}

```

这个解法是错误的。下面一组测试用例不能通过，但是不知道是哪里错了。

```
[1,null,2,null,3,null,4,null,5]
3
```

```

class Solution {
    // key:前缀和, value:节点的前缀和为当前key值的数量, 即路径数量。
    // 一个节点的前缀和就是该节点到根之间的路径和
    private HashMap<Integer, Integer> prefixSum = new HashMap<>();

    public int pathSum(TreeNode root, int targetSum) {
        //前缀和为0的路径数量是1
        prefixSum.put(0, 1);
        return dfs(root, 0, targetSum);
    }

    private int dfs(TreeNode root, int sum, int targetSum) {
        if (root == null)
            return 0;
        sum = sum + root.val;

        // 下面两步操作不能更换顺序, 为什么?
        // 下面的这一步可以写成 (targetSum - sum) 吗? 不可以 ✘
        int res = prefixSum.getOrDefault(sum - targetSum, 0);
        prefixSum.put(sum, prefixSum.getOrDefault(sum, 0) + 1);

        res += dfs(root.left, sum, targetSum);
        res += dfs(root.right, sum, targetSum);
        //为什么这里要回溯?
        prefixSum.put(sum, prefixSum.get(sum) - 1);
        return res;
    }
}

```

更改顺序有一组测试用例无法通过。[1] 0

经过测试，只要 `targetSum` 不为0，其实这两步的先后顺序都可以通过。

438. 找到字符串中所有字母异位词 Medium

给定两个字符串 `s` 和 `p`，找到 `s` 中所有 `p` 的 异位词 的子串，返回这些子串的起始索引。不考虑答案输出的顺序。

异位词 指字母相同，但排列不同的字符串。

示例 1：

```
输入: s = "cbaebabacd", p = "abc"
输出: [0,6]
解释:
起始索引等于 0 的子串是 "cba"，它是 "abc" 的异位词。
起始索引等于 6 的子串是 "bac"，它是 "abc" 的异位词。
```

示例 2：

```
输入: s = "abab", p = "ab"
输出: [0,1,2]
解释:
起始索引等于 0 的子串是 "ab"，它是 "ab" 的异位词。
起始索引等于 1 的子串是 "ba"，它是 "ab" 的异位词。
起始索引等于 2 的子串是 "ab"，它是 "ab" 的异位词。
```

提示:

- $1 \leq s.length, p.length \leq 3 * 10^4$
- `s` 和 `p` 仅包含小写字母

```
class Solution {
    public List<Integer> findAnagrams(String s, String p) {
        ArrayList<Integer> res = new ArrayList<>();
        char[] sArr = s.toCharArray();
        char[] pArr = p.toCharArray();
        int[] sCnt = new int[26];
        int[] pCnt = new int[26];
        int window = p.length();
        for (char ch : pArr) {
            pCnt[ch - 'a']++;
        }
        for (int i = 0; i < sArr.length; i++) {
            sCnt[sArr[i] - 'a']++;
            if (i >= window)
                sCnt[sArr[i - window] - 'a']--;
            if (Arrays.equals(sCnt, pCnt))
                res.add(i - window + 1);
        }
        return res;
    }
}
```

```
    }  
}
```

这个的速度在8~10ms之间，还可以继续优化。

上下两个好像差不多。

```
class Solution {  
  
    public List<Integer> findAnagrams(String s, String p) {  
        List<Integer> res = new LinkedList<>();  
        if (s.length() < p.length())  
            return res;  
        int[] sCount = new int[26];  
        int[] pCount = new int[26];  
  
        char[] sArr = s.toCharArray();  
        char[] pArr = p.toCharArray();  
        for (int i = 0; i < pArr.length; i++) {  
            pCount[pArr[i] - 'a']++;  
        }  
        int left = 0, right = 0;  
        while (right < s.length()) {  
            int curRight = sArr[right] - 'a';  
            sCount[curRight]++;  
            while (sCount[curRight] > pCount[curRight]) {  
                sCount[sArr[left++]- 'a']--;  
            }  
            if (right - left + 1 == p.length())  
                res.add(left);  
            right++;  
        }  
        return res;  
    }  
}
```

这个解法还能再快几秒，大概在5ms，但不是很好想。

```
class Solution {  
  
    public List<Integer> findAnagrams(String s, String p) {  
        String pStr = transform(p, 0, p.length() - 1);  
        HashMap<String, List<Integer>> hashMap = new HashMap<>();  
        for (int i = 0; i <= s.length() - p.length(); i++) {  
            String sStr = transform(s, i, i + p.length() - 1);  
            if (!hashMap.containsKey(sStr))  
                hashMap.put(sStr, new ArrayList<>());
```

```

        hashMap.get(sStr).add(i);
    }
    return hashMap.getOrDefault(pStr, new ArrayList<>());
}

private String transform(String str, int start, int end) {
    String newStr = str.substring(start, end + 1);
    char[] arr = newStr.toCharArray();
    int[] count = new int[26];
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i <= end - start; i++) {
        count[arr[i] - 'a']++;
    }
    for (int i = 0; i < count.length; i++) {
        if (count[i] != 0)
            sb.append(count[i]).append((char) (i + 'a'));
    }
    return sb.toString();
}
}

```

暴力解法，时间在700ms左右。

312. 戳气球 Hard

有 n 个气球，编号为 0 到 $n - 1$ ，每个气球上都标有一个数字，这些数字存在数组 nums 中。

现在要求你戳破所有的气球。戳破第 i 个气球，你可以获得 $\text{nums}[i - 1] * \text{nums}[i] * \text{nums}[i + 1]$ 枚硬币。这里的 $i - 1$ 和 $i + 1$ 代表和 i 相邻的两个气球的序号。如果 $i - 1$ 或 $i + 1$ 超出了数组的边界，那么就当它是一个数字为 1 的气球。

求所能获得硬币的最大数量。

示例 1：

```

输入: nums = [3,1,5,8]
输出: 167
解释:
nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []
coins = 3*1*5      +      3*5*8      +      1*3*8      +      1*8*1 = 167

```

示例 2：

```

输入: nums = [1,5]
输出: 10

```

提示:

- `n == nums.length`
- `1 <= n <= 500`
- `0 <= nums[i] <= 100`

```
class Solution {
    public int maxCoins(int[] nums) {
        int n = nums.length;
        int[][] rec = new int[n + 2][n + 2];
        int[] val = new int[n + 2];
        val[0] = val[n + 1] = 1;
        for (int i = 1; i <= n; i++) {
            val[i] = nums[i - 1];
        }
        for (int i = n - 1; i >= 0; i--) {
            for (int j = i + 2; j <= n + 1; j++) {
                for (int k = i + 1; k < j; k++) {
                    int sum = val[i] * val[k] * val[j];
                    sum += rec[i][k] + rec[k][j];
                    rec[i][j] = Math.max(rec[i][j], sum);
                }
            }
        }
        return rec[0][n + 1];
    }
}
```

直接摘录官方解答。

令 $dp[i][j]$ 表示填满开区间 (i, j) 能得到的最多硬币数，那么边界条件是 $i \geq j - 1$ ，此时有 $dp[i][j] = 0$ 。

可以写出状态转移方程：

$$dp[i][j] = \begin{cases} \max_{k=i+1}^{j-1} val[i] \times val[k] \times val[j] + dp[i][k] + dp[k][j], & i < j - 1 \\ 0, & i \geq j - 1 \end{cases}$$

最终答案即为 $dp[0][n + 1]$ 。实现时要注意到动态规划的次序。

75. 颜色分类 Medium

给定一个包含红色、白色和蓝色，一共 n 个元素的数组，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

此题中，我们使用整数 `0`、`1` 和 `2` 分别表示红色、白色和蓝色。

示例 1：

```
输入: nums = [2,0,2,1,1,0]
输出: [0,0,1,1,2,2]
```

示例 2：

```
输入: nums = [2,0,1]
输出: [0,1,2]
```

示例 3：

```
输入: nums = [0]
输出: [0]
```

示例 4：

```
输入: nums = [1]
输出: [1]
```

提示：

- $n == \text{nums.length}$
- $1 \leq n \leq 300$
- $\text{nums}[i]$ 为 `0`、`1` 或 `2`

进阶：

- 你可以不使用代码库中的排序函数来解决这道题吗？
- 你能想出一个仅使用常数空间的一趟扫描算法吗？

```
class Solution {
    public void sortColors(int[] nums) {
        int left = 0, right = nums.length - 1;
        int index = 0;
        while (index <= right) {
            if (nums[index] == 0)
                swap(nums, index++, left++);
            else if (nums[index] == 2)
                swap(nums, index, right--);
            else
                index++;
        }
    }

    private void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}
```

```

        index++;
    }

}

private void swap(int[] nums, int i, int j) {
    int temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
}
}

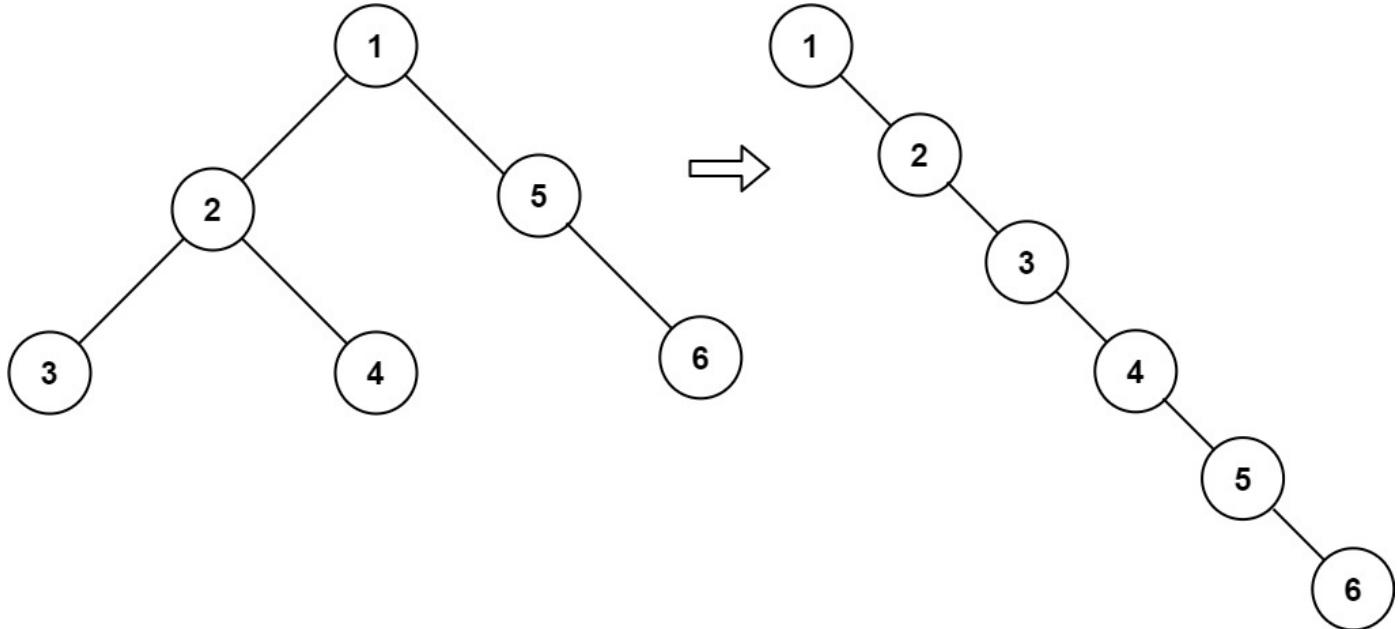
```

114. 二叉树展开为链表 Medium

给你二叉树的根结点 `root`，请你将它展开为一个单链表：

- 展开后的单链表应该同样使用 `treeNode`，其中 `right` 子指针指向链表中下一个结点，而左子指针始终为 `null`。
- 展开后的单链表应该与二叉树 [先序遍历](#) 顺序相同。

示例 1：



输入: `root = [1,2,5,3,4,null,6]`

输出: `[1,null,2,null,3,null,4,null,5,null,6]`

示例 2：

输入: `root = []`

输出: `[]`

示例 3:

```
输入: root = [0]
输出: [0]
```

提示:

- 树中结点数在范围 $[0, 2000]$ 内
- $-100 \leq \text{Node.val} \leq 100$

进阶：你可以使用原地算法 ($O(1)$ 额外空间) 展开这棵树吗？

```
class Solution {
    public void flatten(TreeNode root) {
        if (root == null)
            return;
        TreeNode successor = root.left;
        if (successor != null) {
            TreeNode predecessor = successor;
            while (predecessor.right != null)
                predecessor = predecessor.right;
            predecessor.right = root.right;
            root.left = null;
            root.right = successor;
        }
        flatten(root.right);
    }
}
```

```
class Solution {
    public void flatten(TreeNode root) {
        while (root != null) {
            TreeNode predecessor = root.left;
            TreeNode successor = root.left;
            if (predecessor != null) {
                while (predecessor.right != null)
                    predecessor = predecessor.right;
                predecessor.right = root.right;
                // 下面这两个步骤必须要在这个括号内完成。
                root.left = null;
                root.right = successor;
            }
            root = root.right;
        }
    }
}
```

```

class Solution {
    public void flatten(TreeNode root) {
        while (root != null) {
            TreeNode left = root.left;
            TreeNode right = root.right;
            if (left != null) {
                TreeNode predecessor = left;
                while (predecessor.right != null)
                    predecessor = predecessor.right;
                root.left = null;
                root.right = left;
                predecessor.right = right;
            }
            root = root.right;
        }
    }
}

```

递归和迭代两种方法，后面

139. 单词拆分 Medium

给你一个字符串 `s` 和一个字符串列表 `wordDict` 作为字典，判定 `s` 是否可以由空格拆分为一个或多个在字典中出现的单词。

说明：拆分时可以重复使用字典中的单词。

示例 1：

```

输入: s = "leetcode", wordDict = ["leet", "code"]
输出: true
解释: 返回 true 因为 "leetcode" 可以被拆分成 "leet code"。

```

示例 2：

```

输入: s = "applepenapple", wordDict = ["apple", "pen"]
输出: true
解释: 返回 true 因为 "applepenapple" 可以被拆分成 "apple pen apple"。
注意你可以重复使用字典中的单词。

```

示例 3：

```

输入: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
输出: false

```

提示:

- $1 \leq s.length \leq 300$
- $1 \leq wordDict.length \leq 1000$
- $1 \leq wordDict[i].length \leq 20$
- s 和 $wordDict[i]$ 仅有小写英文字母组成
- $wordDict$ 中的所有字符串互不相同

```
class Solution {  
    public boolean wordBreak(String s, List<String> wordDict) {  
        boolean[] dp = new boolean[s.length() + 1];  
        dp[0] = true;  
        for (int i = 1; i <= s.length(); i++) {  
            for (String word : wordDict) {  
                if (i >= word.length() && word.equals(s.substring(i - word.length(),  
i)))  
                    dp[i] |= dp[i - word.length()];  
            }  
        }  
        return dp[s.length()];  
    }  
}
```

10. 正则表达式匹配 Hard

给你一个字符串 s 和一个字符规律 p ，请你来实现一个支持 $'.'$ 和 $'*''$ 的正则表达式匹配。

- $'.'$ 匹配任意单个字符
- $'*'$ 匹配零个或多个前面的那个元素

所谓匹配，是要涵盖 整个 字符串 s 的，而不是部分字符串。

示例 1:

```
输入: s = "aa" p = "a"  
输出: false  
解释: "a" 无法匹配 "aa" 整个字符串。
```

示例 2:

```
输入: s = "aa" p = "a*"  
输出: true  
解释: 因为 '*' 代表可以匹配零个或多个前面的那个元素，在这里前面的元素就是 'a'。因此，字符串 "aa" 可被视为 'a' 重复了一次。
```

示例 3:

```
输入: s = "ab" p = ".*"
输出: true
解释: ".*" 表示可匹配零个或多个（'*'）任意字符（'.'）。
```

示例 4:

```
输入: s = "aab" p = "c*a*b"
输出: true
解释: 因为 '*' 表示零个或多个，这里 'c' 为 0 个，'a' 被重复一次。因此可以匹配字符串 "aab"。
```

示例 5:

```
输入: s = "mississippi" p = "mis*is*p*."
输出: false
```

提示:

- $1 \leq s.length \leq 20$
- $1 \leq p.length \leq 30$
- s 可能为空，且只包含从 $a-z$ 的小写字母。
- p 可能为空，且只包含从 $a-z$ 的小写字母，以及字符 $.$ 和 $*$ 。
- 保证每次出现字符 $*$ 时，前面都匹配到有效的字符

```
class Solution {
    public boolean isMatch(String s, String p) {
        int m = s.length();
        int n = p.length();

        boolean[][] f = new boolean[m + 1][n + 1];
        f[0][0] = true;
        for (int i = 0; i <= m; ++i) {
            for (int j = 1; j <= n; ++j) {
                if (p.charAt(j - 1) == '*') {
                    f[i][j] = f[i][j - 2];
                    if (matches(s, p, i, j - 1)) {
                        f[i][j] = f[i][j] || f[i - 1][j];
                    }
                } else {
                    if (matches(s, p, i, j)) {
                        f[i][j] = f[i - 1][j - 1];
                    }
                }
            }
        }
    }
}
```

```

        return f[m][n];
    }

    public boolean matches(String s, String p, int i, int j) {
        if (i == 0) {
            return false;
        }
        if (p.charAt(j - 1) == '.') {
            return true;
        }
        return s.charAt(i - 1) == p.charAt(j - 1);
    }
}

```

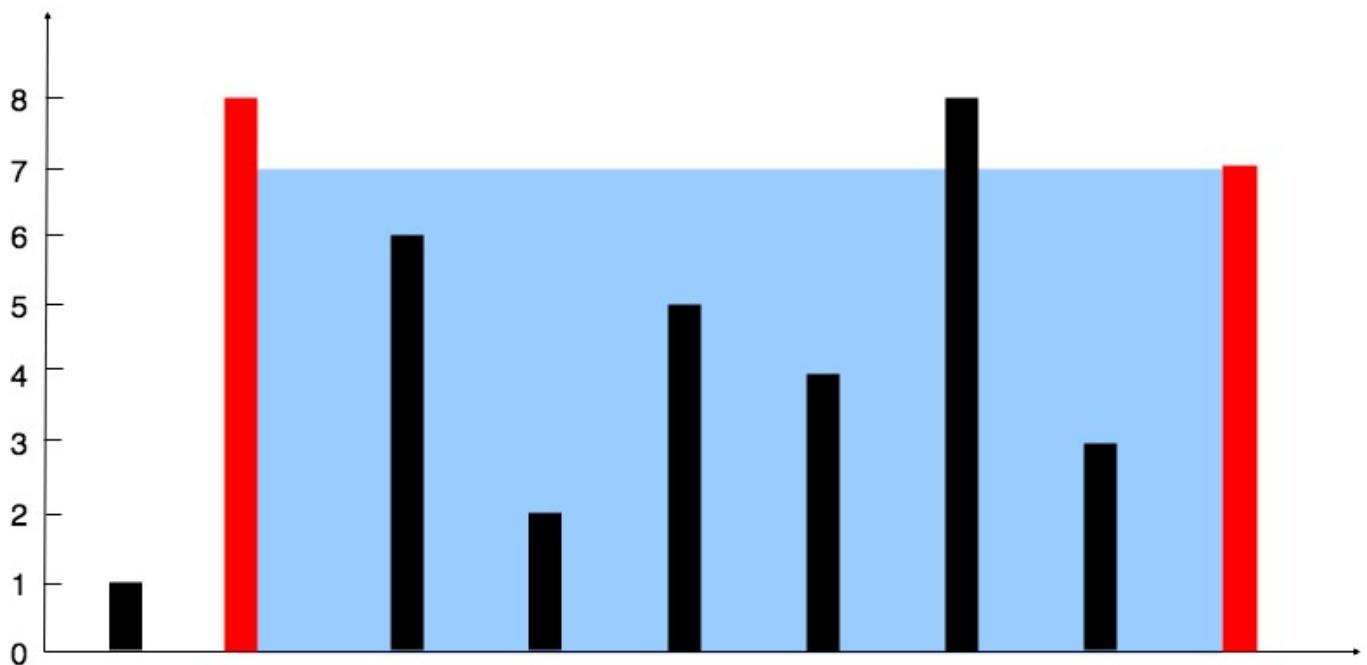
直接摘录答案。

11. 盛最多水的容器 Medium

给你 n 个非负整数 a_1, a_2, \dots, a_n ，每个数代表坐标中的一个点 (i, a_i) 。在坐标内画 n 条垂直线，垂直线 i 的两个端点分别为 (i, a_i) 和 $(i, 0)$ 。找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。

说明：你不能倾斜容器。

示例 1：



输入： [1,8,6,2,5,4,8,3,7]

输出： 49

解释：图中垂直线代表输入数组 [1,8,6,2,5,4,8,3,7]。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。

示例 2:

```
输入: height = [1,1]
输出: 1
```

示例 3:

```
输入: height = [4,3,2,1,4]
输出: 16
```

示例 4:

```
输入: height = [1,2,1]
输出: 2
```

提示:

- `n == height.length`
- `2 <= n <= 105`
- `0 <= height[i] <= 104`

```
class Solution {
    public int maxArea(int[] height) {
        int left = 0, right = height.length - 1;
        int area = 0;
        while (left <= right) {
            int length = right - left;
            int width = Math.min(height[left], height[right]);
            area = Math.max(area, length * width);
            if (height[left] < height[right])
                left++;
            else
                right--;
        }
        return area;
    }
}
```

347. 前 K 个高频元素

给你一个整数数组 `nums` 和一个整数 `k`，请你返回其中出现频率前 `k` 高的元素。你可以按任意顺序 返回答案。

示例 1：

```
输入: nums = [1,1,1,2,2,3], k = 2
输出: [1,2]
```

示例 2：

```
输入: nums = [1], k = 1
输出: [1]
```

提示：

- $1 \leq \text{nums.length} \leq 105$
- k 的取值范围是 $[1, \text{数组中不相同的元素的个数}]$
- 题目数据保证答案唯一，换句话说，数组中前 k 个高频元素的集合是唯一的

进阶：你所设计算法的时间复杂度 必须 优于 $O(n \log n)$ ，其中 n 是数组大小。

```
class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        HashMap<Integer, Integer> hashMap = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            hashMap.put(nums[i], hashMap.getOrDefault(nums[i], 0) + 1);
        }
        PriorityQueue<Integer> pq = new PriorityQueue<>(new Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
                return hashMap.get(o1) - hashMap.get(o2);
            }
        });
        for (Integer key : hashMap.keySet()) {
            if (pq.size() >= k) {
                if (hashMap.get(pq.peek()) < hashMap.get(key))
                    pq.poll();
                else
                    continue;
            }
            pq.offer(key);
        }
        int[] res = new int[k];
        int count = 0;
        while (!pq.isEmpty()) {
            res[count++] = pq.poll();
        }
    }
}
```

```
    return res;
}
}
```

581. 最短无序连续子数组 Medium

给你一个整数数组 `nums`，你需要找出一个 **连续子数组**，如果对这个子数组进行升序排序，那么整个数组都会变为升序排序。

请你找出符合题意的 **最短 子数组**，并输出它的长度。

示例 1：

输入: `nums = [2,6,4,8,10,9,15]`

输出: 5

解释: 你只需要对 `[6, 4, 8, 10, 9]` 进行升序排序，那么整个表都会变为升序排序。

示例 2：

输入: `nums = [1,2,3,4]`

输出: 0

示例 3：

输入: `nums = [1]`

输出: 0

提示：

- `1 <= nums.length <= 104`
- `-105 <= nums[i] <= 105`

进阶：你可以设计一个时间复杂度为 `O(n)` 的解决方案吗？

```
class Solution {
    public int findUnsortedSubarray(int[] nums) {
        int left = 0, right = nums.length - 1;
        while (left < right && nums[left] <= nums[left + 1]) {
            left++;
        }
        while (left < right && nums[right] >= nums[right - 1]) {
            right--;
        }
        if (left == right)
```

```

        return 0;
    int min = nums[left], max = nums[right];
    for (int i = left; i <= right; i++) {
        min = Math.min(min, nums[i]);
        max = Math.max(max, nums[i]);
    }
    while (left >= 0 && nums[left] > min) {
        left--;
    }
    while (right <= nums.length - 1 && nums[right] < max) {
        right++;
    }
    return right - left - 1;
}
}

```

```

class Solution {
    public int findUnsortedSubarray(int[] nums) {
        int n = nums.length;
        int max = Integer.MIN_VALUE, right = -1;
        int min = Integer.MAX_VALUE, left = -1;
        for (int i = 0; i < n; i++) {
            if (max > nums[i])
                right = i;
            else
                max = nums[i];
            if (min < nums[n - i - 1])
                left = n - i - 1;
            else
                min = nums[n - i - 1];
        }
        return right == -1 ? 0 : right - left + 1;
    }
}

```

621. 任务调度器 Medium

给你一个用字符数组 `tasks` 表示的 CPU 需要执行的任务列表。其中每个字母表示一种不同种类的任务。任务可以以任意顺序执行，并且每个任务都可以在 1 个单位时间内执行完。在任何一个单位时间，CPU 可以完成一个任务，或者处于待命状态。

然而，两个 **相同种类** 的任务之间必须有长度为整数 `n` 的冷却时间，因此至少有连续 `n` 个单位时间内 CPU 在执行不同的任务，或者在待命状态。

你需要计算完成所有任务所需要的 **最短时间**。

示例 1:

输入: tasks = ["A", "A", "A", "B", "B", "B"], n = 2

输出: 8

解释: A -> B -> (待命) -> A -> B -> (待命) -> A -> B

在本示例中, 两个相同类型任务之间必须间隔长度为 n = 2 的冷却时间, 而执行一个任务只需要一个单位时间, 所以中间出现了 (待命) 状态。

示例 2:

输入: tasks = ["A", "A", "A", "B", "B", "B"], n = 0

输出: 6

解释: 在这种情况下, 任何大小为 6 的排列都可以满足要求, 因为 n = 0

["A", "A", "A", "B", "B", "B"]

["A", "B", "A", "B", "A", "B"]

["B", "B", "B", "A", "A", "A"]

...

诸如此类

示例 3:

输入: tasks = ["A", "A", "A", "A", "A", "A", "B", "C", "D", "E", "F", "G"], n = 2

输出: 16

解释: 一种可能的解决方案是:

A -> B -> C -> A -> D -> E -> A -> F -> G -> A -> (待命) -> (待命) -> A -> (待命) -> (待命) -> A

提示:

- `1 <= task.length <= 104`
- `tasks[i]` 是大写英文字母
- `n` 的取值范围为 `[0, 100]`

```
/*
 * 解题思路:
 * 1、将任务按类型分组, 正好A-Z用一个int[26]保存任务类型个数
 * 2、对数组进行排序, 优先排列个数 (count) 最大的任务,
 *    如题得到的时间至少为 retCount = (count-1) * (n+1) + 1 ==> A->X->X->A->X->X->A (X
 * 为其他任务或者待命)
 * 3、再排序下一个任务, 如果下一个任务B个数和最大任务数一致,
 *    则retCount++ ==> A->B->X->A->B->X->A->B
 * 4、如果空位都插满之后还有任务, 那就随便在这些间隔里面插入就可以, 因为间隔长度肯定会大于n, 在这种
 * 情况下就是任务的总数是最小所需时间
 *
 * @param tasks
 * @param n
```

```

* @return
*/
public int leastInterval(char[] tasks, int n) {
    if (tasks.length <= 1 || n < 1) return tasks.length;
    //步骤1
    int[] counts = new int[26];
    for (int i = 0; i < tasks.length; i++) {
        counts[tasks[i] - 'A']++;
    }
    //步骤2
    Arrays.sort(counts);
    int maxCount = counts[25];
    int retCount = (maxCount - 1) * (n + 1) + 1;
    int i = 24;
    //步骤3
    while (i >= 0 && counts[i] == maxCount) {
        retCount++;
        i--;
    }
    //步骤4
    return Math.max(retCount, tasks.length);
}

```

答案摘自评论。

399. 除法求值 Medium

给你一个变量对数组 `equations` 和一个实数值数组 `values` 作为已知条件，其中 `equations[i] = [Ai, Bi]` 和 `values[i]` 共同表示等式 `Ai / Bi = values[i]`。每个 `Ai` 或 `Bi` 是一个表示单个变量的字符串。

另有一些以数组 `queries` 表示的问题，其中 `queries[j] = [Cj, Dj]` 表示第 `j` 个问题，请你根据已知条件找出 `Cj / Dj = ?` 的结果作为答案。

返回 **所有问题的答案**。如果存在某个无法确定的答案，则用 `-1.0` 替代这个答案。如果问题中出现了给定的已知条件中没有出现的字符串，也需要用 `-1.0` 替代这个答案。

注意：输入总是有效的。你可以假设除法运算中不会出现除数为 0 的情况，且不存在任何矛盾的结果。

示例 1：

```

输入: equations = [["a","b"],["b","c"]], values = [2.0,3.0], queries = [["a","c"], ["b","a"],["a","e"],["a","a"],["x","x"]]
输出: [6.00000,0.50000,-1.00000,1.00000,-1.00000]
解释:
条件: a / b = 2.0, b / c = 3.0
问题: a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x = ?
结果: [6.0, 0.5, -1.0, 1.0, -1.0 ]

```

示例 2:

```

输入: equations = [["a","b"],["b","c"],["bc","cd"]], values = [1.5,2.5,5.0], queries = [[["a","c"],["c","b"],["bc","cd"],["cd","bc"]]]
输出: [3.75000,0.40000,5.00000,0.20000]

```

示例 3:

```

输入: equations = [["a","b"]], values = [0.5], queries = [[["a","b"],["b","a"],["a","c"],["x","y"]]]
输出: [0.50000,2.00000,-1.00000,-1.00000]

```

提示:

- `1 <= equations.length <= 20`
- `equations[i].length == 2`
- `1 <= Ai.length, Bi.length <= 5`
- `values.length == equations.length`
- `0.0 < values[i] <= 20.0`
- `1 <= queries.length <= 20`
- `queries[i].length == 2`
- `1 <= Cj.length, Dj.length <= 5`
- `Ai, Bi, Cj, Dj 由小写英文字母与数字组成`

方法一：广度优先搜索

我们可以将整个问题建模成一张图：给定图中的一些点（变量），以及某些边的权值（两个变量的比值），试对任意两点（两个变量）求出其路径长（两个变量的比值）。

因此，我们首先需要遍历 `equations` 数组，找出其中所有不同的字符串，并通过哈希表将每个不同的字符串映射成整数。

在构建完图之后，对于任何一个查询，就可以从起点出发，通过广度优先搜索的方式，不断更新起点与当前点之间的路径长度，直到搜索到终点为止。

```

class Solution {
    public double[] calcEquation(List<List<String>> equations, double[] values,
List<List<String>> queries) {

```

```

int nvars = 0;
Map<String, Integer> variables = new HashMap<String, Integer>();

int n = equations.size();
for (int i = 0; i < n; i++) {
    if (!variables.containsKey(equations.get(i).get(0))) {
        variables.put(equations.get(i).get(0), nvars++);
    }
    if (!variables.containsKey(equations.get(i).get(1))) {
        variables.put(equations.get(i).get(1), nvars++);
    }
}

// 对于每个点，存储其直接连接到的所有点及对应的权值
List<Pair>[] edges = new List[nvars];
for (int i = 0; i < nvars; i++) {
    edges[i] = new ArrayList<Pair>();
}
for (int i = 0; i < n; i++) {
    int va = variables.get(equations.get(i).get(0)), vb =
variables.get(equations.get(i).get(1));
    edges[va].add(new Pair(vb, values[i]));
    edges[vb].add(new Pair(va, 1.0 / values[i]));
}

int queriesCount = queries.size();
double[] ret = new double[queriesCount];
for (int i = 0; i < queriesCount; i++) {
    List<String> query = queries.get(i);
    double result = -1.0;
    if (variables.containsKey(query.get(0)) &&
variables.containsKey(query.get(1))) {
        int ia = variables.get(query.get(0)), ib = variables.get(query.get(1));
        if (ia == ib) {
            result = 1.0;
        } else {
            Queue<Integer> points = new LinkedList<Integer>();
            points.offer(ia);
            double[] ratios = new double[nvars];
            Arrays.fill(ratios, -1.0);
            ratios[ia] = 1.0;

            while (!points.isEmpty() && ratios[ib] < 0) {
                int x = points.poll();
                for (Pair pair : edges[x]) {
                    int y = pair.index;
                    double val = pair.value;
                    if (ratios[y] < 0) {
                        ratios[y] = ratios[x] * val;
                    }
                }
            }
        }
    }
}

```

```
        points.offer(y);
    }
}
result = ratios[ib];
}
ret[i] = result;
}
return ret;
}
}

class Pair {
    int index;
    double value;

    Pair(int index, double value) {
        this.index = index;
        this.value = value;
    }
}
```

方法三：带权并查集

我们还可以考虑以并查集的方式存储节点之间的关系。设节点 x 的值（即对应变量的取值）为 $v[x]$ 。对于任意两点 x, y , 假设它们在并查集中具有共同的父亲 f , 且 $v[x]/v[f] = a, v[y]/v[f] = b$, 则 $v[x]/v[y] = a/b$ 。

在观察到这一点后, 就不难利用并查集的思想解决此题。对于每个节点 x 而言, 除了维护其父亲 $f[x]$ 之外, 还要维护其权值 w , 其中「权值」定义为节点 x 的取值与父亲 $f[x]$ 的取值之间的比值。换言之, 我们有

$$w[x] = \frac{v[x]}{v[f[x]]}$$

下面, 我们对并查集的两种操作的实现细节做出讨论。

当查询节点 x 父亲时, 如果 $f[x] \neq x$, 我们需要先找到 $f[x]$ 的父亲 $father$, 并将 $f[x]$ 更新为 $father$ 。此时, 我们有

$$\begin{aligned} w[x] &\leftarrow \frac{v[x]}{v[father]} \\ &= \frac{v[x]}{v[f[x]]} \cdot \frac{v[f[x]]}{v[father]} \\ &= w[i] \cdot w[f[x]] \end{aligned}$$

也就是说, 我们要将 $w[x]$ 更新为 $w[x] \cdot w[f[x]]$ 。

当合并两个节点 x, y 时, 我们首先找到两者的父亲 f_x, f_y , 并将 $f[f_x]$ 更新为 f_y 。此时, 我们有

$$\begin{aligned} w[f_x] &\leftarrow \frac{v[f_x]}{v[f_y]} \\ &= \frac{v[x]/w[x]}{v[y]/w[y]} \\ &= \frac{v[x]}{v[y]} \cdot \frac{w[y]}{w[x]} \end{aligned}$$

也就是说, 当在已有的图中添加一条方程式 $\frac{v[x]}{v[y]} = k$ 时, 需要将 $w[f_x]$ 更新为 $k \cdot \frac{w[y]}{w[x]}$ 。

```
class Solution {
    public double[] calcEquation(List<List<String>> equations, double[] values,
List<List<String>> queries) {
        int nvars = 0;
        Map<String, Integer> variables = new HashMap<String, Integer>();

        int n = equations.size();
        for (int i = 0; i < n; i++) {
            if (!variables.containsKey(equations.get(i).get(0))) {
                variables.put(equations.get(i).get(0), nvars++);
            }
            if (!variables.containsKey(equations.get(i).get(1))) {
                variables.put(equations.get(i).get(1), nvars++);
            }
        }
        int[] f = new int[nvars];
        double[] w = new double[nvars];
        Arrays.fill(w, 1.0);
```

```

        for (int i = 0; i < nvars; i++) {
            f[i] = i;
        }

        for (int i = 0; i < n; i++) {
            int va = variables.get(equations.get(i).get(0)), vb =
variables.get(equations.get(i).get(1));
            merge(f, w, va, vb, values[i]);
        }
        int queriesCount = queries.size();
        double[] ret = new double[queriesCount];
        for (int i = 0; i < queriesCount; i++) {
            List<String> query = queries.get(i);
            double result = -1.0;
            if (variables.containsKey(query.get(0)) &&
variables.containsKey(query.get(1))) {
                int ia = variables.get(query.get(0)), ib = variables.get(query.get(1));
                int fa = findf(f, w, ia), fb = findf(f, w, ib);
                if (fa == fb) {
                    result = w[ia] / w[ib];
                }
            }
            ret[i] = result;
        }
        return ret;
    }

    public void merge(int[] f, double[] w, int x, int y, double val) {
        int fx = findf(f, w, x);
        int fy = findf(f, w, y);
        f[fx] = fy;
        w[fx] = val * w[y] / w[x];
    }

    public int findf(int[] f, double[] w, int x) {
        if (f[x] != x) {
            int father = findf(f, w, f[x]);
            w[x] = w[x] * w[f[x]];
            f[x] = father;
        }
        return f[x];
    }
}

```

191. 位1的个数 Easy

编写一个函数，输入是一个无符号整数（以二进制串的形式），返回其二进制表达式中数位数为 '1' 的个数（也被称为[汉明重量](#)）。

提示：

- 请注意，在某些语言（如 Java）中，没有无符号整数类型。在这种情况下，输入和输出都将被指定为有符号整数类型，并且不应影响您的实现，因为无论整数是有符号的还是无符号的，其内部的二进制表示形式都是相同的。
- 在 Java 中，编译器使用[二进制补码](#)记法来表示有符号整数。因此，在上面的[示例 3](#) 中，输入表示有符号整数 `-3`。

示例 1：

输入：000000000000000000000000000000001011

输出：3

解释：输入的二进制串 000000000000000000000000000000001011 中，共有三位为 '1'。

示例 2：

输入：0000000000000000000000000000000010000000

输出：1

解释：输入的二进制串 0000000000000000000000000000000010000000 中，共有一位为 '1'。

示例 3：

输入：111111111111111111111111111111101

输出：31

解释：输入的二进制串 1111111111111111111111111111101 中，共有 31 位为 '1'。

提示：

- 输入必须是长度为 `32` 的二进制串。

进阶：

- 如果多次调用这个函数，你将如何优化你的算法？

```

public class Solution {
    // you need to treat n as an unsigned value
    public int hammingWeight(int n) {
        int count = 0;
        for (int i = 0; i < 32; i++) {
            if ((n & 1 << i) != 0)
                count++;
        }
        return count;
    }
}

```

`n & (n-1)`，其运算结果恰为把 `n` 的二进制位中的最低位的 `1` 变为 `0` 之后的结果。

```

public class Solution {
    // you need to treat n as an unsigned value
    public int hammingWeight(int n) {
        int count = 0;
        while (n != 0) {
            n &= (n - 1);
            count++;
        }
        return count;
    }
}

```

338. 比特位计数 Easy

给你一个整数 `n`，对于 `0 <= i <= n` 中的每个 `i`，计算其二进制表示中 `1` 的个数，返回一个长度为 `n + 1` 的数组 `ans` 作为答案。

示例 1：

输入： `n = 2`
 输出： `[0,1,1]`
 解释：
`0 --> 0`
`1 --> 1`
`2 --> 10`

示例 2：

输入: n = 5
输出: [0,1,1,2,1,2]

解释:

0 --> 0
1 --> 1
2 --> 10
3 --> 11
4 --> 100
5 --> 101

提示:

- $0 \leq n \leq 105$

进阶:

- 很容易就能实现时间复杂度为 $O(n \log n)$ 的解决方案, 你可以在线性时间复杂度 $O(n)$ 内用一趟扫描解决此问题吗?
- 你能不使用任何内置函数解决此问题吗? (如, C++ 中的 `__builtin_popcount`)

```
class Solution {  
public int[] countBits(int n) {  
    int[] dp = new int[n + 1];  
    for (int i = 1; i <= n; i++) {  
        dp[i] = dp[i & (i - 1)] + 1;  
    }  
    return dp;  
}
```

```
class Solution {  
public int[] countBits(int n) {  
    int[] dp = new int[n + 1];  
    for (int i = 0; i <= n; i++) {  
        int cnt = 0;  
        int j = i;  
        while (j != 0) {  
            cnt += j & 1;  
            j = j >> 1;  
        }  
        dp[i] = cnt;  
    }  
    return dp;  
}
```

406. 根据身高重建队列 Medium

假设有打乱顺序的一群人站成一个队列，数组 `people` 表示队列中一些人的属性（不一定按顺序）。每个 `people[i] = [hi, ki]` 表示第 `i` 个人的身高为 `hi`，前面 **正好** 有 `ki` 个身高大于或等于 `hi` 的人。

请你重新构造并返回输入数组 `people` 所表示的队列。返回的队列应该格式化为数组 `queue`，其中 `queue[j] = [hj, kj]` 是队列中第 `j` 个人的属性（`queue[0]` 是排在队列前面的人）。

示例 1：

输入: `people = [[7,0],[4,4],[7,1],[5,0],[6,1],[5,2]]`

输出: `[[5,0],[7,0],[5,2],[6,1],[4,4],[7,1]]`

解释:

编号为 0 的人身高为 5，没有身高更高或者相同的人排在他前面。

编号为 1 的人身高为 7，没有身高更高或者相同的人排在他前面。

编号为 2 的人身高为 5，有 2 个身高更高或者相同的人排在他前面，即编号为 0 和 1 的人。

编号为 3 的人身高为 6，有 1 个身高更高或者相同的人排在他前面，即编号为 1 的人。

编号为 4 的人身高为 4，有 4 个身高更高或者相同的人排在他前面，即编号为 0、1、2、3 的人。

编号为 5 的人身高为 7，有 1 个身高更高或者相同的人排在他前面，即编号为 1 的人。

因此 `[[5,0],[7,0],[5,2],[6,1],[4,4],[7,1]]` 是重新构造后的队列。

示例 2：

输入: `people = [[6,0],[5,0],[4,0],[3,2],[2,2],[1,4]]`

输出: `[[4,0],[5,0],[2,2],[3,2],[1,4],[6,0]]`

提示:

- `1 <= people.length <= 2000`
- `0 <= hi <= 106`
- `0 <= ki < people.length`
- 题目数据确保队列可以被重建

```
class Solution {
    public int[][] reconstructQueue(int[][] people) {
        Arrays.sort(people, new Comparator<int[]>() {
            @Override
            public int compare(int[] o1, int[] o2) {
                if (o1[0] == o2[0])
                    return o1[1] - o2[1];
                return o2[0] - o1[0];
            }
        });
        ArrayList<int[]> res = new ArrayList<>();
        for (int[] person : people) {
            res.add(person[1], person);
        }
    }
}
```

```
    }
    return res.toArray(new int[1][]);
}
}
```

```
class Solution {
    public static int[][] reconstructQueue(int[][] people) {
        Arrays.sort(people, (int[] a, int[] b) -> (a[0] == b[0] ? a[1] - b[1] : b[0] - a[0]));
        List<int[]> res = new LinkedList<>();
        for (int[] p : people) {
            res.add(p[1], p);
        }
        return res.toArray(new int[res.size()][]);
    }
}
```

151. 翻转字符串里的单词 Medium

给你一个字符串 `s`，逐个翻转字符串中的所有 单词。

单词 是由非空格字符组成的字符串。`s` 中使用至少一个空格将字符串中的 单词 分隔开。

请你返回一个翻转 `s` 中单词顺序并用单个空格相连的字符串。

说明：

- 输入字符串 `s` 可以在前面、后面或者单词间包含多余的空格。
- 翻转后单词间应当仅用一个空格分隔。
- 翻转后的字符串中不应包含额外的空格。

示例 1：

```
输入: s = "the sky is blue"
输出: "blue is sky the"
```

示例 2：

```
输入: s = "hello world "
输出: "world hello"
解释: 输入字符串可以在前面或者后面包含多余的空格，但是翻转后的字符不能包括。
```

示例 3：

输入: s = "a good example"

输出: "example good a"

解释: 如果两个单词间有多余的空格, 将翻转后单词间的空格减少到只含一个。

示例 4:

输入: s = " Bob Loves Alice "

输出: "Alice Loves Bob"

示例 5:

输入: s = "Alice does not even like bob"

输出: "bob like even not does Alice"

提示:

- `1 <= s.length <= 104`
- `s` 包含英文大小写字母、数字和空格 ' '
- `s` 中 至少存在一个 单词

进阶:

- 请尝试使用 `O(1)` 额外空间复杂度的原地解法。

```
class Solution {  
    public String reverseWords(String s) {  
        List<String> str = Arrays.asList(s.trim().split("\\s+"));  
        Collections.reverse(str);  
        return String.join(" ", str);  
    }  
}
```

"\\s" 表示 空格, 回车, 换行等空白符

"+" 号表示一个或多个的意思

"\\s" 表示全部空格

" " 只表示单个空格, 所以不一样

538. 把二叉搜索树转换为累加树 Medium

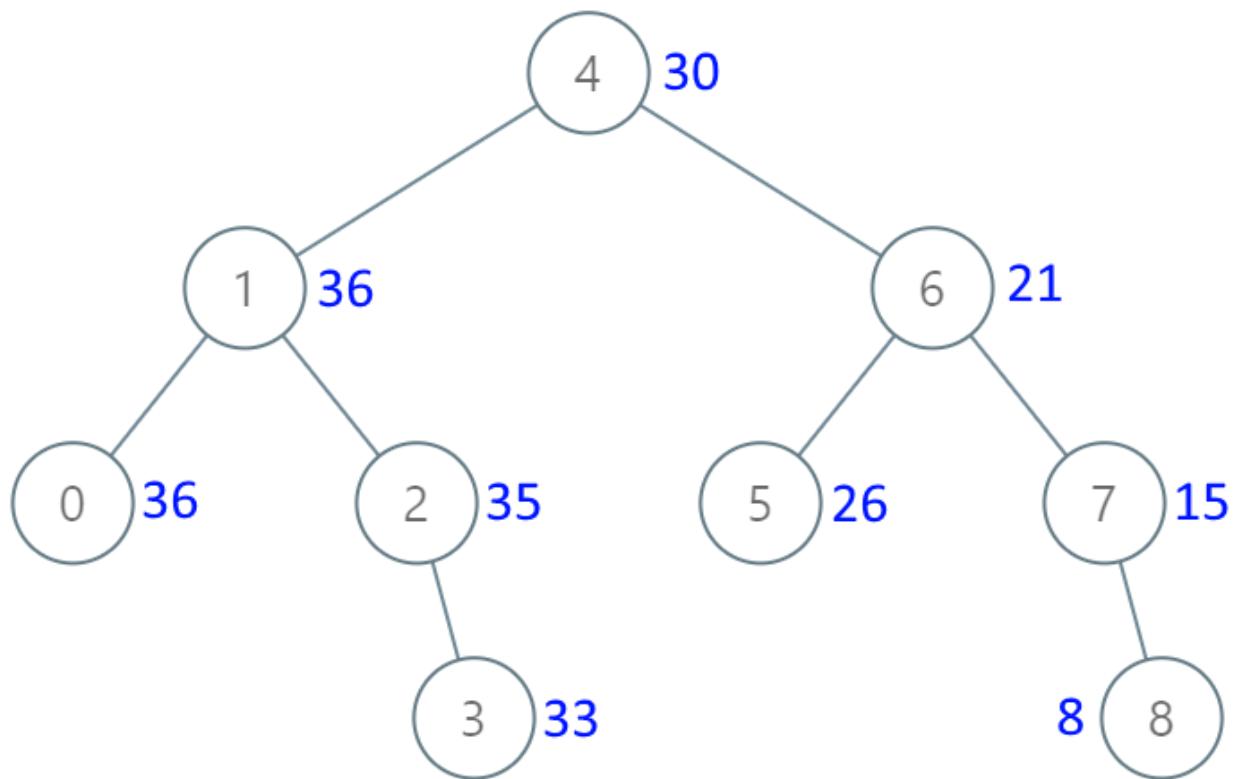
给出二叉 搜索 树的根节点, 该树的节点值各不相同, 请你将其转换为累加树 (Greater Sum Tree), 使每个节点 `node` 的新值等于原树中大于或等于 `node.val` 的值之和。

提醒一下, 二叉搜索树满足下列约束条件:

- 节点的左子树仅包含键 小于 节点键的节点。
- 节点的右子树仅包含键 大于 节点键的节点。
- 左右子树也必须是二叉搜索树。

注意：本题和 1038: <https://leetcode-cn.com/problems/binary-search-tree-to-greater-sum-tree/> 相同

示例 1：



```

输入: [4,1,6,0,2,5,7,null,null,null,3,null,null,null,8]
输出: [30,36,21,36,35,26,15,null,null,null,33,null,null,null,8]
  
```

示例 2：

```

输入: root = [0,null,1]
输出: [1,null,1]
  
```

示例 3：

```

输入: root = [1,0,2]
输出: [3,3,2]
  
```

示例 4：

```
输入: root = [3,2,4,1]
输出: [7,9,4,10]
```

提示:

- 树中的节点数介于 0 和 104 之间。
- 每个节点的值介于 -104 和 104 之间。
- 树中的所有值 互不相同。
- 给定的树为二叉搜索树。

```
class Solution {
    public TreeNode convertBST(TreeNode root) {
        Stack<TreeNode> stack = new Stack<>();
        TreeNode cur = root;
        int sum = 0;
        while (!stack.isEmpty() || cur != null) {
            while (cur != null) {
                stack.push(cur);
                cur = cur.right;
            }
            TreeNode pop = stack.pop();
            sum += pop.val;
            pop.val = sum;
            cur = pop.left;
        }
        return root;
    }
}
```

```
class Solution {
    public TreeNode convertBST(TreeNode root) {
        convertBST(root, 0);
        return root;
    }

    private int convertBST(TreeNode root, int sum) {
        if (root == null)
            return sum;
        root.val += convertBST(root.right, sum);
        return convertBST(root.left, root.val);
    }
}
```

```

class Solution {
    int sum = 0;

    public TreeNode convertBST(TreeNode root) {
        if (root == null)
            return null;
        convertBST(root.right);
        root.val += sum;
        sum = root.val;
        convertBST(root.left);
        return root;
    }
}

```

复杂度分析

- 时间复杂度: $O(n)$, 其中 n 是二叉搜索树的节点数。每一个节点恰好被遍历一次。
- 空间复杂度: $O(n)$, 为递归过程中栈的开销, 平均情况下为 $O(\log n)$, 最坏情况下树呈现链状, 为 $O(n)$ 。

还有一种方法, Morris遍历。

301. 删除无效的括号 Hard

给你一个由若干括号和字母组成的字符串 s , 删去最少数量的无效括号, 使得输入的字符串有效。

返回所有可能的结果。答案可以按 任意顺序 返回。

示例 1:

```

输入: s = "(())()"
输出: [ "(()())", "()()()"]

```

示例 2:

```

输入: s = "(a)()()"
输出: [ "(a())()", "(a)()()" ]

```

示例 3:

```

输入: s = ")()"
输出: [ "" ]

```

提示:

- $1 \leq s.length \leq 25$

- `s` 由小写英文字母以及括号 ' '(' 和 ')' 组成
- `s` 中至多含 20 个括号

```

class Solution {
    private List<String> res = new ArrayList<String>();

    public List<String> removeInvalidParentheses(String s) {
        int lremove = 0;
        int rremove = 0;

        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '(') {
                lremove++;
            } else if (s.charAt(i) == ')') {
                if (lremove == 0) {
                    rremove++;
                } else {
                    lremove--;
                }
            }
        }
        helper(s, 0, lremove, rremove);
    }

    return res;
}

private void helper(String str, int start, int lremove, int rremove) {
    if (lremove == 0 && rremove == 0) {
        if (isValid(str)) {
            res.add(str);
        }
        return;
    }

    for (int i = start; i < str.length(); i++) {
        if (i != start && str.charAt(i) == str.charAt(i - 1)) {
            continue;
        }
        // 如果剩余的字符无法满足去掉的数量要求，直接返回
        if (lremove + rremove > str.length() - i) {
            return;
        }
        // 尝试去掉一个左括号
        if (lremove > 0 && str.charAt(i) == '(') {
            helper(str.substring(0, i) + str.substring(i + 1), i, lremove - 1,
rremove);
        }
        // 尝试去掉一个右括号
    }
}

```

```
        if (rremove > 0 && str.charAt(i) == ')') {
            helper(str.substring(0, i) + str.substring(i + 1), i, lremove, rremove
- 1);
        }
    }
}

private boolean isValid(String str) {
    int cnt = 0;
    for (int i = 0; i < str.length(); i++) {
        if (str.charAt(i) == '(') {
            cnt++;
        } else if (str.charAt(i) == ')') {
            cnt--;
            if (cnt < 0) {
                return false;
            }
        }
    }
    return cnt == 0;
}
}
```

直接摘录官方解答。

```
Queue<TreeNode> queue = new LinkedList<>();
queue.offer(root);
```

root可以为null